Operating System Support for Parallel Processes

Barret Rhoden

Technical Report No. UCB/EECS-2014-223
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-223.html

December 18, 2014

Operating System Support for Parallel Processes

by

Barret Joseph Rhoden

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Eric Brewer, Chair
Professor Krste Asanović
Professor David Culler
Professor John Chuang

Fall 2014

Operating System Support for Parallel Processes

Abstract

Operating System Support for Parallel Processes

by

Barret Joseph Rhoden

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric Brewer, Chair

High-performance, parallel programs want uninterrupted access to physical resources. This characterization is true not only for traditional scientific computing, but also for high-priority data center applications that run on parallel processors. These applications require high, predictable performance and low latency, and they are important enough to warrant engineering effort at all levels of the software stack. Given the recent resurgence of interest in parallel computing as well as the increasing importance of data center applications, what changes can we make to operating system abstractions to support parallel programs?

Akaros is a research operating system designed for single-node, large-scale SMP and many-core architectures. The primary feature of Akaros is a new process abstraction called the "Many-Core Process" (MCP) that embodies transparency, application control of physical resources, and performance isolation. The MCP is built on the idea of separating cores from threads: the operating system grants spatially partitioned cores to the MCP, and the application schedules its threads on those cores. Data centers typically have a mix of high-priority applications and background batch jobs, where the demands of the high-priority application can change over time. For this reason, an important part of Akaros is the provisioning, allocation, and preemption of resources, and the MCP must be able to handle having a resource revoked at any moment.

In this work, I describe the MCP abstraction and the salient details of Akaros. I discuss how the kernel and user-level libraries work together to give an application control over its physical resources and to adapt to the revocation of cores at any time — even when the code is holding locks. I show an order of magnitude less interference for the MCP compared to Linux, more resilience to the loss of cores for an HPC application, and how a customized user-level scheduler can increase the performance of a simple webserver.

Contents

List of Figures

List of Tables

Acknowledgments

I have been extremely fortunate over the past seven years. Not only did I have the opportunity to pursue a project I found interesting, but I did so while working with excellent people.

Both my research opportunity and the group environment are the product of my advisor: Eric Brewer. I would like to thank Eric for his inspiration and guidance — especially his implicit guidance as a role model. Although research is important, it is less important than a work-life balance and being "safe in your own skin." Eric gave me the breathing room necessary to work on Akaros and allowed me to "do the right thing," even if it took longer.

I would also like to thank my committee, especially David Culler, for providing a much-needed historical perspective for my work. Additionally, David Wessel's music application was the original inspiration for the separation of provisioning and allocation in Akaros.

The bulk of my time was spent with other graduate students and developers, and my work would not have been possible without them. Kevin Klues has been my partner in crime for a number of years now. It's hard to believe that the end of our era in Berkeley is finally here. David Zhu and I have had discussions since our first semester here together. Andrew Waterman is an impressive coder who tends to know what I am thinking before I say it. Paul Pearce developed Akaros's initial network drivers, and Andrew Gallatin vastly improved the more recent Plan 9 network stack. I would be lucky to work with any of them in the future.

Ron Minnich and I met in 2009, but we did not work together until our serendipitous pairing as host and intern at Google in 2013. He was instrumental both in porting the guts of Plan 9 to Akaros and in using FTQ. Ron breathed new life into Akaros, greatly increasing its chances of having a life beyond graduate school. More importantly, Ron is the real deal — authentic and generous, and I hope to be more like him.

My family has been very supportive over the years. They have always provided a home in Pittsburgh, complete with nieces and nephews, for whenever I was in town and needed to relax.

Finally, I would like to thank my wife, Tamara Broderick. Not only did she help with FTQ and R, but she has been my biggest supporter. Whether it was coining the name "Akaros" or designing giraffe-themed merchandise, she has always been there for me, and I look forward to returning the favor.

Chapter 1

Introduction

There are limitations to what we can do with computers. System designers make trade-offs, and over time certain designs should be reevaluated — especially when there is a change in the landscape of computing. One such change is the resurgence of interest in parallel computing due to the industry's failure to increase uniprocessor performance[5]. Given this environment, what changes can we make to operating systems to support parallel programs?

Researchers and engineers have pursued parallel computing for decades. The existence of parallel computers is not new, although their ubiquity is on the rise. Another trend in computing is the creation of large-scale clusters of computers in data centers. The largest parallel machines are likely to be found in data centers. Any work on parallel computing ought to address data centers and their workloads, as well as the more traditional concerns of parallel applications.

Typically, high-performance, parallel programs are tuned and want uninterrupted access to resources; after all, the purpose of parallelism is to speed up an application. Certain data center applications also fit this description. Data center workloads tend to be a mix of low-latency, high-priority applications and batch, time-insensitive background applications[13]. Application latency is extremely important. For instance, Google found that a half second of latency in a search result lead to 20% less page views[64]. These low-latency applications are highly tuned, want uninterrupted access to resources, and usually require multiple cores: a modern parallel application.

One example of a high-performance, data center application is TritonSort. Rasmussen et al. fine-tuned a set of machines to efficiently execute a custom sorting algorithm, performing 60% better than the previous winner of the Indy GraySort sorting benchmark with only 1/6 the number of nodes[89]. Their competitive edge was in-depth knowledge of and access to the systems resources, which they carefully balanced to avoid bottlenecks.

Li et al. analyzed sources of tail latency within the data center, identifying certain optimizations and design choices that are independent of specific applications. Some of these design choices are spatially allocating dedicated cores to jobs, dedicating cores for interrupt processing, and using a "thread-per-connection" model[57]. Not only do low-latency applications require tuning and dedicated access to resources, but they also benefit from tinkering in the classic areas of parallelism — namely the management of cores and threads. These highly tuned, low-latency parallel application would benefit from customized operating system interfaces and subsystems.

An important aspect of data center workloads is the background jobs. Data centers provision for the peak load of the high-priority application. For instance, an Intuit spokesman famously said, "Every year, we take the busiest minute of the busiest hour of the busiest day and build capacity on that"[66]. When actual utilization is low, low-priority batch jobs take up the slack[13]. Unlike with classic time-sharing machines, we do not run multiple applications on a machine for fairness or for multi-tasking, but to maintain the utilization of a purchased resource.

For operating systems, there are two important details about the mix of data center workloads: multi-tenancy and variations in utilization. Not only are there multiple applications, but the resource allocations will change over time in response to changes in the workload of the high-priority application. It is not enough to simply allocate an entire machine to an application; due to changing resource requirements, those allocations may change over time and there may be multiple applications.

Historically, changing the resource allocation for competing, parallel applications on a machine can be tricky. When the application has control over dedicated resources, it also has the responsibility to manage those resources. It can be hard for the application to handle the revocation of those resources by the operating system. The operating system preempted those resources to allocate them to the high-priority application, but we still want the low-priority application to continue with its reduced set of resources.

In this work, I will present operating system abstractions to support high-performance, parallel applications. This work is done in the context of Akaros, a research operating system designed for single-node, large-scale SMP and many-core architectures. The primary feature of Akaros is a new process abstraction called the "Many-Core Process" (MCP). The MCP embodies transparency, application control of physical resources, and performance isolation. The defining feature of the MCP is the separation of cores from threads: the operating system grants spatially partitioned cores to the MCP, and the application schedules its threads on those cores. In this manner, an application has fine-grained control over its resources. An important part of the MCP is that there can be more than one of them in the operating system. Not only are they capable of managing dedicated resources, but the application can handle the preemption of those resources, even at very inopportune times.

This work is organized as follows.

- In Chapter 2, I provide background on the history of operating systems and how they dealt with threads, parallelism, and competing applications.

- In Chapter 3, I introduce the MCP and discuss the basics of Akaros, including the kernel scheduler's distinction between *provisioning* and *allocating* resources. This distinction is the central mechanism for supporting the mix of low-latency and batch workloads common in data centers, and provides applications coarse-grained control over its resources.

- In Chapter 4, I go into more detail on the MCP, describing the life of an MCP from both the kernel and the user's perspective. I outline how the system handles traditionally difficult situations caused by page faults and blocking system calls. I present microbenchmarks showing fast user-level threading and an order of magnitude less interference for an MCP's core on Akaros than for a dedicated process on Linux.

- Chapter 5 covers the event delivery system, whereby the kernel and the user work together to ensure applications never miss messages — a critically important service to enable applications to manage their cores. Although the event delivery system is particular to Akaros, its low-level data structures such as the *Unbounded Concurrent Queue* can be used by any shared-memory operating system.

- In Chapter 6, I describe how the kernel preempts cores and how the application recovers from losing its cores in any situation. The difficulty of preemption recovery comes from the application losing its tools mid-recovery; it is hard when the core running the recoverer is revoked. Similarly, preempted cores could have been running code holding locks. I present a class of locking algorithms called *Preemption Detection and Recovery* locks that avoid deadlock and compare the performance of PDR locks to more traditional locking methods.

- In Chapter 7, I analyze the performance of two parallel applications on Akaros: a traditional compute-bound task and a simple webserver. I show how parallel applications on Akaros can be resilient to preemption and how a customized user-level scheduler can improve performance. For instance, a simple webserver on Akaros with a customized scheduler can slightly outperform the same webserver on Linux, despite Akaros's immature network stack. Finally, I show a high-priority webserver with dynamic resource requirements running alongside a low-priority background job, and the entire system handles the preemption and reallocation of cores.

Chapter 2

Background on Operating Systems

At the most basic level, the role of an operating system is to provide an environment for the execution of a user's programs. It manages hardware, often in the form of device initialization and access. It provides abstractions for accessing hardware, including memory, and bootstraps process execution. In many systems, it protects processes from one another and multiplexes resources among them.

Alan Bawden from MIT's Incompatible Time Sharing (ITS) operating system said:

> The basic job of any time-sharing operating system is to share a single processor among a set of independent processes[8].

In that regard, the basic job of any shared, parallel operating system is to share a single multiprocessor among a set of independent, parallel processes.

To fully understand the changes I propose for operating systems to support parallel processes, we need to look back at the design decisions of operating systems over the years. How do programs make requests of the operating system, especially requests that might complete asynchronously? What are the process abstractions presented to programmers, especially related to parallelism and concurrency? How does the operating system deal with multiple competing parallel applications, where a process might not have the entire machine? Parallel program performance is sensitive to interference; how do operating systems try to maximize these applications' performance?

2.1 System Calls

Operating systems provide a variety of methods to service user requests, called *syscalls*. The details of this kernel interface has many implications on the performance and programmability of all programs. The trade-offs in this space have revolved around a few points: avoiding syscall mechanism overheads, exploiting cache locality and avoiding context-switch interference, and dealing with the asynchronous nature of devices and events.

The two main components to a syscall are the request message and the control transfer mechanism to the kernel. The details of the message Application Binary Interface (ABI) are rather trivial: certain arguments are placed in certain registers, or there is a specific layout in memory of the arguments. How the message is delivered depends on the nature of the control transfer. For many systems, a user program makes a hardware-assisted procedure call into the operating system via a trap. *Call Gates* between *rings* of privilege of this sort have been in use for decades, since Multics[95]. The details may have changed, such as how x86 platforms have the `syscall` instruction for fast transfers from Ring 3 (lower privilege) to Ring 0 (higher privilege), but the essence is the same.

Most operating systems today are either Windows or based on Unix (e.g. Linux, the BSDs, OS X), and employ a trap-based syscall mechanism. In recent years, researchers have experimented with other approaches to getting syscall messages to the kernel, taking advantage of multicore processors. Instead of trapping into the kernel, which is an expensive operation, an application can place a syscall message in shared memory and the kernel can run a syscall handler on another physical core. Instead of transferring control of the current processor, which is a form of *time-sharing* with the kernel, running a syscall on a remote core is a form of *space-sharing*.

For example, Kumar et al.[50] employed a similar mechanism for virtual machine monitors (VMMs) and coined it “sidecore”. Ring transfers from guest VMs to VMMs on x86 platforms are even more expensive than regular syscalls, hence their motivation. Soares and Stumm implemented an asynchronous syscall interface in Linux, based on shared memory regions, in FlexSC[99]. An application could trap on the local processor to transfer control, or kernel threads could run remotely in the sidecore style. Their motivation was avoiding control-transfer overheads and to exploit cache locality by minimizing context switches. FlexSC syscalls could run on a remote core or locally, where local syscalls amortize the trap overhead by batching numerous syscalls together. The AsyMOS[77] dedicated cores to devices, designed for network processing. They were less concerned with avoiding control transfer overheads than with utilizing multiprocessors for TCP processing. At the time, offloading TCP to network cards with specialized processors was popular, and the ideas behind AsyMOS would eventually be part of the “TCP onload” discussion[90]. On a humorous note, depending on when you are reading this, either TCP offload or onload may be more popular; this was described to me by a networking professional who had been around for a while as the offloading “Circle of Karma”. Brecht et al. developed a network processing architecture called ETA-AIO (Embedded Transport Acceleration with Asynchronous I/O))[12]. The “ETA” was the dedicated “sidecore”, and the AIO was a set of APIs for posting syscalls and receiving completion events.

Although remote syscalls on “sidecores” have been gaining popularity as multiprocessors become prevalent, remote syscalls are much older. Back in the 1960s, the Control Data 6600[107] had remote “cores” for I/O and system control, which was in essence their system calls. All control and I/O were performed on ten peripheral processors around a central, more powerful processor. They would typically run the main OS on one peripheral and run device drivers on the others. Their motivation was to keep the central CPU and its FPU free from interference from the peripheral devices. This is analogous to modern systems using remote cores for syscalls to increase performance by exploiting cache locality of the user and kernel programs and avoiding context-switching interference.

Remote syscalls have some performance benefits at the expense of complexity and load balancing issues. A single syscall-processing core can become a bottleneck under load and can be a waste when under-utilized. Dynamically adjusting these cores may incur latency penalties. The correct choice is workload dependent, and ideally an operating system could support either approach.

Regardless of whether remote or local syscalls are a better approach for a given workload, a larger issue is that of asynchrony: what do operating systems do when syscalls block, such as on a disk read or a network operation? Performance lost due to cache misses and trap overheads are orders of magnitude smaller than the waiting time for many I/O operations, and that time could be spent by the application performing other computations. The application might not have more work to do, but the operating system does not know. The real issue is one of control: if an application loses access to the CPU because of issuing a blocking syscall, its performance will suffer needlessly.

Most operating systems, even those on single-core processors, need to deal with blocking syscalls. The obvious answer for many systems is to have asynchronous options on their syscalls to avoid blocking, and then allow programs to request an event or poll for completion. For instance, POSIX sockets can set the `O_NONBLOCK` option, and if a socket operation would block, the kernel returns an error code. Then, applications can use syscalls such as `epoll()` or the older `select()` to poll if file descriptors are available for I/O and optionally block until they are available. Likewise, the POSIX standard has a set of AIO calls, such as `aio_read()`, that perform operations similar to their synchronous counterparts (e.g. `read()`). Unfortunately, these interfaces are clunky at best: a program needs to know if it working with a socket file descriptor or a file-system file descriptor to perform its asynchronous I/O. Furthermore, certain operations do not have non-blocking variants; for example, there is no `aio_open()`.

Although asynchronous I/O options in recent OSs tend to be bolted on to existing interfaces, asynchronous I/O interfaces are much older. Digital Equipment Corporation's (DEC) VAX/VMS system, first introduced in the late 1970s[79], had a `QIO` syscall, which queues an I/O command to a particular device[38]. Applications can later wait on the completion or even receive an interrupt or signal called an *Asynchronous System Trap*[31]. Nearly a decade earlier, the Livermore Time-Sharing System, which ran on the CDC 7600, also had asynchronous I/O calls. A user could wait until I/O completed or receive interrupts on completion or errors[102]. These early developers were merely dealing with the asynchronous nature of devices and events, and exposed that model to their programmers. Fundamentally, they did not want programs to lose access to a processor merely because an I/O operation blocked.

If operating systems had suitable asynchronous I/O decades ago, why then do we have lousy AIO interfaces today? The answer is due in part to the threading model; multiple synchronous threads can hide the latency and asynchrony of underlying hardware events. The programmability and overhead of using multiple processes or threads determines the relative importance of asynchronous I/O interfaces. McJones and Swart, from DEC's Topaz system, believed fast threads are a replacement for the "ad hoc" multiplexing techniques common to AIO: polling, waiting, and interrupting[67]. With slower threads and heavyweight processes for I/O, programmers will want lighter-weight AIO services. There is a design choice between asynchronous and synchronous kernel interfaces, but to understand the choice more we will need more background on how operating systems deal with threads and concurrency.

2.2 Threads, Concurrency, and Parallelism

Threads, at their most basic level, are independent streams of instructions within a program. Threads belonging to the same process execute in the process's address space, and they share file descriptors and other process state. Catanzaro[15] outlined many benefits to using threads.

- Hardware parallelism: a single program with threads can easily utilize multiple processors.
- I/O throughput: each thread can be used for distinct, blocking, I/O operations, allowing multiple outstanding I/Os and computation during I/Os.
- Efficiency: threads require less memory than full processes.
- Programming ease: threads make programming asynchronous actions easy.

Yet, as we shall see, the specific benefits of threads depend on the details of the threading model and vary from system to system. To fully understand threads, let us step back and review the original notion of a single-threaded process and explore how threads evolved over the years.

In traditional processes, there is only one thread, and it is simply the code that executes a series of functions, e.g. `int main()`, on a stack whose memory is readable and writable by the process in user mode. The functions eventually make system calls or otherwise trap into the kernel, and the kernel executes functions, e.g. `sys_read()`, on behalf of the user. When the code switches into the kernel, the hardware or the kernel software changes to another stack: one that is neither readable nor writable by the user. The original registers and context from the user program are saved and restored, subject to some interface rules. The registers and other aspects of the context of code executing in the kernel is independent of the user's context. This separate stack and context is the kernel thread, sometimes phrased as the "process executing in kernel" or the "kernel executing in process context", as opposed to interrupt handler (IRQ) context. For each process, the kernel maintains a thread control block and both the user and kernel stacks. There is no distinction between user threads and kernel threads: there is only one thread, visible to the kernel, which can execute in either in the kernel or in userspace. Parallel applications were built by spawning several of these processes, which communicated with some form of IPC, such as shared memory.

When discussing threading models, an important distinction to make is the one between *parallelism* and *concurrency*. Parallelism is rather simply defined as sequences of instructions executing at the same time, e.g. two threads running on two separate processors simultaneously. I view concurrency in two different ways, depending on the programming scenario. When attempting to exploit concurrency for performance, concurrency is merely the number of independent tasks in a program or system, e.g. one task per connection in a webserver. When programming defensively, concurrency refers to any possible interleaving of instruction streams, even those that do not occur simultaneously. For example, on a single processor, context-switched tasks are concurrent, and kernel code and interrupt handlers are concurrent. David Black, who worked on Mach, tackled this distinction by defining *application parallelism* as the actual execution in parallel achieved by a program, while *application concurrency* is the amount of parallel execution achievable given infinite processors[11]. Black's view of application concurrency matches my exploitative view on concurrency. The interleaving view of concurrency is more closely matched by Ed Lee's view: the next state of the system is determined by the next atomic action of either concurrent thread[55]. In either view, parallelism is a subset of concurrency.

Threads provide a convenient way of dealing with both aspects of concurrency. They are an easy-to-use, relatively lightweight abstraction for exploiting both processor and device parallelism. They neatly encapsulate state (e.g. per-thread variables and stacks), which reduces the amount of interleavings a programmer needs to track. Private state helps programmers reason about concurrency by ignoring operations on non-global variables: interleaved operations on independent, private state can be treated as no-ops on global state.

Of course, threads need to communicate with each other, and there are a variety of tools that provide synchronized access. One notable tool is the monitor, which is a shared-memory, software module that encapsulates all aspects of sharing: synchronization, the shared data, and the code that accesses the data[34, 40]. When programmers use monitors or similar tools with threads, programmers can effectively partition their concurrency concerns from thread-local processing. Put another way, when designing a program, it is best to start with threads operating on private state, and to use synchronization tools to protect access to any necessary global state.

The Early Days of Threading

Monitors are a useful tool, and it is no surprise that the development of monitors as a language construct went hand-in-hand with the earliest languages and systems that used threads. One of the earliest languages with support for monitors was Concurrent Pascal[35]. Soon after, Xerox developed the language Mesa[53] in the 1970s, which was used for the Pilot OS, its applications, and other projects. Mesa was designed for Pilot, to some extent, and was driven by the need to support varying amounts of inexpensive processes. These processes were "lightweight" processes, sharing the same address space: a thread. Complementing monitors, Mesa also had language primitives for spawning processes: `fork` and `join`. Mesa was superceded by Cedar[52, 104], and its ideas lived on in many languages, including Java. In more recent times, the Go language and runtime[106] has first-class support for threading and synchronization. Instead of monitors, Go uses channels and mutexes for synchronization, and it can spawn a thread with the keyword `go`.

The successor to the Cedar model of threads was Digital Equipment Corporation's (DEC) Topaz system in the 1980s[105, 67]. At that time, Unix-like systems, such as Mach[1] and Sun were adding on multithreaded capabilities. Topaz was a mix of Unix and Cedar: Cedar threads ran in a BSD 4.2-like environment, where processes could utilize either interface[67]. For synchronization, the system provided semaphores, condition variables, and mutexes, inspired by Hoare[40] and Mesa monitors. Topaz was the overall system, which included both the Taos kernel and a userspace component. There were two Topaz implementations, one native to the Firefly multiprocessor[105] and the other on Ultrix, DEC's Unix variant based on BSD 4.2[109]. In either implementation, there are two types of processes (called *address spaces* in Topaz): a legacy mode capable of running Ultrix binaries in a single-threaded manner and a Topaz mode that extends the Ultrix mode with the ability to support multiple threads[105]. The purpose of these multiple threads was not merely to exploit hardware parallelism. Existing Unix processes could coordinate and share memory, but these are loosely coupled and expensive. Topaz's threads are tightly coupled, within a single program, and are designed for light-weight control for use in Mesa-inspired synchronization[67].

The importance of Topaz was that it not only brought multithreading to the Unix world, as others had done, but that it merged the two worlds of Unix and Cedar and attempted to improve the former with the latter. There are many single-threaded assumptions built into the Unix interface that need to be either redesigned or carefully avoided. For instance, the error variable `errno` was classically process-wide. One solution would be to make it per-thread; Topaz opted to use exceptions instead. Care must be taken when calling `chdir()`, since it affects the entire process. Similarly, `seek()` affects all users of a file descriptor (FD). In both cases, Topaz parameterized the system calls using paths or FDs to take a thread-local parameter, e.g. an `open()` call took a directory handle and an `fread()` call took a file position pointer[67].

Unfortunately, many of their changes never impacted the current Unix interface, as embodied in Linux. One such deficiency is the Unix assumption about file descriptors being given out in a particular order, which is used during process creation. As someone who also attempts to bring lightweight threading to the modern POSIX community, I found their comments about this poor design both humorous and disheartening: nearly 30 years later I had a bug related to that very assumption. Other changes have been picked up or independently developed. Notably, Topaz did not `fork()` processes as in the Unix fork-exec model. It is unclear how to treat multiple threads during a `fork()`. Instead, Topaz's `StartProcess` method took parameters, such as which open file descriptors to pass to the child, with which one could achieve the semantics of the fork-exec model[67]. The V Distributed System[17], from the same era as Topaz, also did not `fork()`, and Go wisely did not build a `fork()` call into its runtime.

Topaz was one system exploring the nature of multithreading in the 1980s, but there were others and their collective development led to different models of threading. Mach's[1] designed kernel-based multithreading because Unix processes were expensive, but also to spread the existing user-level threading to multiple processors. In Mach terms, an overall process or Topaz address space was called a *task*. The V system also had multiple threads in an address space. V threads were called *processes*, and the collection of threads into an address space was called a *team*[17]. All of these early threading systems, Mach, V, Topaz, even Mesa/Cedar used kernel threads, i.e. threads managed by the kernel. In contrast, there existed user-level concurrency/threading packages, which multiplexed lightweight, user threads on top of a single kernel thread, unbeknownst to the kernel. User-level concurrency goes as far back as the early 1960s with coroutines[18], and has existed in various packages throughout the years, such as Green Threads in Java[46]. At this point in history, there existed two models of threading, user and kernel, and the stage was ripe for a third.

Threading Models

There are three well-known models of threading, with the prevailing trait being the relationship between user threads and kernel threads. In all cases, all of the threads run in the same address space.

- Many-to-One (M:1): Many user threads running on one kernel thread. These are threads that operate cooperatively, but on top of a single kernel task/process, and are scheduled by the application. This is the model still used today in some language runtime virtual machines, as well as in Capriccio[9] back in the early 2000s.
- One-to-One (1:1): Multiple pairs of one user thread running on one kernel thread. They are lighter-weight than running multiple, full-fledged kernel processes. This is the model used by Cedar, Topaz, Mach, V, and many modern OSs today, such as Linux and BSD.
- Many-to-Many (M:N): Many user threads running on many kernel threads. The user threads are scheduled by the application, and the kernel threads are scheduled by the kernel. This model was popularized in the early 1990s by Scheduler Activations[2] and Sun's Lightweight Processes (LWPs)[15], and is used today in the Go language runtime[106].

The M:1 threading model is built on a traditional single-threaded process, where M threads are multiplexed on the single kernel thread. The kernel is unaware of the M threads; from its perspective, there is only one thread in the process. In general, the kernel is only ever aware of its own threads. The M threads are in userspace, where userspace manages their stacks, control blocks, and scheduling. The benefits of having userspace manage its threads are:

- Faster thread operations: context switches and synchronization do not require switching into the kernel.
- Minimal features per thread: not all applications need all features provided by kernel threads, such as thread-local storage (TLS) and signals.
- Less memory consumed per thread: there is no need for a kernel stack, and userspace can use customized, smaller stacks for its own threads.
- Applications can decide which thread runs, instead of the operating system.

The downsides to M:1 threading are due to the process having only one kernel thread:

- When one of the user threads makes a blocking call, such as `write()`, the underlying kernel thread blocks. Since the kernel thread is blocked, userspace cannot run any other user threads. The common way to deal with this is some form of asynchronous I/O, which leads to the complex, ad-hoc, programming techniques frowned on by McJones and Swart[67].
- Page faults cause the entire program to halt while the faults are serviced. Programmers can pin or lock virtual memory to prevent page faults, but this is expensive and limits the ability to perform memory-mapped I/O.
- The single kernel thread, and thus all M user threads, can only run on one processor at a time and is ill-suited to SMP systems.

Capriccio[9] was a relatively recent M:1 threading system from 2003, designed for scalable web services using a thread-per-request model. Capriccio interposed on the common I/O calls, such as `read()`, replacing them with wrappers around Linux asynchronous I/O calls. Asynchronous calls alleviated the major concern of M:1 threading: system calls no longer result in the entire program halting, and Capriccio hid the complexity behind the threading library. Additionally, Capriccio developed a linked-stacks technique whereby each thread only used a small amount of stack space at a time, increasing the stack space on demand[9]. Capriccio threads required less resources and had faster context switches than traditional kernel threads, all multiplexed on one kernel thread. Although Capriccio's use of asynchronous I/O alleviated one limitation of M:1 threading, it still suffered from the others, especially its inability to use multiple processors.

By comparison, 1:1 threading models are very simple: they are merely a collection of traditional kernel threads that share the same address space and other process-wide structures, such as the file descriptor table. The benefits of the 1:1 model are the benefits of kernel threading:

- I/O concurrency: since the kernel knows about threads, each thread can page fault or block on I/O independently. This allows multiple concurrent I/Os.

- Hardware parallelism: each kernel thread can run on a different processor.

Kernel threads in any threading model are used for both I/O concurrency and parallelism. One issue with this dual purpose of kernel threads is determining how many kernel threads to create, given a number of processors and a workload. Too many threads is wasteful and can result in inter-thread interference. Too few threads will fail to utilize all of the processors. The common practice for determining the number of threads to use in the 1:1 model is to use heuristics such as "the number of processors plus two".

By the early 1990s, the benefits of both kernel threads and user threads were known, and three independent projects developed the M:N threading model that combines the benefits of the M:1 and the 1:1 models. In the M:N model, userspace maintains M threads on top of a collection of N kernel threads. A blocking operation on one kernel thread does not block all of the threads. The M:N model can exploit I/O concurrency and hardware parallelism, like in the 1:1 model. User thread context switches and synchronization are fast, and the userspace picks which threads run, like in the M:1 model. M:N threading models were developed by Sun's Lightweight Processes (LWPs)[15], Scheduler Activations[2], and Psyche[61].

SunOS Multithreading Architecture

As part of the Solaris operating environment, version 2.2, Sun supported an M:N threading model called the SunOS Multithreading Architecture[15]. Applications were programmed using user threads with an API similar to POSIX threads, with extended features to control the underlying kernel threads. The kernel threads were called *Lightweight Processes* (LWPs), and the environment was an extension to Unix SVR4. The threading architecture included the user-level runtime that scheduled the user threads on the LWPs and controlled the number of LWPs in the system. When user threads run, they are associated with a particular running LWP, which the kernel scheduled on a particular processor. Any blocking calls or page faults are handled by the underlying LWP.

The Solaris Threads library had a few features for controlling the relationship between user threads and LWPs, as well as the number of LWPs in the system. In general, user threads were scheduled based on priority, just like processes in Unix. However, application programmers could *bind* user threads to dedicated LWPs for higher performance, and the threads library could ask the kernel to bind an LWP to a processor[15]. Binding threads to LWPs is similar to pinning virtual memory to physical memory: high performance, but at a cost in developer time and potentially wasted resources.

As with other M:N system, Solaris's LWPs exist to gain the benefits of kernel threads: I/O concurrency and hardware parallelism. To that end, there were mechanisms to ensure there were enough LWPs to satisfy the application. When all LWPs are blocked in the kernel, the kernel would send a SIGWAITING signal to the application. If there were more user threads than LWPs, the runtime would have installed a SIGWAITING handler, so that the runtime would know it has more work to do and request new LWPs. Additionally, the application could call `thr_set_concurrency()`, which instructs the runtime to maintain a set amount of LWPs. The amount of concurrency is actually both the I/O concurrency and the hardware parallelism; equivalent to the old heuristics used in 1:1 threading. If there are more LWPs than user threads, the runtime would park the LWPs on a synchronization variable and eventually free them over time[15].

Fundamentally, Solaris Threads wanted more than to just merge the benefits of kernel threads and user threads. They wanted programs to not lose access to a processor merely because an I/O operation blocked. Hence the SIGWAITING and `thr_set_concurrency()` features, to ensure there are enough LWPs, given the programmer picked the correct value for *concurrency*. A relatively minor enhancement that may or may not have made it into future versions of Solaris Threads would be to have the runtime *always* handle SIGWAITING and have the kernel always send SIGWAITING when any LWP blocked. Another issue Solaris Threads did not address is what they do if there are more LWPs than processors granted by the operating system. Applications could request the binding of an LWP to a processor, but otherwise the application and runtime were blind to the actual scheduling of LWPs. When LWPs are oversubscribed, the kernel time-slices the kernel threads, which inherently determines which user thread runs. It fell on another M:N system, Scheduler Activations, to explore this issue further.

Scheduler Activations

Scheduler Activations[2] was another M:N system that merged the best of kernel and user threading. Unlike LWPs, which evolved from the Unix world, Scheduler Activations came from the Topaz world. The first implementation ran on the Firefly, and the system was built on slight modifications to Topaz's kernel threads. More important than being an M:N system, they tried to change the processing abstraction. Anderson et al. argued that kernel threads were not the correct abstraction, and that applications should be given a *virtual multiprocessor*. To that end, programs were aware of how many physical cores they were granted and were able to control what ran on its cores. The vehicle for running user code on processor or core was called the *scheduler activation*, which is roughly analogous to a kernel thread. It is important to note that the use of something very similar to kernel threads is not inconsistent with the opinion that kernel threads are not the correct abstraction. Normally, kernel threads block and resume without notifying applications, and they are scheduled without the application's knowledge[2]. Scheduler activations are kernel threads without those limitations, similar to what could be done with LWPs with some modifications, including more types of signals.

A scheduler activation is actually both a kernel thread, for the execution of user threads and syscalls, and a vehicle for event delivery[2]. There were two major aspects to this work: one is the exposing of events relevant to scheduling, and the other is the mechanism for delivering these events. Generic event delivery works like so:

1. User threads are bound to activations (i.e. kernel threads) when they run.

2. When the kernel needs to deliver an event, it picks a core running the address space (i.e. process) and stops the activation running on that core.

3. The kernel allocates a new activation, preferably from a pool of recently used activations for better performance.

4. Both the original event and the extra, stopped activation's context are bundled into an event payload, which is attached to the freshly allocated activation.

5. The new activation starts at the runtime entry point and processes the event and handles the previously stopped activation.

A common event is triggered when a kernel thread blocks on a syscall in the kernel or when a user thread page faults. In this case, the kernel picks the current core in Step 2, and the event payload does not have an extra, stopped activation; there is just the "blocked activation" event.

The scheduling events are limited to a small set of events relevant to scheduling. These events were designed such that the kernel can maintain a basic invariant critical to the virtual multiprocessor abstraction: there are always exactly as many running activations as their are processors allocated to the process[2].

- New processor: no event, the runtime is starting fresh on a processor.
- Activation has blocked: event payload is the activation ID, no context needed.
- Activation has unblocked: event payload is the ID of the unblocked activation and the context of the previously running activation.
- Activation has been preempted: event payload is the ID and the context of the preempted activation.

Of special note, when an activation unblocks in the kernel and the system call is complete, the kernel sends an event in lieu of blindly restarting the kernel thread[2]. Solaris threads, by contrast, would restart the LWP. By letting userspace control the unblocking of a system call, the Scheduler Activations kernel both avoids oversubscribing activations on processors and allows the application to pick which thread runs on which core. Any other alternative would require the kernel to choose which activation to run, e.g. time-slicing, and there is no guarantee the kernel would make a good choice.

For every event delivered to the user-level runtime, the kernel allocates an activation, and this activation is the running kernel thread until the next event. Once the application has processed the payload, it must tell the kernel the activation is free. In the case of an unblocked activation, the runtime extracts the user thread's context and eventually tells the kernel the original activation can be reused. The application can release old activations in a batch. Additionally, userspace can make explicit requests to change its processor allocation, either to acquire more or yield existing processors[2].

Scheduler activation event delivery is elegant, but it also has many drawbacks. The limited nature of events is a minor annoyance and one that can easily be extended. However, userspace has no control over the manner of event delivery. For instance, the kernel interrupts a processor to send an event: polling is not an option. Given that activations are the vehicle for event delivery, it is unclear how polling could ever be an option. Furthermore, the kernel picks the processor to interrupt; userspace has no way to communicate its preferences. Imagine a process running on a single core and the kernel sends a stream of "unblock" events: the first event will spawn a fresh activation with the event and a reference to the previous activation. The second event will spawn another activation with the second event and a reference to the activation processing the first event, and so forth. All events are treated the same, are unmaskable, and *chain* together. An application cannot mask interrupts and poll for events, such as many kernel networking subsystems do. Finally, each event requires the allocation, even if from a pool, of a full activation, an interrupt, and the saving/restoring of a hardware context.

Events can occur at any time, including during critical sections. Scheduler Activations has various techniques to detect and recover from preempted critical sections, discussed further in Section 2.4. The basic idea is that when an event payload's "previously running activation" was in a critical section, the runtime immediately restarts that context in such a manner that at the end of the critical section, that context switches back to the event handler[2]. Although this suffices for many circumstances, it does not prevent all deadlocks, which I show below. At a more basic level, the entire event handling part of the runtime is considered a critical section.

These issues with event delivery and handling of critical sections are rooted in a more fundamental issue. One hint at this problem is from the previous paragraph; the event handling runtime is a critical section, as it should be, but it should be more. Although Scheduler Activations was correct in providing a virtual multiprocessor abstraction, its drawback was in using a collection of arbitrary kernel threads to create the abstraction. Scheduler Activations lacks a formal *virtual core* abstraction to build the *virtual multiprocessor* abstraction. As a basic example, how could a user-level scheduler use per-core runqueues? What does that even mean in the Scheduler Activations world; are we talking about hardware cores, numbered based on the number of cores in the computer? If the runtime uses per-physical-core runqueues, what happens when those cores are allocated to other processes, and the activations migrated somewhere else in the system? More importantly, a virtual core abstraction, with appropriate rules and code, could serve as a layer of indirection in which events can be masked, handled, and redirected. Further, a virtual core only needs to be created once per physical core in the system, and events could require only modest amounts of memory.

Psyche

A virtual processor abstraction similar to what I describe was first developed as part of the Psyche Parallel Operating System, which was also developed in the early 1990s, and ran on the BBN Butterfly Plus[61]. The primary feature of Psyche's virtual processor was its software interrupt handling, which was designed to mirror the operations of interrupts in hardware. When the kernel wanted to send an event, it looked up a software interrupt descriptor table in the user's memory. If software interrupts were unmasked, the kernel would perform the software interrupt by pushing the existing context onto the user's interrupt stack, pushing some event-specific information, and then jumping to the handler, in user mode. When in an interrupt handler, other software interrupts are disabled; further interrupts are queued. When a handler completes, it unmasks interrupts and if there are queued interrupts, userspace makes a syscall asking the kernel to push the next interrupt. Otherwise, the handler or the user-level runtime can return to the original thread or do whatever the application wants[61].

It is instructive to compare and contrast the two styles of Scheduler Activations and Psyche, both of which are solving similar problems: how to deliver events to userspace. In both cases, the events are related to the management of user-level scheduling, but the methods of event delivery differ. Delivering one event is easy; the difficulty lies with determining how to deliver another event while the first is being handled. Scheduler Activations creates a chain of event-handling activations, and has little flexibility over how to handle them. All events are delivered, and a full activation exists for each outstanding event. Psyche masks new events, and handles them one at a time at the user's discretion. New events queue in the kernel, at a smaller cost in memory. Psyche relied on trapping in to the kernel to get its next event, and did not have a notion of interrupt priority, but that could easily be changed. Those minor shortcomings are not due to the model of masking software interrupts. One of the contributions of my work is Akaros's event delivery system, which allows the user to handle multiple events at their discretion from within a runtime with a similar notion of Psyche's masking of software interrupts. This flexibility is gained by separating the physical interrupt, i.e. jumping into the runtime, from the event delivery mechanisms. Both Psyche and Scheduler Activation suffer in this regard; for instance, neither of them allow for the polling of events. Ultimately, the virtual processor model is lower level and more flexible, and ultimately may be suitable to build the Scheduler Activations model in much the same way as how one can build synchronous syscalls on an asynchronous interface.

Psyche had many other relevant features. Of special importance was a shared memory region for communication between the kernel and user. One section was read-only for the user, and another was read-write. The read-only section was actually per physical processor, mapped into the same location in all address spaces. This read-only section consisted of a few pseudo-registers with information and pointers to structures in the read-write space, which is where the virtual processor structure was located[61]. The read-write part was the most important; it provided a way for userspace to asynchronously and quickly pass information to the kernel. For instance, the interrupt descriptor table and flags for disabling software interrupts were located in this memory region. The runtime could quickly change these values without syscalls or otherwise involving the kernel. If, for a period of time, the kernel did not need the information in the read-write region, it simply does not need to check; the user merely sets the values such that they are present for whenever the kernel happens to look in the future. This was a wide, yet extremely powerful interface.

The shared memory interface was used for more than just communication with the kernel; it was also a standardized location for specific scheduling data structures. Psyche was designed to work with multiple threading libraries, simultaneously. The shared-memory region provided function pointers and data structures for common scheduler operations[61]. Even though the kernel did not need to see this information, it was very convenient. This region also contained the thread pointer; one of Psyche's goals was to elevate the user thread to a first-class status in their system. It is unclear how much the Psyche kernel knew about a user thread: pointers for identification is convenient, but in my opinion exposing details of the actual thread structure is excessive. By contrast, Scheduler Activations and Solaris Threads did not expose information about the user threads to the kernel.

Although Psyche's virtual processor abstraction was powerful, it had some limitations. First, it was not part of a larger virtual multiprocessor; the virtual processors were largely independent. Actually, every virtual processor was pinned to a physical processor, forever. Multiple virtual processors could exist for a given physical processor, usually from separate applications. All virtual processors on a physical processor were time-sliced, and they were not coordinated with other virtual processors in the same address space (process). Although this setup was beneficial for cache and memory locality (the BBN Butterfly was a NUMA machine), the lack of virtual processor migration could be problematic. Even more so, by not coscheduling the virtual processors, spin-based synchronization could suffer.

As with all M:N systems, Psyche needed to deal with the preemption of virtual processors, including the preemption of lock holders. Their approach was to issue a "two minute warning", by which the kernel would set a flag or send an interrupt when a virtual processor had an impending preemption.The user would poll this flag and would yield. If the warning is suitably long, this would prevent the application from suffering from an inopportune preemption. The default time for the warning was 3 ms (the two minute warning is a phrase from football). The 3 ms timeout was on a 16 MHz system; relative to current processor speeds, the timeout was more like a few microseconds. If a virtual processor failed to yield to the warning, it would be preempted and would wait until the next time-slice.

This scheduling limitation meant that one could not be both safe from deadlock due to preemption and fully dedicate cores to processes, dynamically. Avoiding a deadlock in Psyche requires the time-slicing of a physical processor, which is incompatible with dedicating the processor. Although it is possible to simply not allow applications to create virtual processors on specific physical processors in the first place, such a system would not be able to take advantage of the dynamic resource requirements of parallel programs. It could not safely allocate a temporarily unused resource, due to being unable to permanently revoke that resource at a later time.

Ultimately, Psyche did well with the virtual processor abstraction, but would have benefited from a virtual multiprocessor. For Scheduler Activations, the opposite is true.

Implications of M:N Threading Models

One thing both systems had in common was the ability to perform some sort of asynchronous system call. In these cases, the critical aspect of asynchrony was that the process did not stop running on a processor because a thread's syscall blocked. For Scheduler Activations, a new activation was created when an old one blocks. For Psyche, a "thread blocked on a syscall" interrupt was sent to the virtual processor. Both had mechanisms to receive events when the blocking operation completed. Incidentally, both would have been improved by providing a polling interface to check for syscall completion.

Earlier, I mentioned that the choice between asynchronous and synchronous kernel interfaces depends on the threading model. For 1:1 models, the standard synchronous system call interface is appropriate. On the surface, any M:N user-level threading runtime seems to need some sort of asynchronous system call interface. However, Solaris Threads was M:N, yet it used a synchronous interface. The distinction is in what the "N" stands for: if it is N *kernel threads*, then a synchronous model is fine. If it is N *cores*, then an asynchronous model is necessary: a process must not lose access to the core when a syscall or user thread blocks.

What Happened to M:N Threading Models?

The M:N threading models were not the only approach to parallelism during the early 1990s. Many systems still used 1:1 threading models. For instance, Mach's Cthreads package[19] could support either an M:1 or a 1:1 model, depending on which library was linked.

By the mid-1990s, of all the M:N models, Scheduler Activations became quite popular. There were implementations for BSD[98], FreeBSD[25], NetBSD[112], and even patches for Linux[58]. At the same time, Sun continued to support its M:N threading model in Solaris. Psyche made little noticeable impact with the current operating systems.

However, as the retrofits of Scheduler Activations into existing kernels continued, and as those kernels improved their kernel threading performance, the M:N model fell out of favor. FreeBSD phased out support for Kernel-Scheduled Entities (their version of Scheduler Activations) by 2008[30], and Solaris dropped support in 2002[72]. In both cases, the kernel had a newer kthread scheduler, and the existing M:N schedulers were more difficult. The Solaris release was quite explicit that the benefits of the 1:1 model were due to implementation simplicity and code efficiency. Further, they qualified that:

> Again, this is not to say that a good implementation of the MxN model is impossible, but simply that a good 1:1 implementation is probably sufficient[72].

Likewise, Ingo Molnar, one of the Linux scheduler developers, had many similar criticisms when discussing Scheduler Activations in 2002[74]. His takeaway at the time was that if you can do fast, 1:1 kernel threading, then that is all you need. He cites various concerns with the complexity of M:N systems in the current kernel, especially a set of issues that result from having multiple user-level execution contexts on top of a single kernel-level context.

Ingo mentioned two other things that are important in the broader picture: M:N systems are workarounds for fixable problems, namely slow kernel context switches, and that M:N is only necessary for a small set of use cases[74]. The M:N models exploited obvious shortcomings in existing schedulers, but their implementations failed to keep up with the 1:1 implementations. As hardware and operating systems improved drastically, and as large scale SMPs dwindled in popularity in the desktop and small server node community, the need for M:N systems dissipated, especially considering the implementation costs. Good 1:1 implementations were sufficient.

The trend in threading models has oscillated between 1:1 and M:N, and M:N is once again on the rise. We started with full processes, then developed M:1 models. These lacked kernel support, so we developed 1:1 models. These were too slow for scheduler and synchronization operations, so we developed M:N models. M:N implementations were complex and too slow for many use cases, and 1:1 models were improved. Now in more recent times, high-performance users still want lighter-weight thread operations and the ability to choose which thread runs:

- The Go language runtime[106] operates in an M:N model where the language threads (goroutines) are multiplexed on kernel threads.
- Apple's Grand Central Dispatch (GCD)[45] is a new M:N runtime with very small, fast user-level threads, multiplexed on a workqueue of kernel managed pthreads[56].
- Microsoft added User-mode Scheduling (UMS) to recent versions of Windows[71, 88]. UMS is an M:N-like model that allows user-level context switching on a collection of kernel threads.
- Google has developed a runtime for user-controlled thread scheduling[108]. It is a 1:1 model still, but the application determines which threads run. It is a specific case of the scheduling hints provided by Mach in the 1980s, discussed further in Section 2.3.

M:N threading models may be back, fueled by the resurgence of large-scale SMP and manycore machines[4], yet any successful approach will need to learn from the lessons of previous systems. Keep this list in mind, as it directly motivates my work:

- The kernel thread needlessly couples both parallelism and I/O concurrency and should *not* be the kernel-provided abstraction. Unfortunately, many M:N models retrofitted for existing operating systems use N kernel threads, instead of N cores. A virtual multiprocessor abstraction is key, wherein the available physical cores are exposed to the application.
- User-level runtimes will manage the blocking of its threads; this requires some form of asynchronous system calls or upcalls such that control is returned to the user.

- User-level runtimes will need to handle events, many of which can lead to complicated scenarios. This requires a flexible software interrupt handling mechanism, similar to the virtual processor of Psyche.
- Since userspace is involved in the details of managing the system, there is a wide kernel interface with many rapidly changing parameters. Due to the asynchronous nature of many of these settings, in that they are "update frequently, read rarely", shared-memory regions work well for fast communication.
- Thread-specific details, such as what sort of stack to use, and operations, such as context switches, need to be in userspace for the highest performance.

2.3 Scheduling and Competition

Ideally, an application would have uninterrupted access to the entire machine. In those cases where this is not the case, we are forced to schedule and allocate a limited amount of resources among competing processes. There are a variety of techniques for scheduling parallel programs, regardless of the choice of threading model, such as coscheduling and user-level scheduling. M:N models and any system involving user-level scheduling has an additional problem: how to handle preempted cores. More broadly, resource preemption and revocation is a concern in any situation with competing applications and either fluctuating priorities or resource demands.

Coscheduling and Gang Scheduling

One of the earliest techniques in parallel program scheduling in a multiprogrammed system was based on the idea of giving the entire machine to an application, or at least pretending to when it matters. Simply put, the scheduler would run all tasks (e.g. processes, threads) for an application simultaneously. This technique was called *coscheduling*[81], later called *gang scheduling*[28]. The idea behind coscheduling is that it is beneficial to parallel programs to have all threads of the program that communicate (i.e. the working set) running in parallel. Note this applies primarily to communicating threads, meaning threads that synchronize and may wait on one another, or otherwise frequently interact. For instance, if a particular thread is busy-waiting in a spinlock, then ideally whatever it waits on should be running simultaneously. Coscheduling draws many parallels with virtual memory management: process working sets correspond to memory working sets and both types of allocation can suffer from internal and external fragmentation. Coscheduling was developed in 1982 as part of the Medusa OS for the Cm* multiprocessor[103], before threads or virtual multiprocessors were developed, and its lessons were primarily geared towards processes, but the ideas are equally applicable. Gang scheduling technically differs slightly from coscheduling in that there is no fragmentation of the processor space: either the application completely runs, or not at all. Coscheduling allowed for and controlled various types of fragmentation, which can result from any bin-packing allocation scenario.

The virtual multiprocessor abstraction inherently coschedules an application; this is the essence of Feitelson's[28] intuition for gang scheduling: to provide an environment similar to dedicated, multiprocessor hardware. Generic M:N systems, such as Solaris Threads, do not necessarily coschedule the N threads. Virtual multiprocessor systems, such as Scheduler Activations, do coschedule the N processors; otherwise a preemption event is delivered to the process.

Keep in mind that in coscheduling, an application is not necessarily given the entire machine: it is given only the amount of the machine that it needs. Arguably, it may be given only its working set for fast communication, but this distinction is not important. In this manner, coscheduling and gang scheduling are part of a much larger topic of job scheduling on parallel clusters and distributed systems. Gang scheduling is a form of spatial partitioning or space-sharing (as compared to time-sharing), which is quite common in parallel job scheduling[116, 65]. Space-sharing, simply put, is granting a resource (e.g. a processor) to a job until the job completes, as compared to periodically reallocating the resource over time.

HPC and Cluster Scheduling

Space-sharing is the norm in job scheduling in parallel clusters, typically used in High-Performance Computing (HPC). One popular open-source scheduler is Maui[47, 63], now part of the commercial Moab Cluster Suite. Maui's documentation[44] presents the basics of this class of schedulers. Users submit jobs with the estimated amount of resources and time required, such as "10 processors, 2 GB RAM, 2 hours". The cluster scheduler prioritizes jobs based on various parameters and runs the highest priority job first, until it either completes or runs out of time. For the most part, the cluster is space-shared at any given time.

Of course, when cluster jobs finish or exhaust their large quantum, their resources are reallocated to other applications, which seems like time-sharing. The main distinction between space and time-sharing is one of granularity of the time quanta involved: space sharing just context switches with large time quanta. Another distinction is that space and time-sharing differ in their method of achieving the perception of simultaneous execution. Multitasking and time-sharing are used in interactive or user-facing systems where multiple jobs share a set of processors. These jobs are time-sliced to give the illusion of simultaneous execution. In space-sharing, the jobs are actually run simultaneously on separate processors.

Cluster schedulers such as Maui have additional scheduling capabilities. One important notion is *backfilling*, which schedules jobs out of order to achieve higher utilization and throughput while respecting priorities[44]. When the scheduler creates its plan for resource allocation, inevitably there will be "holes" in the resource allocation. For example, imagine there are three high-priority jobs requiring ten processors each on a 16 processor machine, and a set of low-priority jobs that require six processors each. All jobs take one hour. When the highest priority process is running, there are six cores remaining, which are insufficient for the next highest priority programs. The scheduler will *backfill* by scheduling a lower priority job, since those processors would be unused regardless. The scheduler was able to make this decision since it knows how many resources a job requires and and for how long. Additionally, jobs often finish early, since users over-estimate the resources required; this can create more "holes" in the job plan. Maui can also be configured to preempt a job, either as part of a backfill scenario or due to changing priorities. In these cases, a job is completely suspended, checkpointed, or cancelled and later resumed or restarted[44]. Note that the job is either completely running or not: there is no partial execution; this is gang scheduling.

These HPC cluster schedulers do so due to the programming model: jobs expect all of their resources, or they do not run at all. But not all cluster schedulers behave this way. Many internet businesses run data centers with multiple jobs of different priorities, but these do not have an "all or nothing" programming approach. For instance, Google classifies many of its jobs into separate priority tiers: Tier 1 and Tier 2. The Tier 1 jobs have the highest priority, yet often do not utilize their requested resources. Using HPC parlance, Tier 2 tasks are backfilled onto these resources, until the Tier 1 job's load spikes, at which point sufficient Tier 2 tasks are stopped or killed so that the high-priority Tier 1 job gets the resources it needs[41, 20]. The distinction between Google's system and scheduler's such as Maui is that the high-priority job does not use all of its resources simultaneously; it is not gang scheduled. Where a Maui job runs on all of its requested processors, a Google Tier 1 job could run on less than it requested until an external agent determined it really does need those resources.

Both styles of cluster schedulers attempt to grant otherwise unutilized resources to lower priority jobs. There are other cluster management systems, so to speak, that did the same thing. Both Condor[59] and the Network of Workstations[3] (NOW) were clusters of commodity systems that allowed work stealing across the system. For instance, in Condor, when a workstation had no interactive user (i.e., a human at the console), the cluster manager would schedule part of a job to run on that machine. When a user was detected, the job was descheduled.

Google's system, and many others like it, has less fragmentation of resources, i.e. the "holes" in the Maui scheduler's plan are a form of external fragmentation, using terms from coscheduling and memory management. However, the lack of gang scheduling means parallel processes can suffer from the same sorts of problems that led to coscheduling in the first place: poor communication performance, among other things. This does not need to be the case. If we take a closer look at gang scheduling, we see that the reason for running all tasks of an application is that userspace neither knows what cores it runs on, nor does it have any control over these details. The important concept is that the program knows which cores are running at any particular time; gang scheduling does this by default by simply running all or nothing. One could still have the benefits of the Google-style system, namely less resource fragmentation, higher utilization, and greater flexibility, but also have the benefits of gang scheduling by exposing information and scheduling decisions to userspace.

User-Level Scheduling

Once a process knows which cores it runs on and which threads are running, or at least which threads they would like to run, the next step is allowing the process to pick which thread runs. There is a distinction between user-level *scheduling* and user-level *threading*. User-level scheduling is allowing the application to choose what runs. User-level threading includes user-level scheduling and also has fast and flexible scheduler operations, as discussed in Section 2.2. Both are often coupled in the same system, especially in the M:N systems, but this is not necessary. In fact, user-level scheduling existed in 1:1 systems long before M:N models existed.

Back in the early 1970s, the Hydra OS on the C.mmp multiprocessor[62] had a form of user-level scheduling. Hydra's kernel scheduler, called the Kernel Multiprogramming System (KMPS), had an explicit distinction between long-term scheduling, which was job selection, from short-term scheduling, which was context-switching among selected jobs. In Hydra, this was referred to as *two-level scheduling*, which is a term that frequently is used in user-level scheduling discussions, although with slightly different meanings, as I will discuss later. The user-level aspect to scheduling came in the form of *policy modules* (PMs). PMs were user-level programs that serves roles in both levels of scheduling. For short-term scheduling, PMs set process parameters for the kernel scheduler, such as priority, a mask of processors to run on, and time quantum. *Time quantum* was slightly different than modern quanta: Hydra quanta consisted of a value for the time slice, i.e. the amount of time to run, and a number of slices to run. For long-term scheduling, the PMs provided a stream of processes for the KMPS to run, and the KMPS made the fine-grained decisions of short-term scheduling. When processes exhausted their time quantum or block on PM-level semaphores, they were returned to the PM, which would make a scheduling decision[62]. For the most part, user-level scheduling in Hydra was coarse-grained, while the kernel was responsible for fine-grained decisions.

A decade after C.mmp, the V Distributed System[17] also had a form of user-level scheduling. It had a simple kernel scheduler that took directions from a dedicated scheduler process. This process would influence kernel decisions by adjusting priorities, so that the kernel would reach the decision the scheduler process desired. This technique was subset of the capabilities of Hydra/C.mmp, and otherwise unremarkable. One neat, unrelated aspect to V was that processes did not have kernel stacks. This was because the kernel had no blocking system calls: the kernel was an event-driven microkernel, and had no need for kernel threads or semaphores. With this situation, there is no need to have a particular kernel stack or kernel thread for a process; the system just needs one stack per physical processor to handle traps and syscalls.

Both Hydra and V provided a coarse-grained approach to user-level scheduling: set parameters or otherwise make long-term decisions, and let the kernel scheduler deal with fine-grained decisions. This style of scheduling is only loosely coupled to particular applications: the scheduler process typically does not know any details about the internals of an application, such as which thread is performing a high-priority operation or holding a lock. This is in contrast to other forms of user-level scheduling, such as that done by Scheduler Activations and any system with user threads, where process-specific information is used to make decisions. Both coarse-grained and fine-grained user-level scheduled are useful in their own way, but fine-grained scheduling requires tighter-coupling with the application.

Although M:N user-threading systems invariably require tightly coupled user-level threading, because the kernel does not even know about the user threads, it is possible to have tightly-coupled, application-specific threading with 1:1 systems. Mach's kernel scheduler had facilities to allow user input into scheduling, based on a `thread_switch()` primitive[11]. A thread could tell the kernel to yield the processor, much like in other systems, and more powerfully, it could tell the kernel which other thread to run in its place. Using the latter form of scheduler hint was called Handoff Scheduling. Handoff Scheduling was used heavily for message passing, to make sure the recipient would run quickly. Similarly, Google added a kernel primitive to Linux, which uses a 1:1 model, called `switch_to`, which supports their user-level scheduling[108]. The `switch_to` primitive is equivalent to Handoff Scheduling, minus any of the other hints or options provided by `thread_switch()`. Handoff Scheduling is a good example of how not all kernel threading systems lack the benefits of user-level scheduling. The reality is that there is a toolbox of various techniques, many of which can be applied to a variety of systems.

Earlier I discussed that Hydra's long-term and short-term scheduling was referred to as *two-level scheduling*. Another related form of two-level scheduling was put forward by Feitelson[27]. Instead of long-term job selection and short-term thread scheduling, his two levels of scheduling are the allocation of processors and the decision of which thread uses the processor. The slight distinction is that in Hydra's two-level scheduling, specific resources are not allocated to processes or jobs. In Feitelson's model, which is the popular meaning for two-level scheduling, resources (e.g. processors) are allocated to jobs, which then decide how to use them. Usually, the second level of scheduling is controlled by the process application, and the first level is controlled by the kernel. However, there is no reason why the user cannot be in charge of the first level as well, similar to how Hydra's policy modules control long-term job selection.

Once user-space is responsible for making fine-grained, tightly coupled scheduling decisions, there needs to be a way for multiple scheduling modules to interact. For example, libraries such as OpenMP[80] or Intel's Thread Building Blocks[91] make scheduling decisions and request resources from the kernel. However, if there are multiple such modules, which can occur when linking in multiple libraries, these schedulers do not know how to interoperate. The core issue is that they cannot cannot simply work against the raw kernel interface, even for M:N models; there needs to be one agent and interface per user. In Psyche, this approach would have been dealt with by standardizing the location of the scheduler's data structures and function pointers, so that multiple schedulers and models could be supported within a single application[61, 96]. In more recent times, the Lithe runtime[82, 83] would operate as the mediator to the kernel and allow the composition of schedulers. Lithe has the added benefit of recognizing and handling the hierarchy between user-level schedulers, where threads from one scheduler spawn a collection of threads from a potentially different scheduler.

Competition and Losing Cores

The issue Lithe solves, common to any solution of composing parallel libraries or other competing schedulers within an application, is that of having a poor understanding of the actual resources available to the library. For example, OpenMP and TBB both think they have the entire machine available to them, and when both libraries are used simultaneously, they make poor decisions. Both systems, and Lithe originally, were designed for 1:1 threading model systems, e.g. Linux, and they would spawn an excessive number of kernel threads. Lithe works as an interposition layer to prevent this over-subscription.

Lithe solves a problem of intra-process competition. In a similar manner, inter-process competition involves multiple independent processes, all of which think they have the entire machine available. For instance, the Go runtime knows how many physical cores a system has and strives to have as many kernel threads, called *machine threads*, unblocked as there are physical cores in the system. The runtime does this by pre-spawning new kernel threads before attempting a potentially-blocking syscall. The problem is that there can be multiple applications; the Go runtime should adjust the number of machine threads based on the number of cores available to the application (*virtual cores)*, not the number of total cores.

Both of these examples are the result of relatively young systems attempting to exploit parallel processors, but being built on existing POSIX thread interfaces that conveyed little information about those parallel processors. By contrast, Scheduler Activations would not have had the problems of the Go runtime, where multiple applications could think they had the entire machine to themselves.

It is important to note the dual aspects of resource preemption. From the perspective of the preemptee, the program needs to adjust to the loss of a resource. Another perspective, less commonly thought of, is from the application to which the resources will be given: it needs the resources, and any delay may violate a larger objective. Speedy preemption is important for a quick reallocation to meet the system's coarse-grained scheduling goals.

Writing a kernel interface that tells a process how many cores it has been given is a relatively easy step that would alleviate some of these concerns. However, this step is insufficient: kernel interfaces and runtimes need to handle resource revocation. The root issue is one of competition and that the nature of the competition can change over time. Not only can there be multiple applications in a system, but their relative priorities and workloads can change over time. Additionally, processes will have variations in the amount of parallelism available at any given time, and dynamically reallocating processors among these processes is important for high performance[116]. Processors that the kernel granted to one application can be preempted by the kernel and given to another application. Any user-level runtime or scheduler needs to handle these scenarios.

M:N models that were built on kernel threads on legacy kernel interfaces, such as Solaris Threads or Go, tend to have no information about preemptions. Usually this is because the kernel interface simply does not exist, and the plan is to let the kernel scheduler work it out: eventually the N cores will run. These situations are not ideal for the application's performance. Lock-holders could be preempted, high-priority threads could be waiting, and threads could be running far away from caches where their data exists. User-level schedulers like Scheduler Activations and Psyche had plans for dealing with preemption and could avoid some of these ill effects of preemption. This was somewhat a necessity of having virtual multiprocessor and processor abstractions: on occasion they could disappear. Having a powerful user-level scheduling kernel interface is both a boon and a curse: a process has more control over its processor, but it has more responsibilities. One of these is dealing with preemption.

As discussed in Section 2.2, Psyche had a "two-minute warning" event sent to the process. After a specified amount of time, say 3ms, the processor would be revoked. Psyche's plan was to have the runtime yield before that happened. If that did not happen, then it was not a big deal: eventually that virtual processor would run again on that same physical core. The drawback is that this system is incompatible with truly dedicating physical cores to an application; allowing multiple virtual processors on a single physical processor means that no single application *owns* that processor. For instance, Psyche's virtual processors could not migrate to other physical processors. The system was not built with virtual multiprocessors in mind, and thus a critical section or other important block of code could be preempted permanently and possibly deadlock.

Scheduler Activations could handle a preempted activation, as discussed in Section 2.2, by sending an event to another processor belonging to the application. That event handler would decide how to handle the preemption: specifically it would need to decide which previous context to run, the one it was running or the preempted one. The problem arises when the runtime cannot make that decision. It is possible that both contexts were in critical sections in the runtime code, and the preemption handler would be unable to determine if there are any hidden dependencies between the two, such as one thread holding a lock the other will spin on soon. The most common situation for this sort of deadlock is when lock holders are preempted, though there are many other situations in complicated, synchronizing scheduler code. Scheduler Activations limitation was that it could not handle all situations cleanly.

There are many lessons for scheduling and competition, both from old systems and new:

- Coscheduling / Gang scheduling is one part among many that make up a virtual multiprocessor; it is important for performance that all threads communicating run in parallel.

- Userspace needs to provide input on where and when threads and processes run. It can be involved in two separate levels of scheduling: both coarse-grained resource allocation decisions and fine-grained decisions about which thread runs on which processor.

- Scheduling decisions can come from code either loosely or tightly coupled with an application. More coupling requires more programming effort but with greater payoff.

- Be prepared to handle changing resource allocations. The kernel needs to expose the details of which physical resources are allocated to userspace. Likewise, userspace need to be prepared to adapt and perform well as these resources change.

2.4 Synchronization

User-level scheduling is important for performance; often the application knows better than the kernel which thread to run. Nowhere is this more clear than with user-level synchronization: when a thread holds a lock, it is implicitly scheduled and is usually the highest priority thread. Locking invariably happens in userspace: the operations are faster due to the lack of system calls and the kernel does not need to know the locking strategy. As a result, when the kernel makes scheduling decisions, the kernel does not know which thread is a lock-holder and cannot prioritize it. Many systems have battled with how to prevent kernel decisions from interfering with user-level synchronization; it is a special case of user-level scheduling, yet a very common one.

It is well known that time-slicing is harmful to spin-based synchronization[115]; if the kernel deschedules a lock-holder, the remaining threads will busy-wait and waste the processing time that should be used to run the lock-holding thread. The obvious solution to allowing fast user-level synchronization is to simply gang schedule the entire application, forever. Unfortunately, while dedicating hardware to a single application may be useful in some cases, it is not a generally useful solution and is wasteful of resources.

There are many types of spin-based locks, and based on the nature of the locking strategy, preempting lock-*waiters* can also harm performance. The classic spinlock consists of all contenders looping while attempting to atomically `test-and-set` a memory location to "grab" the lock. Preempting the lock-holder is clearly harmful, since all other threads will spin. Preempting a spinner is harmless. However, there are other locking strategies. The MCS lock[69] is a queue-based, FIFO lock, designed for scalability to large numbers of contenders by minimizing cache coherency traffic. Lock contenders form a linked-list, and the lock is passed down the queue. If an intermediate contender is preempted, eventually the lock will be passed to them while it is offline; in essence, a lock-holder was preempted *before* it was a lock-holder. There are many other locking algorithms with similar issues, such as Lamport's Bakery Algorithm[51] and Ticket Locks[22].

A variety of techniques have been developed to deal with preempted lock-holders. The simplest is for the application tell the kernel, possibly via shared memory, when a thread grabs a lock or otherwise enters a critical section[23], and the kernel can allow the thread to continue running for a while. In Symunix[23], there were two flags: one controlled by the user requesting not to be preempted and another controlled by the kernel alerting the user that it was given a continuance. Any extra time granted was taken from the next time quantum.

With the "please do not preempt" flags approach, the kernel needs to make scheduling decisions. How long does it allow a process to continue in its critical section? Should the kernel migrate this process to another processor allocated to the application? Delaying the preemption may be unacceptable; delaying a core preemption means we delay the allocation to another, possibly higher priority process. Migrating to another processor is also tricky: what if the thread on that processor is also in a critical section? Even worse, what if that second thread is waiting on the lock held by the first thread? Many of these schemes to avoid preempted lock-holders do not consider nested locks, which are very common in more complicated scenarios.

As Wesniewski, Kontothanassis, and Scott point out, the flags approach does not work for queue locks[113]. Wesniewski et al.[113] propose two modifications to MCS locks for use with approaches similar to the Symunix flags in which a lock-holder attempts to make sure the next thread in the queue is running. One method is a handshaking protocol in which the original lock-holder waits for a memory write from the new lock-holder to ensure it is running. Another involves extending the kernel interface to include four states of various levels of preemption for a process, which the lock-holder inspects. In both methods, if the next thread in line is preempted, it is skipped over and it will retry. The extra states are due to race conditions and related to the queue lock, and unfortunately they clutter the kernel interface with lock algorithm-specific details. Additionally, the method still relies on the kernel to delay the preemption or immediately reschedule the lock-holder. The flags approach is a preventative approach, and it is often harder to preempt than to recover.

Rhee and Lee[92] developed another method for dealing with preemption with MCS lock participants. Part of their motivation was to not check and set the kernel interface flags for every lock acquisition, and instead only *recover* from preemption. Instead, before joining an MCS queue, the process tells the kernel the address of its `struct qnode` (the bookkeeping structure added to the linked-list). The sign-up code executed while joining, not during the lock handoff, which may help during heavy contention. When a lock-holder unlocks and passes the lock to the next queue member, it performs a kernel operation similar to Black's Handoff Scheduling to ensure the new lock-holder runs. The primary distinction of Rhee and Lee's approach is when the preemption of a lock-holder occurs: the *kernel* inspects the qnode to determine if the process was a lock-holder. This technique closely couples the kernel and the locking algorithm: the kernel needs to know what a qnode is and how it works[92]. Due to the large number of locking algorithms available to applications, tying the kernel interface to a specific protocol is unduly limiting, especially consider how no single locking algorithm works for all workloads and levels of contention. Even if the kernel had the ability to understand multiple types of locks, extending a kernel interface to support new locks would be painful.

Scheduler Activations developed another technique for determining if a thread was in a critical section. All critical sections were annotated or otherwise marked, such that by inspecting the program counter (PC) of a suspended thread, a preemption handler could determine if the thread was a lock-holder or not[2]. Building PC-based detection into the kernel would be extremely difficult, due to the close coupling of the PCs and the address space. Instead, Scheduler Activations could perform this optimization because userspace was responsible for scheduling.

When lock-holders, or other threads in critical sections, were preempted in Scheduler Activations, the kernel sends an event via a new activation to another processor belonging to the address space. The runtime on that processor would handle the event. The handler inspected the PC, and if the preempted context was in a critical section, then the handler changed the PC to a block of identical code that executed the remainder of the critical section. The identical code was different in one way: it yielded to the runtime when it was complete. The handler switched to the preempted context, which would finish its business and then yield[2].

There are three minor issues with this approach: critical sections cannot call functions (they fall back to a flags-based approach), all critical sections need to be copied, possibly at compile time, and critical sections are not always obvious. It is not sufficient to only treat lock acquisitions and releases as a critical section. Individual locks can be part of a larger critical section, involving nested or sequential locks. For similar reasons, Scheduler Activations treated their entire runtime as a critical section, but similar issues would arise for application code.

More seriously, both the PC-marking and the kernel flags approach suffer from a common issue: false positives and false negatives. False negatives are extremely costly, since the preemption is not even detected at all. These techniques can only detect that a thread is attempting to grab a lock, not that the thread actually has the lock. To guarantee no false positives, the lock acquisition code must atomically grab the lock and announce that it is the lock-holder. For the flags approach, if one sets the flag before locking, there is a window where the lock is not acquired. If the flag is set after locking, there is a chance for false negatives. Similar issues exist for the PC approach: the false negative can occur if the PC only consists of code after the lock, and false positives otherwise.

Although Symunix and other kernel-scheduling based models can handle both false positives or negatives, user-level thread schedulers cannot. If the kernel fails to handle a preempted thread in a very rare case, eventually the kernel will time-slice the preempted thread and no deadlock will occur. For Scheduler Activations, the results are worse. Recall that when a lock holder is preempted, another context is preempted and an event is sent to that core. The event handler needed to decide which context to run: the originally preempted one or the secondary context. The dilemma is that both contexts can be in critical sections, or at least contending on critical sections, and it is impossible to determine which is the lock-holder. Either thread could be spinning, waiting on the lock held by the other.

Even worse, if one eliminates the false positives from the critical-section-detection code, both contexts could legitimately be lock-holders, and yet a poor decision by the runtime could result in a deadlock. Consider a system with three spinlocks, A, B, and C. There are two separate code paths: acquire A then C, and acquire B then C. Two separate contexts could acquire and A and B, and then one of them acquires C. If they both are preempted, the runtime must choose which to run, but it is unable to determine which to restart. If there is only one processor available, then the process will deadlock. Additionally, it is possible that the process will never get the extra cores it requires to break itself out of the deadlock. This is a limitation of the Scheduler Activations model, in that the runtime must choose between two options without the flexibility to handle odd cases.

There are other techniques for handling preempted lock holders or queue lock participants. Scott and Scherer developed a version of the MCS lock that can time-out, which can be used to detect a potentially preempted lock-holder[97]. Failure is a useful technique, especially in parallel databases, but it can be difficult to program when any lock acquisition could fail. Later, He, Scherer, and Scott developed another technique to detect preemption for MCS participants: each thread writes what time it thinks it is in a shared-memory region. If a thread falls behind, then it likely was preempted[36].

Dealing with preempted lock holders is complicated, and more so for user-level schedulers. Our lessons are:

- Preemptions will happen, and we need a clean way to detect preemption when it does.
- There are a variety of locking strategies, and one size does not fit all. We do not want to build a particular locking strategy into the kernel interface.
- Detecting critical sections is difficult: unless a thread atomically grabs the lock and marks itself as a lock-holder, there will be false positives or false negatives.
- User-level schedulers make preemption recovery more difficult: if they choose the wrong thread to run, they could deadlock. Additionally, the preemption handling code itself could be preempted.

2.5 Performance Isolation

Parallel applications, especially those that communicate among their threads, are sensitive to interference. For predictable, high performance, parallel applications need performance isolation from other programs. If a machine is dedicated to a single application, then that application still needs some form of isolation from the OS.

With regards to processors, there are a few different types of interference that are under the control of the operating system. Three three types are really two more generic types: the OS's scheduling decisions and the OS itself.

- Interprocess scheduling: the operating system did not give the process sufficient resources, e.g. cores.
- Intraprocess scheduling: for systems where the kernel schedules a process's threads, the kernel may run the "wrong" thread.
- The operating system itself: interrupts and background tasks can interfere with an application that has been granted a core.

In this work, I focus primarily on processors, but obviously processors are not the only resource in a system. Contention for RAM, caches, bandwidth, and devices can also dramatically affect performance. Dealing with any sort of performance isolation will require considering all resource types.

Operating System Noise

The operating system itself is responsible for various types of interference, even when the core is dedicated to a particular application. Many general purpose operating systems are designed such that every core runs its own scheduler. Although not as independent as nodes in a distributed system, each core is relatively on its own: each has a timer tick, a run queue, and background jobs. For instance, on Linux these background threads include the migration thread, ksoftirqd, the generic event workqueue, kblockd and other housekeeping. Furthermore, there are a variety of device interrupts which are often sent to all cores to lower IRQ latency. Operating systems such as Linux provide tools to reroute interrupts away from cores and to pin applications to cores. Even in this case, certain interrupts such as the timer interrupt are not reroutable. These various types of interference are referred to as operating system noise or jitter.

Interference problems with general purpose operating systems are well known in the High Performance Computing (HPC) community[84, 29, 48]. Several custom HPC operating systems have been developed to provide better support for HPC applications[75, 93, 54]. These are typically lightweight kernels that run on compute nodes of supercomputers, and their goal is to reduce OS noise.

As an example of why a brief interruption can be harmful to performance, consider a large application running on 1024 processors. The application consists of phases, with barriers in between each phase. If one processor is delayed slightly, then all remaining processors are delayed at the barrier. In essence, the delay of one processor is magnified. This problem is specific to parallel programs that communicate; independent, single-threaded tasks that do not communicate only suffer the initial delay. The effect of interference on an application is based on the characteristics of the interference and the communication. Low-frequency, long-duration interruptions from background tasks are especially damaging[29].

Sottile and Minnich developed a benchmark to both measure and, more importantly, analyze system noise. The Fixed Time Quantum (FTQ) benchmark[101] measures how much work is done in a small time quantum, over a larger time interval. The ideal situation is for there to be no interference at all. The benefit of FTQ over previous benchmarks is that the time-slice is fixed, which means the benchmark's output can be analyzed for periodic signals. For instance, imagine a scheduler tick every 10ms: while the interference may be hard to detect from looking at the raw output, if one runs an FFT on the data, the 10ms spike is readily apparent. I discuss FTQ in later sections, as it is very useful to determine the sources of OS noise and to compare different operating systems' isolation.

Linux is also pursuing a path towards less OS interference. The latest is a series of patches by Frederic Weisbecker called "Adaptive NoHZ"[111]. The idea is to turn off the scheduler tick on cores that have only one process running. The patches remove the OS jitter associated with the scheduler tick (3% on FTQ). Adaptive NoHZ is now a kernel option, as of 3.10[21]. Linux still has some background interference, such as global IPIs for cache management, which other people are working on removing. Features such as per-core runqueues will be harder to remove, and are the result of extending a uniprocessor OS to a small-scale SMP.

Resource Allocation

Many researchers in the OS and real-time community have worked towards performance isolation on general purpose OSs[32, 73, 26, 78, 114]. For example, Verghese, Gupta, and Rosenblum developed a *Software Performance Unit* (SPU)[110] which allows the sharing of resources between high-priority jobs and background tasks. SPUs could isolate CPU time, memory, and disk bandwidth. One drawback is that SPUs rely on processes to actively share unused resources with background tasks, instead of allowing the system to manage these resources. Mercer, Savage, and Tokuda developed a processor *reserves* system in Real-Time Mach[70]. Reserves provided processor isolation by reserving the processor at particular times for a real-time application, while allowing time-sharing during the other times. Reserves were static and for applications that have a predetermined schedule.

Systems such as VServer[100], Solaris Zones[87], and Linux Containers[60] provide resource partitioning within an operating system. However, they attempt to completely isolate applications: the systems isolate not only performance but also configuration, namespace, and all forms of interaction. These container systems are similar to virtual machines (VMs) and they are often considered lightweight virtualization. VMs are also used for resource isolation. Although very useful, VMs are simply the wrong abstraction for interacting applications on a single system. They needlessly couple protection with resource management and do not normally allow flexible sharing or reallocation of resources. Other than for convenience, protection domains do not need to be coupled to resource management. As an example of the reverse, consider a basic chroot: it has an isolated namespace and cannot interact with other processes, but there is no performance isolation. In that regard, Banga, Druschel, and Mogul developed a *resource containers* system[6]. Resource containers allow accurate accounting of resource utilization, independent of the execution context, effectively decouple resource management from protection.

Instead of merely partitioning resources in software, systems can also partition resources in hardware. Back in the 1970s, C.mmp[62] had hardware switches that could physically partition processors and memory. These switches were programmable by software running on processor 0. The original intent was to run multiple independent versions of Hydra, but a lack of RAM made that infeasible. Instead, the feature was used to perform maintenance in parallel with normal operations and to isolate faulty hardware.

Any serious multi-tenant system will need some form of resource isolation. When a system also can reduce its OS noise, parallel applications can achieve predictable, high performance. Many systems have done so, and there are some lessons we can learn:

- Both resource allocation and OS noise can affect application performance; both need to be addressed.
- Performance and resource isolation do not need to be coupled to all forms of isolation. It is more flexible to provide performance isolation as one option among many.
- Static partitioning is inefficient when applications have dynamic resource requirements. We would like to both reserve resources for high-priority applications and utilize idle resources.

Chapter 3

The Basics of the Many-Core Process and Akaros

The primary change to operating systems to better support parallel processes is a fundamental change to the process abstraction itself. In my work, this new process abstraction is called the "Many-Core Process" (MCP). The MCP has many characteristics of previous solutions to parallel processing and is the intersection of many of the "lessons learned" from prior work. In this chapter, I will describe the MCP at a high level and discuss relevant details about the Akaros Operating System, as a preparation for the details of the MCP in the next chapter.

3.1 MCP Basics

The MCP is similar to the familiar Unix process, likewise called an address space in Topaz and many other systems. MCPs are a vehicle for running user-level code, are the unit of protection and control in the kernel, communicate with each other, and are executing instances of a program. The kernel maintains a process control block for each MCP, containing information such as the open file table and the page tables. MCPs differ from Unix processes and other kernel-threading based systems in that they directly provide a virtual multiprocessor. The kernel grants an application cores directly, instead of threads, and exposes various structures relating to the management of cores. Additionally, MCPs are designed from the ground-up to support user-level threading and scheduling. These decisions are based on lessons from prior work:

- The kernel thread needlessly couples both parallelism and I/O concurrency and should *not* be the kernel-provided abstraction. A virtual multiprocessor abstraction is key, wherein the available physical cores are exposed to the application.
- Thread-specific details, such as what sort of stack to use, and operations, such as context switches, need to be in userspace for the highest performance.

- Userspace needs to provide input on where and when threads and processes run.

The MCP extends the traditional process model in the following ways:

- All of an MCP's cores are gang scheduled, and a process will always be aware of which cores are running.
- Faults are redirected to and handled by the process in a software-interrupt handling context.
- There are no kernel tasks or threads underlying each thread of a process, unlike in a 1:1 or M:N threading model. The kernel *may* use threads internally, but they are not connected to the process abstraction or the user-kernel interface.
- All system calls are asynchronous and never block the user's core.
- Resources, such as cores and memory, are explicitly granted and revoked.
- MCPs will not receive unexpected, unwanted interrupts. When a core is granted, it is completely dedicated to the process.

Earlier I remarked that Scheduler Activations provided a virtual multiprocessor, but failed to provide a virtual processor. Likewise, Psyche did the opposite: it had virtual processors, but not a virtual multiprocessor. The MCP provides both: the application has a virtual multiprocessor made up of virtual processors. Unlike Scheduler Activations, the MCP cores are not a scheduling policy for kernel threads (activations), but instead act more like Psyche virtual processors that act similar to hardware processors.

From the user's perspective, an MCP is a collection of cores, running user code and syscalls in parallel. Userspace schedules and manages its own threads on these cores. Most of userspace's code for managing the MCP is a user-level scheduler library, linked and closely-coupled with an application. The user-level scheduler is also called a second-level scheduler (2LS), where "second level" has Feitelson's meaning[27]. The two levels of scheduling are the allocation of processors and the decision of which thread uses the processor. The kernel is responsible for the first level, and the 2LS is responsible for the second (thread scheduling). Note that loosely coupled, userspace daemons can influence the first level of scheduling with provisioning, as discussed below in Section 3.2. As we saw from previous work, userspace needs to be involved in both levels of scheduling.

Other lessons from previous work contribute to the MCPs features. Gang scheduling is a natural fit for virtual multiprocessors. OS noise can affect application performance; hence the lack of unexpected interrupts. The combination of gang scheduling and a lack of interference allows better cache utilization and fast synchronization. Asynchronous syscalls or some mechanism is necessary to return control to userspace when syscalls block.

It is worth stating again that there are no kernel tasks or threads backing any part of an MCP: neither user thread nor virtual core. In older systems, kernel threads were either used explicitly or implicitly, as in the case of Scheduler Activations, to build the parallel processing abstraction. In existing systems, such as Linux, the kernel thread in the form of a pthread is used for both parallelism and I/O concurrency. The MCP differentiates from these models by explicitly separating cores from threads. Simply put, cores are for parallelism, and threads are for concurrency. The kernel interface is only concerned with cores; userspace does not know or care whether or not there are kernel threads at all.

Before going into any greater detail on the MCP, we need to learn a little about Akaros.

3.2 Akaros Details

Akaros is a research operating system for high performance and parallel applications on single-node, large-scale SMP and many-core architectures. Although the MCP is a major component of Akaros, it is part of a broader approach to supporting high-performance applications, especially in a data center. Akaros's philosophy is one of transparency: the OS exposes information about the underlying system to application and provides interfaces for them to request exclusive access to explicitly granted resources.

A critical point about transparency is how Akaros deals with resources and virtualization. Akaros grants physical resources to applications, but does so through a virtual naming system. Consider a resource, such as RAM. Virtualization is used for naming and is often used to share scarce physical resources among processes. In the case of RAM, a page of memory has a virtual address that can map to a physical address via a page table, and it may or may not be backed by an actual physical page. From the perspective of a process's instructions, the page exists; it just may require a page fault handler to allocate a physical page to map. Interposing on a virtual resource is very useful for operating systems. The OS can conveniently share a resource among many processes such that they all believe they have their own resource. However, there is a tradeoff: the convenience of virtualization comes at the cost of performance when a resource is not physically present. Akaros uses virtualization for naming, but applications will always be aware of the existence underlying resources. For instance, Akaros uses virtual memory and paging hardware, but userspace is in control of whether or not the virtual pages are backed by physical pages, and userspace has read-only access to the actual page tables.

From the programmer's perspective, Akaros supports some aspects of the POSIX interface and has a variety of custom extensions for its own use. The OS and toolchain is compiled with GCC, and there are ports for glibc and Go. Basic programs like busybox and pthread-based applications compile and run with minimal modifications. Akaros also has major portions of the Plan 9 Operating System[85] incorporated into the kernel; specifically the namespaces, networking stack, and various device drivers. Akaros runs on x86 and RISCV in both 64 and 32 bit modes, though 32 bit support is likely to go away. Although supporting existing toolchains and infrastructures is useful both for researchers and adoption, the main focus of Akaros is in how it differs from existing OSs, not in how it is similar.

Kernel Scheduler Design

The Akaros kernel scheduler's job is to allocate spatially partitioned, gang scheduled cores, with the additional desire of no kernel interference per core. As systems have more and more cores available, spatial partitioning becomes increasingly valuable. Likewise, with many cores, traditional per-core kernel schedulers may become a scalability bottleneck and a source of interference.

The Akaros kernel runs a centralized scheduler with a single run queue, in contrast to the more common per-core run queues. To gang schedule with per-core run queues, an OS needs to coordinate between the multiple schedulers so that the same process is scheduled at the same time. In contrast, the Akaros scheduler simply instructs a small executive on the cores to run the appropriate process. It is analogous to having per-core run queues with a maximum of one process per queue, with less complexity and no timer interrupts. The global scheduler code can run on any core, and it is driven by both interrupts as well as requests from userspace. Typically, the scheduler runs in response to a timer interrupt on core 0.

In traditional operating systems, the time quantum is the same throughout the system, and larger quanta mean higher throughput at the expense of the responsiveness of other processes. The Akaros kernel scheduler uses different time quanta on separate cores, based on their type. Cores are either *coarse-grained* cores (CG) or *low-latency* cores (LL).

LL cores, also called *management* cores, run a typical timer tick at 100Hz or 1000Hz. Application event handling, interrupt handling, single-core processes (SCPs), and kernel housekeeping tasks are scheduled on LL cores at a fine granularity. LL cores are analogous to front-end processors in mainframe and cluster computers.

MCPs are scheduled with a coarse granularity on the CG cores, and no undesired interrupts are sent to CG cores. The coarse granularity implies less overhead and better cache and TLB performance due to fewer context switches. An MCP reaps greater benefits from isolation and gang scheduling when it is given a larger time quantum. The actual amount of time an application runs on a CG core is dependent on the overall scheduling policy; there is no specific quantum, and they are spatially partitioned. When an application has a core granted to it, it is the only application on that core. Based on priorities, the kernel may never reallocate the core.

In the current code base, core 0 is the only LL core, and the plan is to allow the scheduler to dynamically change a core's type. LL cores are basically the same as cores in existing OSs, and the distinction only matters when discussing CG cores. In Akaros terms, current general purpose OSs consist entirely of LL cores.

Provisioning and Allocation of Resources

Akaros's resource management is driven by a policy that makes a distinction between provisioning and allocation. Recall from prior work's lessons learned:

- Userspace needs to provide input on where and when threads and processes run. It can be involved in two separate levels of scheduling: both coarse-grained resource allocation decisions and fine-grained decisions about which thread runs on which processor.
- Static partitioning is inefficient when applications have dynamic resource requirements. We would like to both reserve resources for high-priority applications and utilize idle resources.

The Akaros scheduler allows userspace to *provision* a set of resources for a process. Provisioning a resource to an application is a guarantee of future, uninterrupted access to that physical resource. When the application asks for the resource, the kernel will perform the *allocation*. Provisioning is directed by userspace; this is the coarse-grained resource allocation decision-making that userspace needs to be able to make. The fine-grained scheduling decisions are made by user-level schedulers (2LSs) *within* an MCP.

Resources that are provisioned, but not allocated to the provisionee, can be allocated to another process, similar to backfilling in cluster schedulers. When the provisionee requests the resource, the kernel revokes the resource from the current user, and allocates it to the provisionee. Provisioning and preempting allows the system to respond to changes in workload and priorities, unlike static partitioning schemes. Typically, resources are provisioned to an application based on peak load, and the amount used at any particular time may be less. High-priority applications have the resources they need, and resources are not wasted.

Provisioning in Akaros is done with the syscall:

```
sys_provision(int pid, int res_type, int res_val)
```

Allocation requests are posted to a user-writable shared memory region, where there is an array of the following structs:

```
struct resource_req {
    unsigned long            amt_wanted;
    unsigned long            amt_wanted_min;
    int                      flags;
};
```

The kernel will not run the process unless at least `amt_wanted_min` is set, though this is not used currently. A process can ask the kernel to check this structure by using the syscall `sys_poke_ksched()`; otherwise the kernel will check it periodically.

Older versions of resource requests passed the values in the system call, instead of in memory. The reason for the indirection is that it removes locking and minimizes some race conditions in userspace where multiple cores may be modifying `amt_wanted`. It allows userspace to operate directly on the real fields, notably `amt_wanted`, with atomic operations and without needing to maintain shadow state.

Although the provisioning system is built to handle multiple types of resources, currently Akaros only supports provisioning cores. For an MCP, this means that a provisioned core is uninterruptible and irrevocable while it remains provisioned. An unprovisioned, allocated core can be preempted at any time.

Of course, provisioning can change over time. The kernel expects a user-level daemon to set the provisions and adjust them over time, similar to C.mmp's policy modules. In the data center, these decisions would be made by a cluster manager, such as Mesos[39], and the daemon would be the agent controlling the single node.

Kernel Messaging

Akaros has a kernel-to-kernel messaging service, where messages are simply work that is shipped remotely, delayed in time, or both. They consist of a function pointer, a few arguments for that function, and source and destination id numbers. Kernel messages (KMSGs) of a given type will be executed in order, with guaranteed delivery. These were originally an implementation of low-level hardware messaging based on Active Messages[24]. KMSGs are very useful in MCP management code and are the primary means of giving a core to a process.

Initially, kernel messages were meant to be a way to immediately execute code on another core (when interrupts are enabled), in the order in which the messages were sent. Although useful, this was not sufficient for some use cases. For one, some subsystems want their functions to run when the kernel is not in interrupt context. More specifically, some kernel messages do not return because they start running another context. The kernel cannot run these sorts of messages while interrupting arbitrary kernel code. Failing to return means the previous context is completely destroyed, and locks, reference counts, and pointers could be lost. Instead, certain types of kernel messages are queued up and executed at a convenient time. These delayed messages are also very useful for deferring work on the core sending the message, similar to the usefulness of self-IPIs (interprocessor interrupts).

Currently, there are two types of KMSGs, distinguished by which list they are sent to per core: *immediate* and *routine*. Immediate messages will get executed as soon as possible, directly in interrupt context. Routine messages will be executed at convenient points in the kernel. These spots include when the kernel is about to pop back to userspace or calling an idle function (`smp_idle()`), which checks for work to do and otherwise halts the core. Routine messages are necessary when their function does not return. They should also be used if the work is not worth fully interrupting the kernel. Finally, they should be used if their work could affect currently executing kernel code, such as a syscall.

The general approach for sending kernel messages is to grab the destination core's lock, and add a memory-allocated block containing the message to the appropriate per-core linked list. Then send an IPI to that core. For immediate messages, the IPI is mandatory. For routine messages, an IPI is usually sent, but certain optimizations are available. If the sender can determine by inspecting certain flags for the destination that the destination core is in the kernel and will later check routine messages, it can avoid the IPI. This optimization is similar to the *grace periods* used in Read-Copy-Update[68] (RCU), a kernel synchronization method used in Linux. Another potential optimization is to use fixed-sized arrays for the messages, and only use the locks, lists, and memory allocator when the arrays overflow. Care needs to be taken to maintain the proper ordering of messages when transitioning to and from the array and lists. I have not needed to implement either optimization yet.

MCP management code makes heavy use of routine kernel messages. Even though these MCP management functions return, they must still not be immediate messages. These messages can change the ownership of a core, which can interfere with some system calls that are in progress. The safe thing to do is to delay the work until it is safe to run routine messages.

With routine messages, there are no concerns about the kernel holding locks or otherwise interrupting its own execution. Additionally, routine messages are allowed to block in the kernel on a semaphore or condition variable. Despite the hazards of immediate messages, there are still plenty of use cases. For instance, an example of an immediate message would be a TLB shoot-down. Immediate messages are basically just interrupts with arguments, subject to the same restrictions of interrupt handlers in operating systems.

Other than process management, a major user of kernel messages is the ability to launch or resume a kernel thread.

Kernel Threading and Asynchronous System Calls

The MCP abstraction is one of cores, not threads. The kernel does not know about the user's threads, only that the user executes on cores. Likewise, the user has no hints as to whether or not the kernel uses threads. Internally, the kernel uses a form of lightweight threading that does not affect the user's execution model. A `struct kthread` has very little to it: a set of registers, a stack, and some slight bookkeeping. There is no floating point state; that is saved per-core, when relevant. Kthreads are used for executing system calls and performing internal kernel work.

Originally, the plan for Akaros was to be a purely event-driven kernel. I mentioned earlier that V's processes did not have kernel stacks. This was because the kernel had no blocking system calls: the kernel was an event-driven microkernel, and had no need for kernel threads or semaphores. Akaros was going to do the same thing: have a single stack per physical processor to handle traps and syscalls. However, Akaros is not a microkernel, and its syscalls are more complicated than just sending messages to user-level servers.

Why not just be event driven for all I/O? Why do we even need kernel threads? In short, I/O subsystems are not as simple as "perform a blocking operation and when it completes, run a function." Although that is what the block device driver will do, for example, the subsystems actually needing the I/O are more complicated. Consider the potentially numerous blocking I/O calls involved in opening a file in a common file system such as ext2. The path lookup alone could require several reads, and inodes may have several levels of indirection to reach an actual block. Having a continuation for each one of those points in the call graph is a painful way to code. Additionally, it is often unclear if a particular operation will block or not. For instance, when performing a path lookup, directory entries (dentries) along the path may or may not be in a cache. When programming these systems, it is much simpler to use a thread, and it only costs a page plus a small data structure.

Note that using threads does not mean that all I/O needs to use kthreads, just that some scenarios will benefit from it. Event-driven styles are still useful, and it is possible to pursue a hybrid approach. For instance, in a `read()` call, the kernel can pursue a threading style for the intermediate I/Os (such as inode lookups), and use an event driven style for the final device read. In this scenario, there is a simple continuation attached to the I/O, sufficient to clean up and finish the system call, but not requiring the full stack and thread state. I have not built this in Akaros, mostly due to lack of time and that it would require possibly major adjustments to the existing OS subsystems. Regardless, events and threads in the kernel are both tools in a toolbox, and ideally we would pick the right tool for the job. When choosing between events and threads, the developer should consider the number of potential blocking calls and the amount of live state to maintain across the blocking calls.

The critical aspect of kthreads in Akaros is that they are not part of the user-kernel threading model. In typical OSs, some aspect of a kernel thread is involved in the execution of a syscall. For 1:1 systems, the connection is obvious; when the kernel thread executing the syscall blocks, so too does the user thread. Likewise, for M:N systems, when N is the number of kernel threads. For systems like Scheduler Activations, even though N is the number of processors, the kernel thread is connected to the activation. When a kernel thread blocks on a syscall, the user restarts on a new activation. A large part of the distinction between Akaros and these systems is the asynchronous system call interface. For systems with synchronous syscall interfaces, the control flow is linked to whether or not a syscall completed or not. With an asynchronous interface, the user retains control of its core independently. Since kernel threads are not part of the model of execution, even when control returns to the user, the kernel may or may not have used blocking kernel threads. The kernel is free to explore both event-driven or thread-based approaches for executing syscalls.

The main benefit to decoupling kthreads from a virtual core or per user process/task is so that when the kernel blocks, it can return to the process. This is the essence of the asynchronous syscall interface, though it is not limited to syscalls; I discuss page faults in Section 4.4.

In Akaros, the syscall contract is based around the syscall struct:

```
struct syscall {
    unsigned int                num;
    int                         err;            /* errno */
    long                        retval;
    atomic_t                    flags;
    struct event_queue          *ev_q;
    void                        *u_data;
    long                        arg0;
    long                        arg1;
    long                        arg2;
    long                            arg3;
    long                            arg4;
    long                            arg5;
    char                            errstr[MAX_ERRSTR_LEN];
};
```

To issue a syscall syscall, userspace fills in the number and the arguments, then submits the struct to the kernel. Typically, the program traps into the kernel, passing the pointer to the struct. For calls that do not block, usually the kernel will finish the call and return to userspace. For calls that block, the kthread blocks and a new one takes over the core and returns to userspace. Either way, the application does not lose its core. When finishing syscalls, the kernel writes in the return value, sets a flag, and optionally sends an event, as discussed further in Section 5.2.

Although most syscalls in Akaros are local, meaning that they are submitted to the kernel via a trap, there is also a subsystem that allows the posting of syscalls into shared-memory ring buffers where the execution of a syscall can happen on another core. Since the contract between the user and kernel is the syscall struct alone, it makes no difference whether or not the local kthread executes the syscall, if another kthread executes on another core, or if there is no kthread whatsoever.

When a kthread executes a syscall on behalf of a process, it executes in that process's address space. Typically, this is called executing in *process context*, as opposed to *IRQ context*. When a kthread blocks, if it was in process context, it saves a reference count on the process. In that sense, a given kthread temporarily belongs to a process. However, when the syscall is complete, that relationship ends, and the kthread reverts to merely handling traps and interrupts. If a process yields a core, the kthread remains and will service other syscalls on behalf of whichever new process runs on that core.

System call handling code does have one issue: after a kthread blocks and control returns to the process, later the kthread will unblock, complete its call, and unwind its stack. When it does so, it must not "pop" the original user context that trapped and return again, since that context was already restarted. The solution is for all trap handlers, and interrupt handlers too, to not return at all. Instead, all of these low-level routines call common functions that figure out what a core should do; this is `smp_idle()` and `proc_restartcore()`. On trapping into the kernel, the currently running user context is saved in per-cpu data, so that it can be restarted independently of any kthread operations. The job of `smp_idle()` is simply to figure out if there is a user context and restart it, as well as handling any routine kernel messages, and if not, to halt the core.

Synchronization Methods

Akaros has a variety of synchronization primitives, including busy-waiting and sleeping methods, as well as some less-common tools. Basic familiarity with them is necessary to understand how MCPs are designed and implemented.

For busy waiting, Akaros uses typical spinlocks and the obligatory "IRQ-save" versions, which disable interrupts before grabbing the lock. It is necessary to disable IRQs before acquiring a lock that may be grabbed by an interrupt handler to prevent deadlock. Otherwise, an interrupt handler could interrupt a lock holder and then spin indefinitely in a circular-wait scenario, since the lock holder implicitly waits for the interrupt handler to return.

Akaros also uses the *kref* pattern for reference counting, which is used heavily in Linux[14]. Krefs are used to reference count objects within the kernel. The typical pattern is that a certain clean-up routine is executed when the last reference is dropped and the refcount hits zero. Although there are many more details for krefs, the cleanup pattern is the most relevant to MCPs.

Akaros has a custom synchronization pattern I call *post and poke*. Post and poke is a wait-free[37] way (no spinning or blocking) to make sure some code is run, usually by the calling core, but potentially by any core. Under contention, every caller just posts work, and one core will carry out the work. Callers post work somewhere, the meaning of which is particular to their subsystem, then set the function pointer in `struct poke_tracker`, and finally call `poke()`. For example, posting work could be appending an item to a queue. Once `poke()` returns, the function either has run or is currently running somewhere else. The function pointer is not run concurrently with itself; it can be assured of single-threaded behavior. The guarantee is that the function will run at least once when `poke()` is called, so that the posted work is present for at least one run of the function. In the future, `poke()` could send a kernel message to an LL core to ensure the work is done somewhere, but not necessarily on the calling core.

The code for `poke()` is relatively simple, except for the memory barriers. Akaros has a few types of memory barriers, such as `wmb()` (write), `cmb()` (prevents compiler reordering), and `wrmb()` (write then read, preventing RAW hazards). These barriers are architecture dependent. For instance, on x86, `wmb()` and `rwmb()` are just a `cmb()`, while both are full hardware memory barriers on RISC-V. Some, but not all, atomic operations provide a hardware memory barrier.

```
void poke(struct poke_tracker *tracker, void *arg)
{
  atomic_set(&tracker->need_to_run, TRUE);
  /* will need to loop repeatedly if someone else keeps posting work */
  do {
    /* want an wrmb() btw need_to_run and in_progress.  the atomic_swap
     * provides the hardware mb, so we need a compiler barrier. */
    cmb();
    /* poke / make sure someone does it.  if we get a TRUE (1) back, someone
     * else is already running and will deal with the posted work. */
    if (atomic_swap(&tracker->run_in_progress, TRUE))
      return;
    /* clear the 'need to run', since we're running it now.  new users will
     * set it again.  this write needs to be wmb()'d after in_progress.  the
     * swap provided the HW mb(). */
    cmb();
```

```
        atomic_set(&tracker->need_to_run, FALSE);    /* no internal HW mb */
        assert(tracker->func);
        tracker->func(arg);
        wmb();  /* ensure the in_progress write comes after the run_again. */
        atomic_set(&tracker->run_in_progress, FALSE);    /* no internal HW mb */
        /* in_progress write must come before need_to_run read */
        wrmb();
    } while (atomic_read(&tracker->need_to_run));
}
```

Post and poke is not necessarily novel; it is merely a useful pattern I used on a few occasions, both in the kernel and in userspace. It is lock-free and wait-free and should scale relatively well: under contention, most members only do one write and one read. It embodies a principle I have found very useful in developing a parallel OS: make spurious messages harmless. If messages, whether they are interrupts, pokes, or events, are indications to look elsewhere to do work, then it is okay to send numerous messages to accomplish a single task. Since messages can get lost or delayed, we can just send another message without fear of the underlying work being done more than once — the messages are idempotent. This technique pops up all over the place in Akaros, from the virtual core implementation to event delivery options. The argument for safe, spurious messages is similar to that of Mesa semantics versus Hoare semantics for condition variables: Mesa-style condition variables are more flexible and have simpler invariants[53].

For blocking synchronization, Akaros has the usual semaphores and condition variables. Rather simply, when these calls sleep, the calling kthread blocks. Additionally, Akaros supports a form of Plan 9's *rendezvous*[86]. Rendezvous captures a common condition variable pattern: sleep, if a function pointer (an argument to `rendez_sleep()`) returns false. Another thread or an interrupt handler can wake the sleeping thread. One difference between Akaros and Plan 9 rendezvous is that Akaros uses Mesa semantics for the condition, while Plan 9 uses Hoare semantics. Specifically, Akaros's `rendez_sleep()` is: sleep *while* a function returns false, and Plan 9's is: sleep *if* a function returns false.

Abortable System Calls

Operating systems want some way to abort a system call. For a common example, applications may want to cancel or timeout a networking operation. In OSs with 1:1 threading and synchronous system calls, the typical mechanism for this is to send a signal to the thread. For instance, POSIX `read()` calls can be interrupted with a signal, and the read returns with errno set to EINTR. Typically, code will set an alarm to signal itself before performing a blocking call that it may wish to timeout. In systems with asynchronous calls, usually there is some instruction or API to cancel a call. For instance, in VMS, QIO calls can be aborted with an ABORT command[38]. Akaros also needs to abort system calls, and given its asynchronous syscall interface, signals are not sufficient.

Before going into details on Akaros's abortable system calls, we need to know about error handling in Plan 9. Plan 9, and now Akaros, has an error handling mechanism based on `setjmp()` and `longjmp()`, whereby a context is saved and later jumped to when there is an error. A common usage:

```
 1 {
 2   some_data = alloc_data();
 3   if (waserror()) {
 4     /* take this branch in case of error */
 5     kfree(some_data);
 6     /* jump to the previous waserror() in the call graph */
 7     nexterrror();
 8   }
 9   /* any of these functions can call error(‘‘some error message’’),
10    * which will longjmp to the waserror's if branch at line 4 */
11   some_function();
12   another_function();
13   /* pop the waserror from line 3 from the error stack */
14   poperror();
15   kfree(some_data);
16 }
```

Code written with `waserror()` and `error()`, which includes all of the syscall paths related to networking and file systems, can handle any error from deeper in the call graph, can restore invariants, and ultimately return the error message to a syscall. In this manner, `error()` is similar to the `UNWIND` exception in Mesa[53], and `waserror()` is the programmer's way to install an `UNWIND` handler.

In Akaros, any syscall with a kthread sleeping in a `rendez_sleep()` is abortable. When this happens, `rendez_sleep()` itself throws a "syscall aborted" error, which trickles all the way back to userspace. The only rule for abortable syscalls is that code needs to restore invariants in waserror blocks before calling sleep, same as handling any error generated from the code paths. Cleanup involves freeing memory and unlocking semaphores, for example.

However, arbitrary sleeps in Akaros are not abortable, yet. For instance, if a kthread is waiting on an arbitrary semaphore, we do not allow an abort. The reasoning is that the kthread will eventually acquire the qlock; the code is not usually waiting on external sources to wake up. That is not completely true — a kthread could be blocked on a semaphore, and the semaphore holder could be abortable. In the future, I could build some sort of "abort inheritance", usable by privileged applications. (There is a danger of aborting another process's kthread). Alternatively, we could make semaphores, called *qlocks* in Plan 9, abortable too, though that would require rewriting all qlocking code to be unwindable. In short, code written with `rendez_sleep()` expects the sleep to be able to throw an error.

The rules for writing code that will sleep was easy. The harder part to syscall aborting is safely waking a kthread. There are several layers to go through from a thread or syscall in userspace down to the condition variable a kthread is sleeping on in the kernel. Given a user thread, find its syscall. Given a syscall, find its kthread. Given the kthread, find the condition variable. During all of these lookups, syscalls complete concurrently, kthreads get repurposed for other syscalls, and condition variables could be freed. The memory for syscall structs themselves are often on the stacks of user threads, and when they complete, the memory is both gibberish and potentially in use.

Ultimately, I decided on a system of safe abort attempts, where it is harmless to be wrong with an attempt. Instead of dealing with the races associated with memory freeing and syscalls completing, the aborts will only be attempted if it is safe to try. This is another example of the poking principle: make spurious messages harmless. Since the contract between the kernel and the user is the syscall struct, then this struct should be involved in the cancellation of the syscall. When userspace wants to abort a syscall, the kernel uses that pointer as a key for a lookup. Only if the lookup succeeds is it safe to dereference the pointer or otherwise cancel the syscall.

All abortable kthreads/sleepers/syscalls are on a per-process list, indexed by syscall struct pointer. They are only on the list when they are abortable, meaning the condition variable inside the rendezvous can be poked. The invariant is that when they are on the list, they are in a state that can be safely aborted: the kthread is working on the syscall, it has not unwound its call graph, etc. The details of this protection are sorted out with `__reg_abortable_cv()` and `dereg_abortable_cv()`, since it is really the condition variable that the kernel is trying to find. From the kernel's perspective, nothing bad can happen if you ask to abort an arbitrary syscall for a particular process.

The actual abort takes the "write/signal, then wake" approach. The aborter tracks down the kthread via the lookup, the success of which guarantees the sleeper is sleeping, marks the syscall's flag with SC_ABORT, and attempts to wake the kthread with a condition variable broadcast. The per-process list is protected by a lock: some form of synchronization is required. The benefit to this approach is that this penalty is only paid on when a kthread is blocking: a relatively expensive operation. It is only during this time, when a kthread and its syscall are on a list, that an external observer can be certain of a mapping between kthread and syscall. By comparison, in 1:1 models, kernel threads are implicitly working on their particular syscall: one that has no struct and whose commands were passed via registers during a trap.

On the user's side, applications with user threads that block in userspace on syscalls can set an alarm to cancel the syscall for the user thread, similarly to requesting a signal. Userspace has an alarm service which allows them to run arbitrary handlers after a certain amount of time, managed by the virtual core code. Likewise, user threads can block on system calls, which I discuss further in Section 4.4. Additionally, userspace can set an alarm and issue multiple syscalls within its alarm's timeout window; the alarm will cancel whichever syscall is currently running.

There are downsides to the "safe abort attempts" approach. The system can only abort syscalls when they are at a certain point; if they are not currently sleeping, the call will fail. Technically, the abort could take effect later on in the life of a syscall. Related to this limitation, userspace must keep attempting to cancel a syscall until it succeeds, if it so chooses. It may also be told an abort succeeded, even if the syscall actually completes. Ultimately, we cannot "fire and forget" our abort attempt. These are not huge problems, and they are less of a problem than my older approaches.

For instance, the original idea I had was for userspace to flag the syscall (`flags |= SC_ABORT;`). It could do this at any time. Whenever the kthread was going to block in an abortable location (e.g. `rendez_sleep()`), it would see the flag and abort. It might already be asleep, so we would also provide a syscall that would "poke" the kthread responsible for some syscall pointer, to wake it up to see the flag and abort. The first problem was writing to the syscall struct flags. Unless we know the memory is actually the syscall we want, this could result in randomly writing to memory (such as a user thread's stack). I ran into similar issues in the kernel: one cannot touch a kthread struct unless you know it is the kthread you want. Once I started dealing with the syscall to kthread mapping, it became clear I would need a per-process lookup service in the kernel, which acts as a way to lock a reference to the kthread. The memory safety problems, both for the kthread struct and the syscall struct, are all solved by synchronized lookups to ensure safety.

As a smaller note, I considered registering every kthread with the process right away, specifically, when we link the syscall to the kthread, for the syscall-to-kthread lookup service. This would be expensive, since every syscall pays the lookup tax. We want syscalls to be fast, but the infrequent aborts can be expensive. The obvious change was to only save the mapping for only the abortable kthreads. The tradeoff is that we cannot flag syscalls for aborting unless they are in an abortable state. This approach, the one I settled on, requires multiple pokes by userspace. In theory, userspace needs to deal with that scenario anyways, in case they attempt to abort before the kernel can register in the first place.

As an example of how powerful the cancellation interface is, consider an application that wants to time out its reads and writes. Kevin Klues wrote a simple webserver in C called Kweb, which we ported from Linux to Akaros. The Linux port requires a variety of epoll calls, while loops, and non-blocking socket options. The Akaros port merely sets an alarm and issues the normal reads and writes.

It is also worth noting that many `rendez_sleep()` calls actually return right away. For instance, this is common if some data is already in the queue, or whatever the condition is that we want to conditionally sleep on is true. Since registering the kthread and syscall is a little bit heavier than just locking the condition variable, I use another common code pattern: *check, signal, check again.* I frequently use this technique with shared-memory synchronization throughout the kernel, such as with event delivery. The initial check for some condition is an optimization, and if it is true the code can skip the heavier-weight synchronization operation. When the condition is false, then we need to fall back to the "signal, check again", which provides the true synchronization, at a cost. This pattern is similar to the "test, then test and set" approach to spinlocks, which is when all contenders read the lock first, and only attempt a cacheline-invalidating write if they think they can grab the lock.

Chapter 4

Life for a Many-Core Process

In the previous chapter, I gave an overview of the MCP, but held off on many details. How does the kernel create and manage MCPs? What does userspace do when it finally gets cores? What are the benefits of dedicating cores to userspace? In this chapter, I discuss how the kernel and user work together to build and manage MCPs, and how userspace libraries can operate within the MCP abstraction. Finally, I will present the results of microbenchmarks related to performance isolation and user-level threading.

4.1 Procinfo and Procdata

Before diving in to how MCPs are made, it will help to know a little about the kernel interface. Since userspace is involved in the details of managing MCPs, there is a wide kernel interface with many rapidly changing parameters. A critical part of this interface is a pair of shared-memory regions called *Procinfo* and *Procdata*, both mapped into the user's address space at known addresses. Procinfo is read-only for the user, and Procdata is read-write. The MCPs use of a shared-memory region is similar to Psyche, which stands to reason: both systems provide a virtual processor.

The shared-memory interface has the obvious benefit of faster communication; it takes much less time to update a memory location than to perform a syscall. There are subtler benefits as well. Syscalls are convenient ways for userspace to initiate communication with the kernel, but it is more difficult for the kernel to communicate with the user. Upcalls are one such mechanism, but they can be as expensive as a syscall, if not more so, and upcalls can interfere with an application since they occur when the kernel wants, not when the user wants. Unlike syscalls or upcalls, shared-memory communication is bidirectional and requires no control transfer. The lack of control transfer means the kernel on one core can communicate with a process running on another core. Ultimately, both the user and the kernel can frequently update values without interfering with one another. Additionally, many of the types of information are only important at certain times; they are "update frequently, read rarely". For instance, the only time the kernel cares if the user enables software interrupts is when the kernel wants to send an interrupt. For all of these reasons, a shared-memory interface is very useful for the implementation of an MCP.

4.2 Virtual Cores

An MCP is made up of virtual cores (vcores), similar to virtual processors in Psyche, which are pinned to physical cores (pcores). These physical cores are designated as CG cores (coarse-grained) by the kernel scheduler. The mapping of virtual cores to physical cores is shared in Procinfo. The actual virtual core has the following fields:

```
struct vcore {
    uint32_t            pcoreid;
    bool                online;
    uint32_t            nr_preempts_sent;
    uint32_t            nr_preempts_done;
    uint64_t            preempt_pending;
    uint64_t            resume_ticks;
    uint64_t            total_ticks;
};
```

These virtual cores are laid out in an array in the MCP's Procinfo called the *vcoremap*. At any moment, userspace can determine if a virtual core is online or not, meaning mapped to a physical core, and on which physical core the vcore is running. Likewise, in Procinfo there is a *pcoremap*, which is a reverse mapping from physical cores to virtual cores, though a process can only see mappings for its own vcores. The tick counters accumulate how much time a vcore is online, so that userspace can run a virtual clock, if it so desires.

All virtual cores are not always running; that is impossible to guarantee generally for all processes on a system with shifting priorities. Instead, userspace can tell which vcores are running or not. When virtual cores are running, the physical core is completely dedicated to that virtual core. There is no per-core kernel runqueue, let alone time slicing. The kernel routes device and timer interrupts to LL cores, so the virtual cores receive no unexpected interrupts.

At this point, a virtual core sounds like a thread or process, albeit one with an interference-free and transparent scheduling policy. However, the nature of a virtual core is that it acts like a processor, and not a thread, in that the kernel can deliver a form of software interrupt. The code that runs on a virtual core is more like an operating system, even though it runs in user mode, and can disable these interrupts.

The defining feature of the virtual core is *vcore context*, which is a layer of software analogous to interrupt context in OSs. Its primary purpose is for event handling, thread scheduling, and other user-level scheduling operations. For a better understanding of vcore context, let us look at a structure in procdata corresponding to the vcore:

```
struct preempt_data {
    struct user_context         vcore_ctx;
    struct ancillary_state      preempt_anc;
    struct user_context         uthread_ctx;
    uintptr_t                   transition_stack;
    uintptr_t                   vcore_tls_desc;
    atomic_t                    flags;
    int                         rflags;
    bool                        notif_disabled;
    bool                        notif_pending;
    struct event_mbox           ev_mbox_public;
    struct event_mbox           ev_mbox_private;
};
```

These fields are writable by userspace and are sufficient to implement interrupt handling mechanisms. First, note that userspace tells the kernel its stack pointer and TLS (thread local storage) descriptor. The kernel does not trust the user, therefore it verifies the addresses are user addresses. Every time the kernel starts a vcore or puts it into vcore context as part of a software interrupt, the kernel initializes and runs a user context. The context uses the stack pointer and TLS from the vcore's preempt data structure. The program counter is taken from the program's ELF start point, called `_start` in glibc. The start code eventually calls an entry function called `vcore_entry()`.

The stack is reused, from the top, every time the vcore reenters vcore context. Clearly this means that userspace needs a way to disable this behavior. The act of the kernel placing a vcore into vcore context is called a *notification*, equivalent to hardware pushing the kernel into interrupt context. Userspace can disable notifications by setting the `notif_disabled` flag, equivalent to the kernel executing an instruction to disable interrupts. Whenever the kernel would like to notify a vcore, it sets the `notif_pending` flag, regardless of whether or not notifications are disabled. The kernel then only puts the vcore into vcore context if notifications are enabled.

The purpose of `notif_pending` is to prevent missed messages, or other circumstances which require the vcore to run, when enabling notifications. As far as the kernel is concerned, a vcore is in vcore context only if notifications are disabled. There are two ways out of vcore context: running a user thread (called a *uthread* in Akaros) or yielding to the kernel. The code for both takes special precautions to clean up the vcore, enable notifications, and then check `notif_pending`. If the flag is set, then vcore context is reentered. In the case of yielding to the kernel, `sys_proc_yield` simply returns.

Starting a uthread is slightly trickier: once notifications are enabled, the vcore's stack could be reused by a subsequent notification. The solution here is to use the uthread's stack. The code that starts a uthread's context (called "popping") uses the space *below* the uthread's current stack pointer for bookkeeping, and then changes to the uthread stack before enabling notifications. If a new notification arrives, the code that is interrupted, which will restart later, is the popping code that restarts the original uthread context. If `notif_pending` is set, which is checked after enabling notifications, the popping code will issue a system call to ask the kernel for help in restarting vcore context. To do so, the kernel will simply deliver a notification to the vcore. The uthread context that was being restarted is saved into procdata and the kernel restarts the vcore at its entry point.

One concern with using the stack below a uthread's stack pointer is that the application may be using that space, depending on the ABI. The code that pops a uthread's context assumes that memory below the stack pointer is scratch space. Application code must be compiled such that it does not save variables below the stack pointer. For instance, when compiling 64 bit x86 code, the "red zone" must be disabled.

Ultimately, a vcore will not leave vcore context permanently while `notif_pending` is set, ensuring the vcore code does not miss a message. The problem of missing messages while leaving vcore context is analogous to Saltzer's "wakeup waiter" problem[94], which covered the race conditions associated with a process sleeping while a concurrent signal arrives that wakes up the process. To complete the analogy, the `notif_pending` flag is Saltzer's "wakeup waiting" switch — a flag that alerts the sleeping or exiting code that a message arrived since the last time the flag was cleared.

The vcore preempt data struct also has storage for two contexts, each of which can hold the general purpose registers for a context. The two slots are used in different circumstances. Whenever the kernel notifies a vcore, meaning it places the vcore into vcore context, it first saves the previously running uthread into the `uthread_ctx`. The scheduler code, running in vcore context, can then save the context into the uthread struct or just restart it directly. The second context is for saving vcore context and is only used when the kernel preempts a core. When the kernel starts a vcore, it either builds a fresh vcore context or uses the saved context, based on `notif_disabled`. Simply put, if the core was in vcore context, which means `notif_disabled` is set, and is interrupted, it will restart exactly where it left off.

Another purpose of vcore context is to handle uthread faults and traps. When there is a trap, the kernel attempts to handle it. If it cannot, the trap is reflected back to userspace. The kernel passes the trap information in the uthread context and restarts the vcore in vcore context. One limitation to this model is that vcore context cannot have unhandled faults. Although this is easy for most faults, such as dividing by zero, page faults are more difficult. Vcore context is not allowed to have any hard page faults; the kernel will kill the entire MCP otherwise. Hard page faults are those that the kernel cannot resolve without blocking. Page faults on anonymous memory are soft faults, but accesses to mmapped files, such as code and data segments of a binary, can be hard faults. For this reason, low-level MCP library code pins all binaries. The extra requirement of userspace to manage memory is part of Akaros's memory and resource management philosophy: more opportunities for optimization, but more responsibility.

A final detail to note about vcore context is in how it compares to Psyche's software interrupt handling. In Psyche, there is an interrupt table of function pointers, and the kernel builds the user context to particular functions based on the interrupt number. Vcore context adds a level of indirection, where userspace can do whatever it wants, whether that is handling events, performing scheduler operations, or simply nothing at all.

A final detail to note about vcore context is in how it compares to Psyche's software interrupt handling. In Psyche, there is an interrupt table of function pointers, and the kernel builds the user context to particular functions based on the interrupt number. Vcore context adds a level of indirection, where userspace can do whatever it wants, whether that is handling events, performing scheduler operations, or simply nothing at all. The latter option is arguable the most important; spuriously entering vcore context is harmless, which is in accordance with my design patterns. As I mentioned before, allowing spurious messages allows for greater flexibility in solving synchronization problems. Additionally, vcore context's status as an intermediate layer, instead of being just the execution of a function in a table, allows for other uses. For instance, user threads can easily enter and exit vcore context by carefully setting and clearing `notif_disabled` and checking `notif_pending`, analogous to kernel code disabling and enabling interrupts. One example where a uthread must disable notifications is when acquiring a spinlock that can be acquired from vcore context. If notifications are not disabled, the uthread holding the lock could be interrupted and the vcore could try to acquire the lock, resulting in a deadlock. Kernel code must take the same precaution when acquiring locks that are also acquired in IRQ handlers.

4.3 MCPs from Creation to Entry

Running simple user code on dedicated cores can be done quite easily; send a kernel message or IPI and pop into userspace. It is more difficult to allocate cores to processes while also dealing with processes making system calls, yielding cores, and changing their allocation requests. To make things even more difficult, add in multiple processes with different priorities and the ability for the kernel to preempt cores. Suddenly giving out cores is not as simple, and there are a few subtle design points that can vastly improve the implementation. At all levels of existence, managing MCPs requires coordination between the kernel and the process. Let us start at the beginning.

Creation and Initialization

MCPs are just processes with extra modes, interfaces, and bookkeeping. Both MCPs and single-core processes (SCPs) use the same process structure, and for many parts of the kernel, such as the memory manager, there is little difference between the two. MCPs actually start out as SCPs, which gives userspace time and flexibility to prepare itself to be an MCP.

In contrast to the fork-exec model, all processes in Akaros are spawned in an unrunnable state with a `sys_proc_create()` call, which takes a program as its argument. Akaros actually has `fork` and `exec` syscall, only to temporarily support legacy applications. The parent process can pass file descriptors, if it so chooses. In the future, Akaros will have syscalls for the parent to set up shared memory regions with its child, or otherwise affect its address space. When the parent is ready, it calls `sys_proc_run` to make the child process runnable.

Spawning processes instead of forking is more appropriate in a multithreaded world, as pointed out by Topaz[67] and discussed in Section 2.2. Furthermore, explicitly passing references, such as file descriptors, to children avoids various security concerns associated with forking while concurrently opening files in another thread.

Once a process is created in SCP mode, it is up to userspace to perform low-level initialization. Typically, the first time a program attempts to make a multithreaded function call, such as `pthread_create()`, the user-level scheduler library will initialize itself and switch to MCP mode. The bare minimum initialization required is to allocate stacks and TLS regions for vcore context, for each vcore that the process will request. Additionally, the scheduler will usually create a uthread structure to track the initial SCP thread that starts at `main()`, referred to as “thread 0”. Optionally, the 2LS can initialize event queues and any other scheduler-specific data structures.

All initialization is supported by helpers in Uthread and Vcore libraries, which are part of Parlib. Parlib is a library that provides interfaces for user-level schedulers and helpers for common operations related to the Akaros kernel interface. Every binary on Akaros links with Parlib, in a way similar to how programs in Linux link with a C library. All second-level schedulers inherit from and link against uthread code. For example, Akaros has a pthread scheduler that is built on top of uthreads, and `struct pthread's` first element is a `struct uthread`. In this manner, a pthread in C code can be cast to a uthread — a common technique for object-oriented C code.

When the user-level scheduler is finally prepared, it asks the kernel to switch it to MCP mode. Changing to an MCP is a one-way deal; the process now runs on CG cores and is responsible for handling its own faults, scheduling threads, and handling events. Up until this point, the kernel could deal with many of these details. For instance, process wakeup and syscall completion code is very similar, regardless of whether a process is an MCP or an SCP. The kernel often treats an SCP as if it has only one vcore (vcore 0), which it uses for saving contexts and signalling notifications. There are certain steps SCPs can perform before becoming a full MCP to take additional control over its environment. Even SCPs eventually initialize themselves enough to perform basic operations in vcore context. For example, POSIX signals are delivered as events, and legacy SCPs need signals. However, this initialization only occurs after the majority of the C library initialization occurs: specifically, after ld.so mmaps the program binary and shared libraries. Until userspace performs any initialization, the kernel manages flags such as `notif_disabled`.

Allocating Cores to an MCP

One of the more distinctive aspects to being an MCP is that the process now runs on dedicated CG cores. By contrast, SCPs are typically time-sliced on LL cores; recall that in existing OSs, all cores are similar to LL cores, with per-core schedulers, interrupts, and time-slicing. The kernel manages CG cores with kernel messages, which can be sent from both remote cores and even a core to itself. At a very basic level, to give a physical core to a process, the kernel maps an unused vcore in the vcoremap, then sends a message to the physical core. The basic MCP management kernel messages are:

- `__startcore`: Run a process's vcore on a physical core. If the vcore was previously interrupted in vcore context, restart that context. Otherwise, start a fresh vcore context.
- `__notify`: If the process's vcore is not in vcore context, save the uthread context in procdata and start a fresh vcore context.
- `__death`: Stop running whatever context was previously running, discard the context, and prepare to idle the core.
- `__preempt`: Save the running context in the appropriate slot in procdata, and prepare to idle the core.
- `__set_cur_ctx`: Helper to set the current context, used by `sys_change_to_vcore()`, which is basically a kernel-assisted context switch between vcores.

Older versions of these functions directly altered the course of the kernel's execution. For instance, `__startcore` used to take a context and pop directly into userspace. The removing functions `__preempt` and `__death` would directly call code to clear the page table base register and just halt the core (`abandon_core()` and `smp_idle()`, respectively). None of these functions returned, so the type of the kernel messages had to be *routine*. As a reminder, routine kernel messages are executed when the kernel has finished its work, such as a syscall or IRQ handler, and it is safe to not return; see Section 3.2 for more details. Likewise, there used to be a few non-KMSG functions that did not return, such as `proc_run()` and `proc_destroy()`, for similar reasons. Ultimately, having these functions and KMSGs led to various complications, especially involving scenarios with vcores being preempted and restarted concurrently. As another example, functions like `proc_destroy()` were in situations where they would need to check messages and *possibly* not return if the core they were running on was a target of a `__death` message.

All MCP core and context management functions, which apply to a specific physical core's local state, now operate on a set of per-cpu variables. Instead of taking direct action, like popping into userspace, these functions set flags and operate on contexts so that the kernel will later perform the operation requested. Separating the functions from the final actions is a useful level of indirection. There are a series of relevant per-cpu kernel variables at play here:

- `struct proc *owning_proc`: Tracks which process owns the core. If the core is in userspace, then this is the process that is running. If the core is in the kernel, then this is the process that will run when the kernel finishes its work and returns to userspace.
- `int owning_vcoreid`: Tracks the vcoreid of `owning_proc` that will run on this physical core.

- `struct user_ctx *cur_ctx`: The user context for the current vcore. The pointer usually points to storage space in the per-cpu info struct; the user context is copied into this location during entry into the kernel. The current context is both the context that trapped into the kernel and the context that will be launched/popped when the kernel returns to userspace.

- `struct proc *cur_proc` (a.k.a., *current*): Tracks the process whose address space the kernel currently has loaded on the core. The current proc could differ from the owning proc, and it is usually used by syscall code to determine which process is executing.

With the indirection provided by these variables, `__startcore` does not directly run a particular context. Instead, it copies the context into `cur_ctx`, and later when the kernel calls `smp_idle()`, the kernel will pop the context and enter userspace. Importantly, the kernel only pops to userspace after handling *all* kernel messages. If there are a series of messages, such as a `__startcore`, followed by a `__preempt` and then a `__startcore` for another process, all of these commands will operate on the per-cpu variables, and the first process would never actually start up. Additionally, the physical core receiving the messages could also be in the middle of a syscall that affects the current context, such as yielding the current vcore.

The startcore-preempt-startcore example also shows why ordered, reliable messages are important. If those messages are delivered out of order, all sorts of havoc would ensue. In early versions of Akaros, I tried using interrupt vectors for each of the message types; that approach clearly fails due to the lack of ordering. The entire reason for developing the kernel messaging subsystem, described in Section 3.2, was to provide the reliable, ordered messaging needed for MCP code.

As mentioned earlier, old versions of the MCP management KMSGs did not return. KMSGs that do not return must be *routine*, meaning that they only run when the kernel can handle not returning: it has no invariants to restore, locks to unlock, memory to free, or work to complete. However, with the change to MCP kernel messages so that they always return and only modify per-cpu variables, do those kernel messages still need to be routine? The current process pointer does not matter either — kernel messages only change the owning process, not the current process. However, process management messages *do* need to be routine due to other invariants. I use routine messages, instead of immediate ones that take effect in interrupt context, so that syscalls and trap handlers can be assured that the vcoreid and current context does not change in the middle of an operation. Otherwise, it is possible for the user context that trapped into the kernel to be preempted and restarted on *another* core concurrently. Or if there was a preemption of the original vcore and a startcore for another vcore, a syscall could be confused about which context it should operate on. When using routine messages, syscalls that touch the vcoremap still need to be careful, but basic operations such as emulating the `rdtscp` instruction on x86 or a debugging syscall that returns the vcoreid all just work without needing to worry about concurrent preemptions. In short, deciding between an immediate or a routine message based solely on whether or not the message returns is insufficient — all invariants must be considered. It is critical to not use an immediate message that could violate an invariant. In this case, syscalls that have not blocked expect their vcoreid and current trapframe to not change.

Another per-cpu variable worth mentioning is `owning_vcoreid`. In early implementations of Akaros, code running on a core would simply look at the pcoremap, which maps physical cores to virtual cores, to determine the vcoreid. The problem is that concurrent preemptions may zero the pcoremap. Without `owning_vcoreid`, the unmapping of the pcoremap had to wait until the __preempt message runs on the target core. This requirement could lead to circular wait scenarios with the kernel scheduler. There is a potential deadlock where kernel scheduler code runs on a core that gets preempted and then attempts to restart that vcore somewhere else; the scheduler itself must run the handler to complete the preemption. There are a few other issues exposed by using the vcoremap directly, which require an understanding of how preemption works, discussed in Section 6.1. Preempting in this manner is a mess, and the simple fix is `owning_vcoreid` — a variable which makes sense in hindsight considering its similarity to `owning_proc`: together they describe what runs on a core. Note this means the vcoremap is not always up to date, and userspace could be confused. To deal with this concern, let us revisit allocating cores to a process.

When the kernel wants to give a physical core to a process, it picks an available vcore, writes the mapping into the vcoremap and pcoremap, and sends a `__startcore` message. All modifications to the vcore mapping are protected by a *seq counter* (pronounced "seek"). Seq counters are a common technique, like spinlocks, for read-mostly objects, where multiple readers can read concurrently with a single writer. Before and after writing, the writer increments the counter such that while a write is in progress, the counter value is odd. Readers check the counter before and after reading, and they repeat their read operation if the counter is either odd or not the same during both reads. In this manner, readers can atomically perform multi-word reads. In the case of the kernel changing the vcoremap, userspace can tell that a change is in progress and wait until the kernel is complete. Otherwise, there could be moments of inconsistency, such as the vcoremap disagreeing with the pcoremap.

To pick an available vcore, the kernel consults the per-process vcore lists: online, bulk-preempted, and inactive. The online vcores are those vcores currently mapped and executing user code, or soon to be executing and with a kernel message in-flight. The distinction between bulk_preempted and inactive help with preemption and event delivery, discussed in Section 6.1. For our purposes now, the kernel can just take a vcore from one of these lists, preferring bulk_preempted.

When it comes time to send the kernel messages to start the vcore, the kernel needs to deal with reference counting. As mentioned earlier, processes use krefs to keep the process alive until its last external reference is freed. Reference counting is more difficult with MCPs than with 1:1 threading models. In 1:1 models, the process is just one kernel thread with its userspace component, but with MCPs, there is one kernel object for the process and code using that object running in parallel. The various per-cpu variables, such as `cur_proc` and `owning_proc` each count as a separate reference. In case it is not clear, a process cannot be freed while a core is still running in that process's address space and using its page tables. To avoid a flurry of cache-line contention on the process's kref variable, the kernel preemptively ups the reference count by two for each new core allocated to the process. In the unlikely case that this was too many, such as if a `__startcore` was sent to a process that already had `cur_proc` set correctly, the KMSG handler notices and drops the extra reference.

MCP reference counting can be tricky, primarily due to functions that do not return. If a function takes a reference counted object and does not return, then it must decrement the reference count. In the old versions of the codebase, functions like `proc_destroy()` may or may not return, the uncertainty of which was a programming hassle. Callers needed to wrap calls to these functions with increfs and decrefs. Reducing the number of MCP functions that might not return simplifies reference counting greatly. The major change to reduce the number of these functions was the layer of indirection between MCP management KMSGs and actions like actually popping into userspace. Again, this relatively simple technique greatly simplifies the implementation of the MCP management code. Currently, there is only one function that takes a process reference and might not return: `proc_yield()`. Yielding a vcore might fail and return to userspace if notif_pending is set, which is how the kernel enforces the policy that userspace does not leave vcore context when a notification is pending.

Finally, the kernel has picked a physical core, mapped it to a free vcore, properly reference counted the MCP, and sent the `__startcore` kernel message. When userspace starts up on a CG core, it will start in vcore context. If the vcore was previously in vcore context and preempted, which is detectable by inspecting `notif_disabled`, that context is simply restarted. Otherwise, the kernel initializes and starts a fresh vcore context, regardless of whether or not a uthread was running previously. An alternative approach, and the one I originally pursued, was to restart whatever context was previously running, either vcore context or not. In this manner, a preemption and a restart would act like a time-slice in the usual OS process management style. There are two drawbacks. First, userspace is responsible for its scheduling and resource management; if the kernel stops and restarts a vcore, userspace should get a chance to do something about it — perhaps adjust its scheduling policy. More importantly, userspace *needs* a chance to interpose on the automatic restart of a previously preempted context, as explained in greater detail in Section 6.3. Regardless, userspace is finally in control of its vcore, and can schedule threads or do whatever it likes.

4.4 User-level Scheduling

Once the kernel granted gang-scheduled cores to an MCP, it is up to userspace to determine what to do with its virtual multiprocessor. Userspace is aware of which vcores are online, and it must choose what code runs where. Many applications will want some form of user-level threading, and the Uthread library contains the infrastructure and abstractions common to most all thread-scheduling libraries.

Applications do not need to use the Uthread library, nor do they even need to use threads at all. Threads are relatively cheap, but an application could use the MCP environment to run an event-driven program, managing events and issuing asynchronous syscalls directly to the kernel. If an application chooses this route, I would consider still using uthreads and a very simple scheduler with one uthread per vcore, since the uthread code does so much heavy lifting for the application.

Connecting User-level Schedulers to Uthreads

At this point in time, the only full-fledged user-level scheduler I wrote is a pthread scheduler, so I will explain how user-level threading works as a combination of pthread and uthread code. Let's start with the basics: the thread structures.

```
struct uthread {
    struct user_context u_ctx;
    struct ancillary_state as;
    void *tls_desc;
    int flags;
    int state;
    struct syscall *sysc;
    struct syscall local_sysc;
    void (*yield_func)(struct uthread*, void*);
    void *yield_arg;
    int err_no;
    char err_str[MAX_ERRSTR_LEN];
};
extern __thread struct uthread *current_uthread;
```

A uthread is little more than its context, including floating point, and bookkeeping. A uthread's context may or may not be in the uthread struct; when it runs, the context is in the processors registers, and when it is interrupted, its context is saved in procdata. Thread-local storage (TLS) is actually optional for uthreads; specific schedulers can ask for it on a per-thread basis, though "thread 0", created by glibc, always has a TLS. Note that a uthread has `err_no`. When code accesses the POSIX `errno` variable, for uthreads this uses the variable in the uthread struct. For vcore context code, `errno` uses storage in thread-local storage, as is customary in glibc. All vcore contexts have a TLS, which stores the most important variable to a scheduler: `current_uthread`. `current_uthread` tracks which uthread runs on a vcore and can be used by any thread to get a pointer to its thread control block.

Many of the parts of the uthread struct are important, but what is also important is what is *not* there. There is no stack in the uthread struct. There is a stack *pointer*, saved in the context, but there are no other assumptions about the stack: that is the responsibility of the specific second-level scheduler that links with uthreads. For example, Capriccio[9] and Go both use linked stacks, so assuming the stack is a contiguous block with a particular stacktop would be inappropriate. In contrast, the pthread struct has a stacksize and stacktop:

```
struct pthread_tcb {
    struct uthread uthread;
    TAILQ_ENTRY(pthread_tcb) next;
    int state;
    bool detached;
    struct pthread_tcb *joiner;
    uint32_t id;
    uint32_t stacksize;
    void *stacktop;
    void *(*start_routine)(void*);
    void *arg;
    void *retval;
    sigset_t sigmask;
    sigset_t sigpending;
};
```

The uthread struct is the first element of the pthread struct, allowing the casting between uthread and pthread structs. In this manner, pthreads inherit from uthreads. Pthread code also handles other thread-related details that may be specific to a particular scheduler, such as signals and joining. Embedding the uthread struct in the pthread is one manner in which schedulers interact with uthread code. Another primary way is through an array of function pointers for scheduler operations called `schedule_ops`. These functions are a collection of upcalls from uthread code into a specific scheduler.

```
struct schedule_ops {
    void (*sched_entry)(void);
    void (*thread_runnable)(struct uthread *);
    void (*thread_paused)(struct uthread *);
    void (*thread_blockon_sysc)(struct uthread *, void *);
    void (*thread_has_blocked)(struct uthread *, int);
    void (*thread_refl_fault)(struct uthread *, unsigned int, unsigned int,
                              unsigned long);
    void (*preempt_pending)(void);
    void (*spawn_thread)(uintptr_t pc_start, void *data);
};
```

The most important of the scheduler operations is `sched_entry()`. I will describe the others below when they are relevant to particular scenarios or operations. The `sched_entry()` is the entry point called when a vcore starts up. When a vcore starts, either for the first time or in response to a notification, it actually starts at glibc's `__start` symbol, which loads the vcore's TLS and calls `vcore_entry`. Unless an application overrides various weak symbols in glibc, `uthread_vcore_entry` executes. In `uthread_vcore_entry`, there are various checks necessary for handling preemption, which I describe later. The remainder is merely a quick call to `handle_events()`, followed by calling the 2LS-specific `sched_entry()`. In the case of pthreads, `pth_sched_entry()` runs any previously running `current_uthread`, which would be the case if a pthread was interrupted for a notification. Otherwise, the pthread scheduler simply picks a pthread and runs it. Schedulers can customize their `sched_entry()` however they see fit. Perhaps they use a task bag for their threads or per-vcore runqueues. If there are no threads to run, application-specific knowledge can determine if a vcore yields and returns to the kernel or if it spins or halts temporarily if new work is anticipated.

Vcore Context Details for User-level Schedulers

The `sched_entry()` functions are one of many that operate in vcore context. Vcore context is a low-level environment, and it has rules.

- No code can busy wait on user code without dealing with preemptions, because the vcore it waits on may be preempted. Code that requires spinlocks should use the PDR locks, described in Section 6.2. Custom busy waiting code must call `cpu_relax_vc()`, passing the vcoreid of the vcore it waits on, if known, and itself otherwise. The function `cpu_relax_vc()` will periodically ensure the target vcore is online and react accordingly, discussed further in Section 6.2.

- Once you leave vcore context, there's no going back. Vcore context code starts at the top of its stack, every time the kernel starts a fresh vcore (when notifications where enabled) or a uthread enters vcore context. If vcore context code leaves vcore context, it must never return by any mechanism, though free restarts are acceptable. In these circumstances, for example, no locks can be held. The vcore's state, such as TLS variables, must be left ready for the next fresh start. Code in vcore context cannot call a helper to reenter vcore context and expect to return, since the original code's stack will be clobbered.

- Calls to `handle_events()` might not return. Certain preemption scenarios are simplified by allowing a vcore to restart, as discussed further in Section 6.3.

- No blocking. When a uthread blocks, it is managed by vcore context. There is no context that can manage when vcore context blocks. Any blocking syscall will result in a busy wait on the kernel to complete the syscall. User-level blocking, such as when a uthread blocks on a mutex, is implemented by dropping into vcore context. If vcore context code tried calling a uthread helper to block, it would clobber its own stack and never return properly.

- No faulting. When uthreads fault or trap and the kernel is unable to satisfy the fault in a non-blocking manner, vcore context is responsible for resolving the fault. If vcore context faults, then the kernel kills the process. Other systems, such as Scheduler Activations, rely on the kernel to suspend the faulting scheduler context. Since MCPs are responsible for memory management, it is not unreasonable to have the MCP pin the code and data pages of binaries it accesses. This pinning happens in the early phases of process initialization.

- Limit recursive calls and stack consumption. Vcore context runs on a small stack, much like kernel threads and IRQ handlers. Ideally, code will spend a very limited amount of time in vcore context.

- Entry and exit only through authorized helpers. Entering and exiting vcore context is very tricky. Users are highly discouraged from "rolling their own". Acceptable ways to leave vcore context include yielding to the kernel or running a uthread: both of which are careful to not miss notifications. Entering vcore context from user code typically happens by only one function: `uthread_yield()` described below.

Embedding the uthread struct in a scheduler-specific thread struct was one aspect of the interface between a 2LS and the uthread library The second aspect was the table of upcalls from uthread to the 2LS. The final aspect is a collection of helpers an application can call into the uthread library. Two of the more common helpers are `run_uthread()` and `uthread_yield()`. Helpers to run a uthread typically manage the uthread's state, load the floating point context if appropriate, load the TLS, and pop into the uthread's context. Yielding is more complicated.

There are many reasons for a uthread to enter into vcore context. Originally, there were two that all 2LSs supported: cooperatively yielding and exiting.

Additionally, the pthread scheduler wanted to enter vcore context for joining and blocking on barriers and condition variables. To simplify 2LSs, all of these methods were condensed into a single yield call, which takes a callback that is later run in vcore context. `uthread_yield()` has many moving parts and is worth seeing in its entirety:

```
void uthread_yield(bool save_state, void (*yield_fn)(struct uthread*, void*),
                   void *yield_arg)
{
  struct uthread *uthread = current_uthread;
  volatile bool yielding = TRUE;
  uthread->yield_fn = yield_fn;
  uthread->yield_arg = yield_arg;
  uthread->flags |= UTHREAD_PINNED;
  uint32_t vcoreid = vcore_id();
  struct preempt_data *vcpd = vcpd_of(vcoreid);
  disable_notifs(vcoreid);
  if (save_state) {
    uthread->flags |= UTHREAD_SAVED;
    save_user_ctx(&uthread->u_ctx);
  }
  if (!yielding)
    goto yield_return_path;
  /* From here on down is only executed on the save path (not the wake up) */
  yielding = FALSE;
  if (save_state && (uthread->u_ctx.type != ROS_SW_CTX)) {
```

```
21      save_fp_state(&uthread->as);
22      uthread->flags |= UTHREAD_FPSAVED;
23    }
24    if (__uthread_has_tls(uthread))
25      set_tls_desc(get_vcpd_tls_desc(vcoreid), vcoreid);
26    else
27      __vcore_context = TRUE;
28    set_stack_pointer((void*)vcpd->transition_stack);
29    __uthread_yield();
30    assert(0);
31  yield_return_path:
32  }
```

There are many important details to `uthread_yield()` that touch on aspects of user threading and the complexity of dropping into vcore context.

- Note the `UTHREAD_PINNED` flag at line 8. Any time a uthread needs to know which vcore it is on for correctness reasons, such as in this example when it disables and enables notifications, it must pin itself to the vcore, meaning that the scheduler will not migrate it to another vcore. Without pinning, if this uthread were to read the vcore id (from TLS) at line 9, but then get interrupted, preempted, and migrated to another vcore before line 11, it would operate on the wrong vcore's `notif_disabled` flag. For more details on `UTHREAD_PINNED`, refer to Section 6.3.

- `save_user_context()` at line 14 will return twice, once right after the call, and again when the uthread is restarted. The volatile stack variable `yielding` determines which path to take when the function returns each time.

- Floating point state is conditionally saved, depending on whether or not the architecture port has support for software contexts yet. Akaros has two types of contexts: hardware and software. Hardware contexts are generated by interrupts or traps; these tend to be unexpected and parts of their format may be determined by hardware. Software contexts are much smaller, and their format is determined by software. For x86, I used the AMD SysV ABI calling convention for software contexts. Hardware contexts do not contain floating point state, but software contexts contain the minimum required by the ABI: the `mxcsr` register and FPU control word. Relying on the ABI in this manner means that function calls outside a compilation unit, such as a call to `uthread_yield()`, have no expectation of the FPU being saved, other than `mxcsr` and `fpcw`, and in these cases we do not need to save or restore the entire FPU's state. The kernel knows how to start and stop software contexts too: x86 syscalls made with `sysenter` use software contexts when interfacing with the kernel for simplicity and performance.

- Thread local storage is optional, on a per-uthread basis. Thread 0, which is the one that runs `int main()`, will have a TLS. Others do not require it, and for some machines not using TLS speeds up performance. If a uthread does not use TLS, it uses its current vcore's TLS for variables such as `current_uthread`, `__vcoreid`, and `__vcore_context`.

- Lines 24-28 are where the uthread enters vcore context: we change to the vcore's stack and TLS. Notifications are disabled so that this transition is atomic. Note that if the kernel interrupts the vcore during this phase, the vcore, and hence the uthread, will be restarted as if it was not interrupted.

The last bit of `uthread_yield()` calls out to a helper function, mostly to convince the compiler to not use any stack local variables after we jumped to the vcore's stack. The bottom half is relatively simple:

```
static void __attribute__((noinline, noreturn))
__uthread_yield(void)
{
  struct uthread *uthread = current_uthread;
  uthread->flags &= ~UTHREAD_PINNED;
  uthread->state = UT_NOT_RUNNING;
  uthread->yield_fn(uthread, uthread->yield_arg);
  current_uthread = NULL;
  uthread_vcore_entry();
}
```

The yield function is a 2LS's opportunity to perform certain tasks. For a simple cooperative yield, the 2LS can place the uthread on a run queue. When blocking on a mutex, the 2LS can put the uthread on the queue in the mutex's struct. It is very important to not touch the uthread once we call the yield function; the uthread could be concurrently restarted on another vcore. Once the yield is complete, we simply restart the vcore from its entry point.

Vcore context has a specific meaning: code running on a vcore's stack in its TLS. Uthread code and vcore code can tell if it is actually in vcore context by examining the TLS variable `__vcore_context`. But from the perspective of other cores or the kernel, vcore context is defined by the `notif_disabled` flag. If notifications are disabled, the rest of the system will treat a context as if it were vcore context. For instance, if that context is preempted, it will be restarted at some point in the future where it left off. Conversely, if code that is in the *actual* vcore context (stack and TLS) enables notifications, then the kernel could notify that vcore and clobber the vcore's stack. Clearly that should never be done.

The state of *appearing* to be in vcore context is very useful for user threads. It is equivalent to disabling interrupts in an operating system, which is necessary for grabbing spinlocks that could also be grabbed from IRQ or vcore context. The uthread library provides helpers for disabling and enabling notifications. These helpers pin the uthread to the vcore and then disable notifications. After reenabling notification, they check the `notif_pending` flag to see if it missed a notification. If so, it needs to reenter vcore context fully; the easiest 2LS-independent way is to ask the kernel for a self notification. When a uthread has disabled notifications, it must abide by a few rules, even though it is not actually in vcore context. Any faults will kill the process, since the kernel thinks the vcore is in vcore context. Blocking syscalls or yielding can be problematic; when the uthread restarts, notifications will be reenabled and its vcoreid may have changed, which can confuse the uthread's code. To avoid serious bugs, if a uthread is pinned or has disabled notifications and attempts to block on a syscall, then it will behave the same as vcore context: spin until the kernel completes the call.

Blocking Syscalls

A critical function for user-level schedulers is to handle blocking system calls. Capriccio[9] and many other systems adapted to legacy OSs interposed on common library calls, such as `read()` and `write()`, and called asynchronous versions from within their runtime. On Akaros, the job of a user-level runtime is simpler since all system calls have an asynchronous interface.

Akaros provides synchronous and asynchronous helpers for issuing system calls; uthreads and even vcore code predominantly use the former. If a syscall blocks, the kernel may or may not block a kthread internally. Regardless, control returns to userspace, and the synchronous helper inspects the syscall struct, sees the syscall is not complete, and calls out to the uthread library. After a few checks to make sure the uthread is in fact an unpinned uthread, the blocking helper calls `uthread_yield()` with the 2LS's `thread_blockon_sysc` as the yield callback.

The 2LS is free to do whatever it likes with the uthread and the syscall it is blocked on. The pthread scheduler takes the mundane approach of removing the pthread from the list of active threads and registering for an event when the syscall completes. From the perspective of the scheduler, either registration succeeds and an event will fire when the syscall completes, or registration fails because the syscall completed already. When the "syscall complete" event fires, the pthread scheduler puts the thread back on the run queue.

Other schedulers can pursue different policies — this space is very much up for customization for particular schedulers. Certain schedulers will want to prioritize unblocked threads, perhaps based on a thread's priority or some deadline. Some schedulers may not even want an event, choosing instead to poll syscalls for completion. Additionally, the *type* of event can be customized for different schedulers. Pthreads has per-vcore event queues and requests a notification (IPI) for a completed syscall; others may have a global event queue or no IPIs. I discuss the details of event handling and its various options in Section 5.2.

The act of registering for an event with a syscall requires a special uthread helper that synchronizes with the kernel's syscall completion code:

```
bool register_evq(struct syscall *sysc, struct event_queue *ev_q)
{
  int old_flags;
  sysc->ev_q = ev_q;
  wrmb();
  do {
    old_flags = atomic_read(&sysc->flags);
    while (old_flags & SC_K_LOCK)
```

```
            old_flags = atomic_read(&sysc->flags);
        if (old_flags & (SC_DONE | SC_PROGRESS)) {
            sysc->ev_q = 0;
            return FALSE;
        }
    } while (!atomic_cas(&sysc->flags, old_flags, old_flags | SC_UEVENT));
    return TRUE;
}
```

The bulk of the synchronization between the kernel and user is on the syscall struct's `flags` field. When the `SC_EVENT` flag is set, the kernel sends an event to `sysc->ev_q`. Userspace performs a compare-and-swap to make sure that it atomically sets the `SC_EVENT` flag so long as the syscall is not complete. For its part, the kernel sets the `SC_DONE` and `SC_K_LOCK` flags, checks for the event flag, then clears `SC_K_LOCK`, using atomic OR instructions.

The reason for the `SC_EVENT` flag is that the kernel needs to atomically complete the syscall and check for an event queue. Userspace needs to atomically sign up for an event while making sure the syscall has not already completed. The atomicity of operations on the syscall's flags allows us to satisfy these requirements.

One technique to note is that the kernel does not actually perform a compare-and-swap. If the kernel has a compare-and-swap loop with the user, a malicious user could attempt a small denial of service by constantly changing the value of `old_flags`. As a general principle, I never have the kernel operate in a loop controlled completely by the user. I call the technique used here in `register_evq()` "CAS-ing with the Kernel". The kernel sets the `SC_K_LOCK` flag, which userspace interprets as an instruction to busy wait on the kernel. CAS-ing with the kernel here adds a small amount of overhead for both the user and the kernel in exchange for a small amount of peace of mind. The tradeoff may or may not be worth it. I use the K_LOCK technique in a few other places, notably in dealing with preemption, discussed in Section 6.1.

The main points behind uthreads blocking on syscalls is that userspace does not lose control of its core and that the application's scheduler is completely in control of what to do. Certain use cases may desire to use the raw asynchronous interface directly, but for applications that prefer a basic threading abstraction, uthreads should suffice. It should also be noted that a uthread's syscall can easily be aborted, as discussed in Section 3.2, if a scheduler or application desires this. The important part is that the user is in control of its core, regardless of what the kernel does internally. The kernel may or may not spawn a kthread for a blocked system call, but userspace does not care: the contract with the kernel is the syscall struct, and userspace does not lose its core.

Blocking Faults and Traps

Apart from blocking syscalls, the other classic problem with user-level threading is dealing with faults, especially page faults. System calls are expected and instigated by the user, and thus are capable of being controlled by the user, as in the case of Capriccio interposing on library calls. In contrast faults are often unexpected and much harder to handle.

One approach would be to allow no faults whatsoever; in the case of memory, pin every page the application mmaps and then never fault. Unfortunately, this approach also removes our ability to exploit virtual memory for a variety of benefits. For one, we could no longer mmap files without populating the entire file at map time. All anonymous memory regions would be pinned, which is a waste of memory. No faulting rules out alternative types of mmapped memory regions where the OS can reclaim memory arbitrarily and the application recreates the lost data on demand. Finally, certain applications *plan* to fault on certain memory accesses. For instance, the Go runtime occasionally throws a segmentation fault in valid circumstances; it catches it and moves on as part of normal operations. Simply ignoring all faults is not an option.

The approach I took in Akaros is to allow the kernel to attempt to handle a fault before reflecting unhandled faults back to userspace. If the kernel can handle a fault without blocking, then the user's context is simply restarted as if nothing happened. For instance, if there is a page fault on an anonymous memory region, the kernel usually can allocate a page, zero it, and map it into the users address space without blocking. For file-backed memory regions, the backing page may already be in the page cache; if so, the kernel merely needs to map the page into the page table.

However, if the kernel cannot handle the fault without blocking, it passes it back to userspace. To do so, the kernel squeezes the fault information into the user context, and then notifies the vcore: meaning it acts as if the vcore received a `__notify` kernel message, saving the previous context (with fault information) in procdata and starting a fresh vcore context at `vcore_entry()`. If the fault was from vcore context, the kernel just kills the process; vcore context code should not fault. Once in vcore context, the fault information is coupled to the uthread and its context. Since the vcore could be dealing with a preemption event concurrently, there is the possibility that its `current_uthread`, which just faulted, might not get run immediately. For this reason, we delay the handling of the fault until the next time the uthread is started. Specifically, the `run_uthread()` helpers check the context for fault information before running the uthread, and if there was a fault, the uthread code upcalls to the 2LS scheduler operation `thread_refl_fault()`.

Once the specific second-level scheduler receives the fault, it can do whatever it likes. The pthread scheduler checks the fault information and reacts accordingly. If the fault was for a file-backed mmapped region, the scheduler issues a raw asynchronous syscall to load the page cache and populate the virtual address. In essence, the kernel could not resolve the fault without blocking, so userspace issues a blocking syscall to resolve the fault. In a sense, the page fault is an implied syscall in traditional operating systems that block and service faults. In Akaros, the syscall is explicit. Most schedulers should perform the same steps for page faults on file-backed regions. Other faults can be handled in a 2LS-specific manner. For instance, pthreads sends a `SIGSEGV` signal for other page or general protection faults and a `SIGFPE` for divide by zero.

Similar to how Scheduler Activations uses the activation to solve all 2LS problems, Akaros uses vcore context to deal with blocking calls, page faults, and thread context switches. Next, I will show some of the benefits of the user-level threading.

Thread Scheduling Microbenchmarks

One of the major, well-known benefits of user-level threading is fast context switches. Although it is no surprise that user-level threading is fast, it is worth making sure Akaros's MCPs have fast threading.

The following tests were run on an Intel Xeon E5-2670, 2.6GHz. This Sandy Bridge machine has two processors, each with 16 hyper-threaded cores (Symmetric Multi-Threading), for a total of 32 hyper-threads. Each processor has a 20 MB L3 cache. The machine has 256 GB RAM, and a few NICs. The NIC, used in later experiments, is a 1 Gb Intel I350 (PCI Device Id 0x1521). Speedstep, Turbomode, and related technologies are disabled, as is Advanced Cacheline Prefetching (which basically increases the size of a cache line to 128 bytes). This machine is referred to as "c89". All Linux tests were run on a 3.11 kernel and stock Ubuntu. Unless otherwise specified, Linux test programs are pinned to cores, and IRQs are routed away from those cores (if possible). The context switch microbenchmarks were run on Akaros commit 0b940e7e.

The basic thread context-switch latency test is to launch a specified number of pthreads on a single physical core, each of which yields for a specific number of loops:

```
pthread_thread() {
  for (int i = 0; i < num_loops; i++)
    pthread_yield();
}
```

The various configurations tested include whether or not to use TLS (when available) and choices between stock pthreads and uthread-backed pthreads on Linux. Kevin Klues ported parts of Parlib, including uthread and vcore code, to Linux. Basic uthreading works, and vcores are emulated by Linux tasks. Features critical to the MCP are not supported, such as exposing the vcoremap to userspace and handling preemption. Regardless, it provides a basic, comparable platform for user-level threading on Linux for experimentation.

The results for each configuration are shown in Table 4.1. Each test ran for 10,000,000 total loops and I report the average time per context-switch.

There are two major points to make here: TLS has an excessive effect on this machine, and Akaros's default pthread scheduler is inefficient. I discuss each point in turn and cover specific improvements below.

Depending on a machine's architecture, changing the processors TLS descriptor can be expensive. On RISC-V, for instance, TLS is simply a register and that has very little overhead to change. On x86, the cost for changing TLS depends on the machine.

As a reminder, the TLS descriptor is a segmentation register that provides a window into the address space before the page table translation. For an x86-based example, a virtual address using TLS will specify a register and an offset, such as `fs:0x123`. Segmentation

Table 4.1: Average Thread Context-Switch Latency (nsec)

	Linux Pth with TLS	**Linux Uth with TLS**	**Linux Uth w/o TLS**	**Akaros Uth with TLS**	**Akaros Uth w/o TLS**
1 Thread	**254**	474	251	340	174
2 Thr.	**465**	477	251	340	172
100 Thr.	660	515	268	**366**	**194**
1000 Thr.	812	583	291	408	221

hardware translates this virtual address into a linear address. If the base of the segment specified by `fs` is 0x456000, then the linear address is 0x456123. The linear address is passed to the paging hardware, which translates the address to a physical address. The usage of "virtual address" for "linear address" is quite common, even when talking about x86 machines, since most address references have identity-mapped segments (for instance, all non-TLS addresses).

On older x86 machines, typically the kernel sets the TLS just like any other segmentation register: by writing the base value into a descriptor table, such as the Local Descriptor Table (LDT), and writing a segmentation register to point at that descriptor entry. Writes to the LDT are cheap, and writes to the segmentation register are cheap. But the combination of the two, if either value is a modification instead of rewriting the old value, triggers a hardware-based "segmentation walk." Old versions of Akaros (32 bit) had an insecure mechanism to change the TLS from userspace using this technique. The insecurity comes from allowing access to the LDT, since the table can include call gates and other non-addressing-related fields. Even when performing the change from user-space, the segmentation walk takes around 70-80 nsec — entirely too slow.

Slightly newer x86 machines provide a model-specific register (MSR) to directly write the TLS base register. These registers bypass the segmentation walk and are faster than setting an LDT entry. Unfortunately, access to MSRs such as `MSR_FS_BASE` is limited to the kernel only, and writing to MSRs is relatively slow even for the kernel.

Newer x86 machines, starting with Ivy Bridge, provide new instructions that allow both the kernel and userspace to set the TLS base registers. These registers bypass the segmentation walk and do not write to MSRs, resulting in very fast TLS changes. Unfortunately, the machine I use (c89) is Sandy Bridge, and does not have this capability. To deal with this, I built a fast-call mechanism into Akaros's syscall handling. The normal syscall path involves saving registers, jumping to a kernel stack, and calling into dispatch code, but the fast-call path simply changes the TLS base and returns. For those curious, I use a non-canonical value for the syscall struct pointer to signal the fast-call; no legitimate syscall could ever accidentally submit that address. The end result is that Akaros's TLS changes, like any OS on an x86, non-Ivy Bridge machine, requires a syscall to change the TLS.

To see the effects of TLS on context-switches, let's look at Table 4.1. Note the difference between one thread and two threads on Linux pthreads. The Linux kernel avoids changing

TLS unless it is necessary. There is no such benefit with the uthread library. On both the Linux and Akaros versions of Parlib, if a uthread has TLS, then there are TLS descriptor changes for every context-switch, even if there is only one thread. The reason is that vcore context has its own TLS that shares the same TLS register as a uthread. When compiling with GCC, all user code uses the same register for TLS (e.g. `fs` on 64 bit x86), while the kernel uses another register for its TLS. Dropping into vcore context as part of the yielding operation from a uthread with TLS will require a TLS change to the vcore's TLS.

Akaros's uthread library lets the application or 2LS pick whether or not a particular uthread has TLS. For Linux uthreads, TLS support is a compile-time option. In either case, if the uthread does not have a TLS, the vcore's TLS will remain loaded during the uthread's execution. Ideally all hardware would support fast TLS changes, but since not all applications need TLS, Akaros will have the option to turn it on or off. Of note, the Go runtime uses TLS internally, but its user-threads (called go routines) do not need TLS.

The other significant feature from Table 4.1 is that even without TLS, 194 nsec for a context-switch is very high. The reason is that this version of Akaros's pthread scheduler was inefficient, somewhat by design. One of the major purposes of the pthread scheduler is to find bugs by doing things that schedulers *might* want to do, even if they are not ideal. Correctness first, then performance. For example, the 2LS not only has a global runqueue, but also maintains an "active threads" list. Although the global runqueue does not matter when running on a single vcore, every time a pthread yields, the lock protecting the lists is grabbed multiple times.

Table 4.2: Akaros Context-Switch Latency (nsec)

	With TLS	**w/o TLS**	**No Sched Locking**	**No Locking, No Asserts**	**Switch-To**
2 Thr.	340	172	95	88	55
100 Thr.	366	194	113	105	N/A

Table 4.2 shows the effect of various enhancements to the Akaros pthread scheduler. Removing locking and asserts (of which there are many) cuts down on half of the overhead of a context-switch. The Switch-To column tests a direct context switch, which bypasses the 2LS decision-making process. In this case, the pthread still calls `uthread_yield()` to changing into vcore context, but in-lieu of calling `vcore_entry()`, it simply pops another thread. As a note for people implementing switch-to in their 2LS, skipping `vcore_entry()` could miss an event. In that case, `notif_pending` is still set and we will notice and recover when we attempt to leave vcore context when popping the user's context. There may be more opportunities for tuning Akaros's user-level context switching; at the extreme-end, `setjmp()` and `longjmp()` are very fast and have no bookkeeping, scheduling decisions, or even entry into vcore context. It is up to an application or other custom 2LS if they want to pursue more aggressive approaches.

4.5 Impact of Dedicated Cores

One of the major benefits to the MCP is that the kernel dedicates physical cores to an application, providing an interference-free platform. The process interacts with these cores through the vcore abstraction, but ultimately there is a pinned, unvirtualized, physical resource under the application's control. Applications want the performance of a dedicated machine; the MCP is the abstraction that provides it. The guarantee to userspace is that when cores are granted to a process, the kernel will not interrupt, timeslice, or otherwise interfere with a core, without the process's permission. Of course, cores can be preempted, but the process is alerted to this fact, and the kernel will not revoke cores that were provisioned, as discussed in Section 3.2. Even non-provisioned cores are unlikely to be preempted. The current Akaros kernel scheduler is provisioning, first-come-first-served (FCFS) scheduler, meaning that it will preempt to satisfy provisioning requirements, but otherwise will only allocate idle cores. The stated policy for future MCP kernel schedulers is that any time-slicing of MCPs will be done at a coarse granularity.

As I discussed in Section 2.5, prior work in the HPC community showed how important performance isolation is to parallel processes. OS noise, in particular, can have detrimental effects, especially on synchronizing programs at large-scale. In one anecdote from Ron Minnich, a printer daemon that woke up every 30 seconds lowered the performance of an application on a large cluster by 25%.

The classic approach for measuring interference is to read the time, do a fixed amount of work, then check the time again. Any variations in the elapsed time is a sign of interference. Although this approach is correct, it misses out on a critical aspect of interference: it is often periodic.

To not only detect noise, but to analyze its root causes, Sottile and Minnich developed the Fixed Time Quantum (FTQ) benchmark[101]. FTQ measures how much work is done in a small, fixed, time quantum, over a larger time interval. The main parameters to FTQ are the frequency of the sampling and the number of samples, and the output is a collection of data points of the form {timestamp, work_amount}. The most important step is to run a Fast Fourier Transform on the results to help determine the causes of interference. Note that the output's work_amount value is not as important as its predictability; {25, 25, 25} is less noisy than {2, 100, 2}.

FTQ Results and Analysis

In this section, I present the results of FTQ on Akaros and Linux 3.11 on the machine "c89", described above, and on a machine called "hossin", described farther below. All runs are on a single core, usually core 7 or core 31. Core 7 is on the same socket as core 0, which tends to have higher interference on both Linux and Akaros. Core 31 is on another socket. Unless otherwise stated, the cores are MCP/CG cores for Akaros, and for Linux FTQ is pinned to an otherwise idle core for which interrupts are routed away (from it and its hyper-threaded sibling). On Akaros, the 2LS is instructed to never yield its vcore, nor ask for any future vcores; this does not matter much for a single-core run of FTQ, but it can have effects on runs across the entire machine. FTQ is set to run at a frequency of 10,000 Hz, which is a quantum of 100 usec. Note that quanta are rarely exactly 100 usec. The benchmark reads the timestamp counter (TSC) for the start time and performs work in a loop, polling the TSC again until the quantum has elapsed. Invariably the amount of elapsed time will exceed the quantum; note that reading the TSC takes about 30 TSC ticks on c89. FTQ adjusts for overrunning a quantum by shortening the following quantum by the excess amount, so that the overall frequency remains accurate.

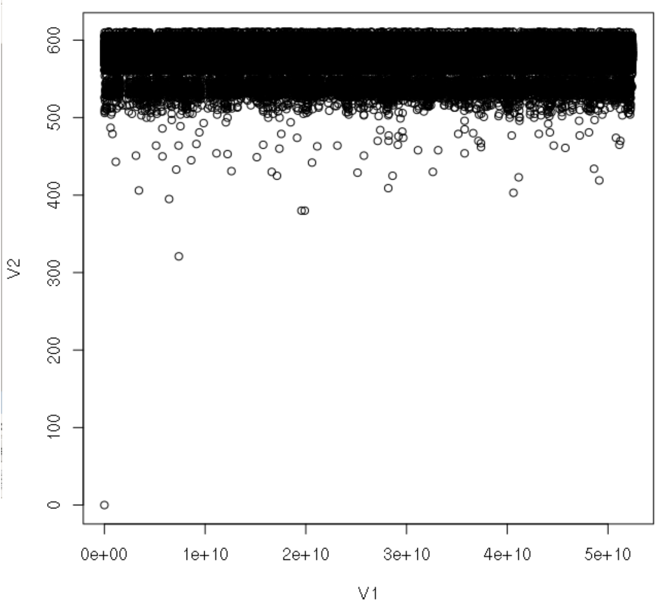

Figure 4.1: FTQ Raw Output, c89, Linux, Core 7

Let's begin with a picture of the raw data from a run on Linux, core 7, in Figure 4.1. The X-axis is the timestamp, starting from 0, and the Y-axis is the work amount for that quantum. Looking at the raw data can be useful, but often it is not particularly easy to spot trends. After applying the FFT, we have a choice of what range of frequencies to look at; this is purely a visualization issue, where certain signals are easier to see if you zoom in.

For example, Figure 4.2 shows up to 3000 Hz, but there is so much noise it is hard to see what is going on.

Figures 4.3 and 4.4 show the same FFT, zoomed in at 500 and 100 Hz, respectively. Looking at the 100 Hz Figure 4.4, we can clearly see spikes at 40 Hz and 80 Hz. Often a signal will repeat at multiples of its frequency; if there is a square-ish signal in the raw data at say X Hz, there will be signals at X * 1, X * 2, ...

There is somewhat of an art to interpreting these FTQ results, perhaps due to my lack of signal processing expertise. From the 100 Hz figure, we can detect a 40 Hz signal, and

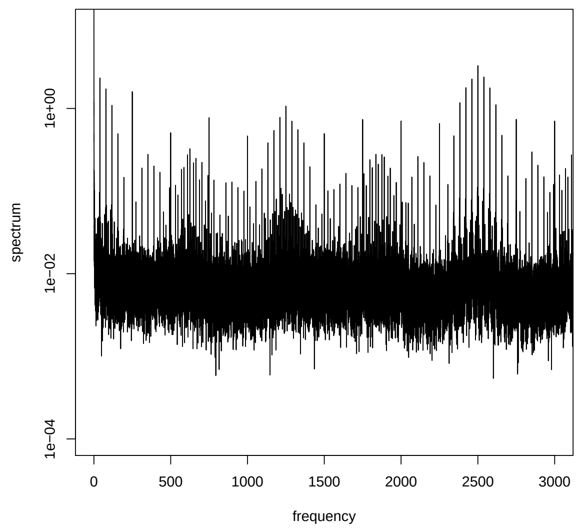

Figure 4.2: FFT @ 3000 Hz, c89, Linux, Core 7

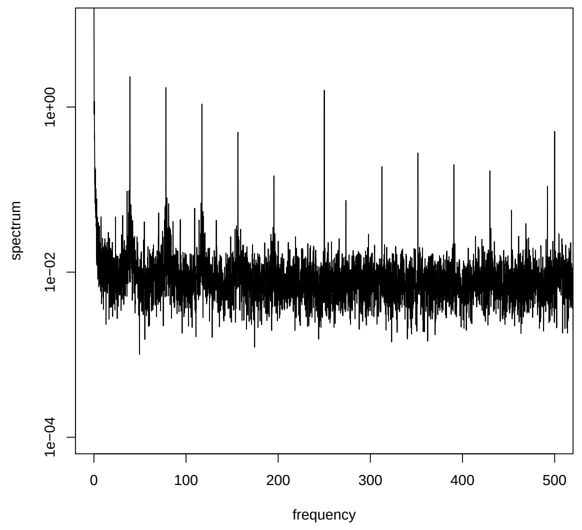

Figure 4.3: FFT @ 500 Hz, c89, Linux, Core 7

when we look at the 500 Hz figure, we see the multiples of 40 Hz up to 200 Hz. Those all appear to be from the same underlying source of interference. By contrast, the spike at 250

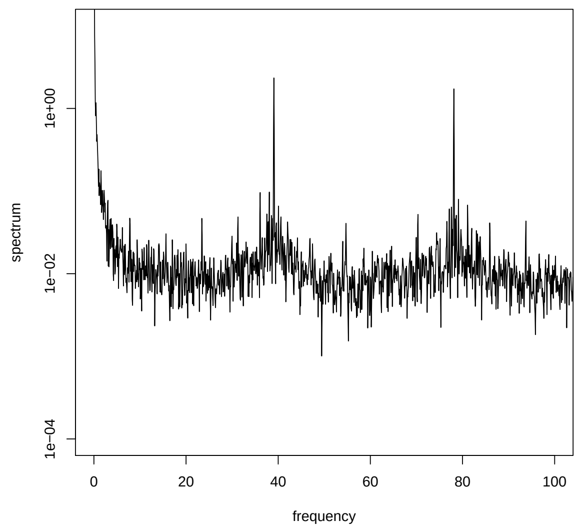

Figure 4.4: FFT @ 100 Hz, c89, Linux, Core 7

Hz seems like a new source of interference, one that has its multiple at 500 Hz. Armed with that knowledge, we can see signals in the original 3000 Hz figure at multiples of 250 Hz.

The next step is to determine what those sources of noise may be. For instance, 250 Hz is the kernel scheduler tick. I do not the source of the 40 Hz signal, though I suspect something with the particular Intel platform, as I discuss below.

Additionally, the overall noise floor can be of interest. From discussions with Ron Minnich, FFTs from the same type of machine will have similar baseline amounts of noise.

Given that Akaros has absolutely nothing running on a core other than a process, FTQ results should show that lack of interference. Figure 4.5 is a representative example of early FTQ results on Akaros, zoomed to 100 Hz. There are huge spikes at multiples of 18 Hz; there are so many that the 3000 Hz figure (not shown) is mostly a black square. Also, note the very small signals at 40 Hz and 80 Hz, similar to the signals on Linux.

The source of the 18 Hz signals is not something Akaros is doing; rather they are from what Akaros does not do well: initialize all devices. Intel machines have a layer of control known as System Management Mode (SMM), often used by the BIOS or other very low-level software for platform management. SMM is triggered by a special interrupt, the SMI, and there are many sources of SMIs in the platform. One of the features of SMM is that the entire processor, meaning all cores, are placed in SMM during an SMI. I originally ran into SMM issues when I noticed that certain benchmarks took longer to run when started from the keyboard, instead of via the serial port. The reason was USB legacy keyboard support; anything related to legacy hardware is usually implemented with SMM.

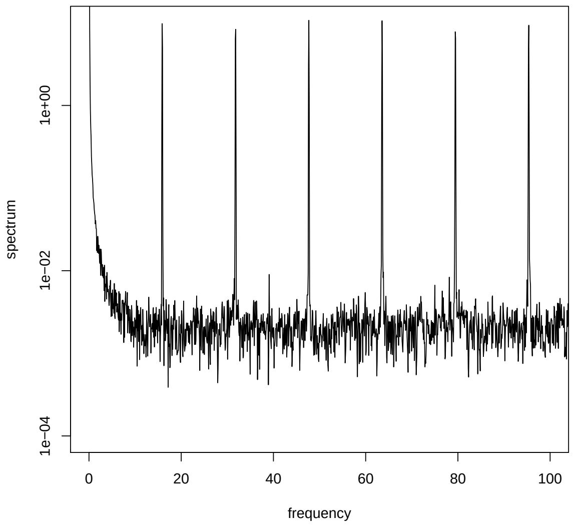

Figure 4.5: FFT @ 100 Hz, c89, Old Akaros, Core 7

Although initializing hardware properly is not really part of the research into whether or not the MCP abstraction provides isolation, it is necessary to get a decent measurement. Diagnosing the interference due to SMM with no other evidence is tricky; I was lucky to figure out the keyboard interference. FTQ's FFTs provide some much needed assistance by providing the frequency of interference, in this case around 18Hz. 18.2 Hz is the frequency at which the 8254 Programmable Interval Timer (PIT) rolls over (1.19 MHz / 65536). I looked into the initialization of various timer hardware, such as the HPET and RTC, but it turns out the issue was a platform timer in the Platform Controller Hub (PCH), which is a chip including the old southbridge functionality. Once I disabled that timer, Akaros's FFTs for core 7 on c89 are in Figures 4.6 (100 Hz), 4.7 (500 Hz), and 4.8 (3000 Hz).

Note in Figure 4.6 (100 Hz) that the 40 Hz and 80 Hz signals are still there, though the signals are weaker. It may be those are SMM related, since the operating system is not causing them, and since Linux has them, then they may be an artifact of the platform. Another alternative is that the benchmark itself causes the signal. The act of writing the results into the pinned data buffer can trigger various machine-dependent signals. For instance, cache and TLB misses may happen periodically, and at values dependent on the size of the particular caches. Similarly, both Akaros and Linux have a signal around 1250 Hz; Akaros's is more apparent, and Linux's is an increase in the overall noise.

Despite the various platform idiosyncrasies, Akaros's MCPs have less interference overall than Linux on core 7, when viewed at 100 Hz, 500 Hz, and 3000 Hz. The difference is subjectively easier to see as one zooms out. All of these tests were on Core 7, which shares a

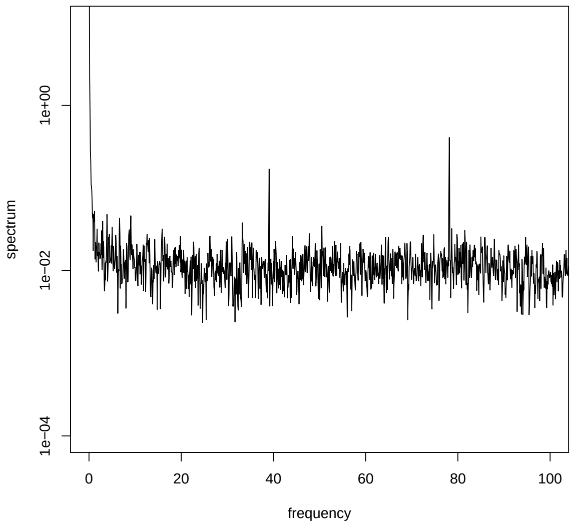

Figure 4.6: FFT @ 100 Hz, c89, Akaros, Core 7

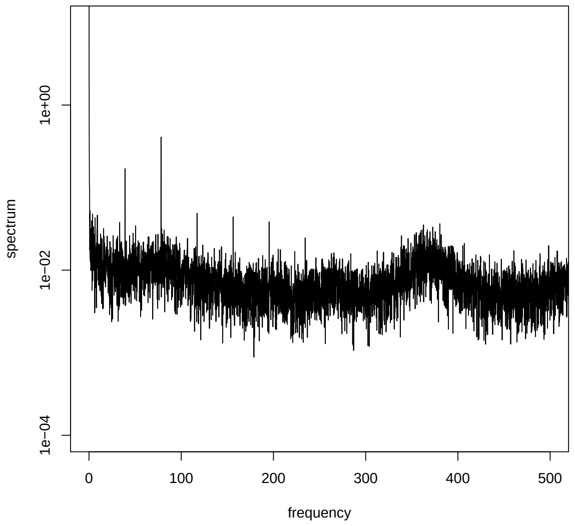

Figure 4.7: FFT @ 500 Hz, c89, Akaros, Core 7

socket and LLC with Core 0. For both Linux and Akaros, Core 0 is busy with various tasks; on Akaros, Core 0 runs a kernel timer tick, receives various interrupts, and runs SCPs. To

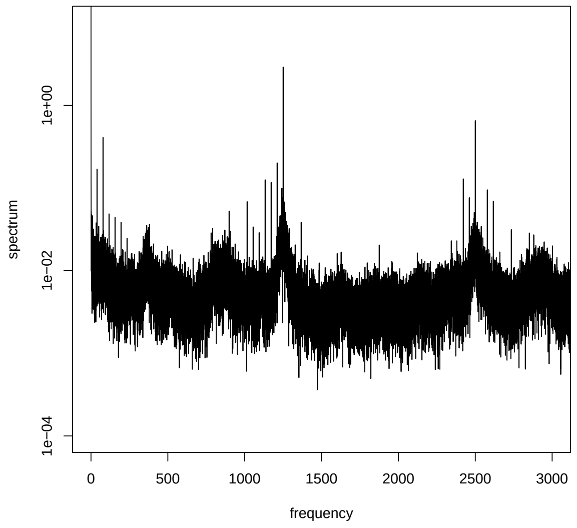

Figure 4.8: FFT @ 3000 Hz, c89, Akaros, Core 7

gain more insights into the nature of noise and how FTQ works, I ran the same tests on Core 31, which is on another socket. For ease of comparison, I show both OSs at a given level of zooming.

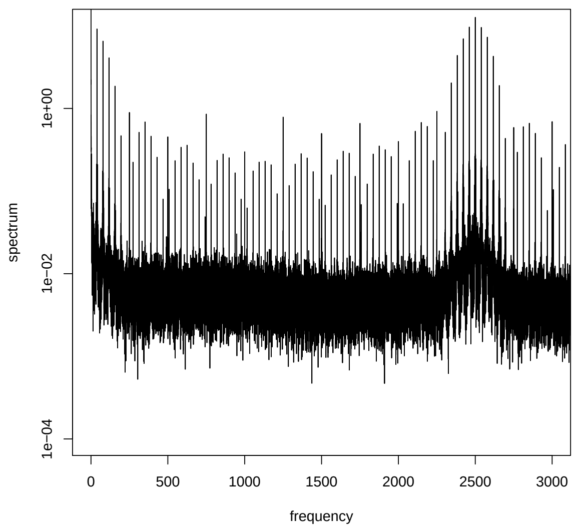

Figure 4.9: FFT @ 3000 Hz, c89, Linux, Core 31

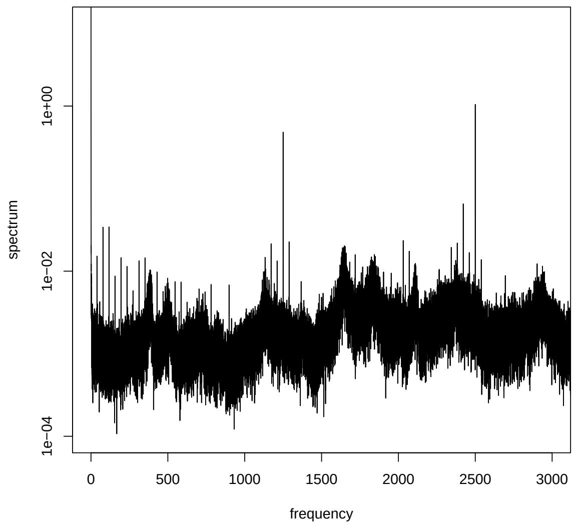

Figure 4.10: FFT @ 3000 Hz, c89, Akaros, Core 31

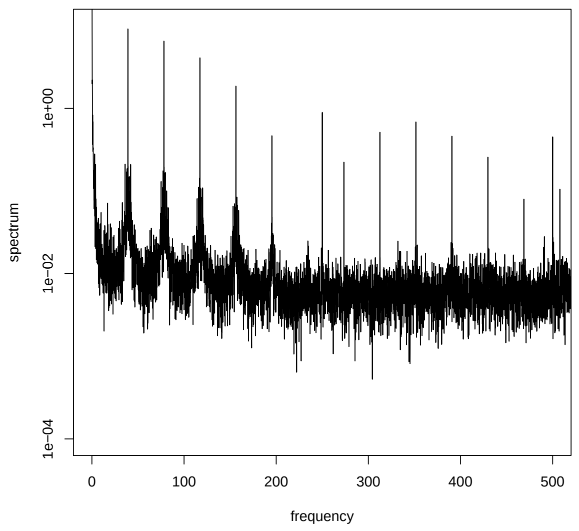

Figure 4.11: FFT @ 500 Hz, c89, Linux, Core 31

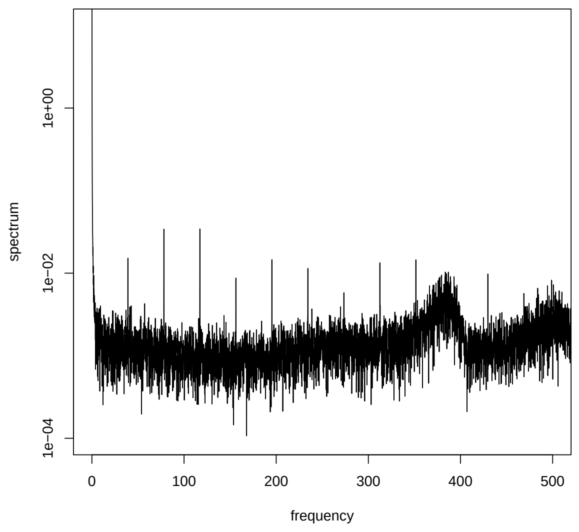

Figure 4.12: FFT @ 500 Hz, c89, Akaros, Core 31

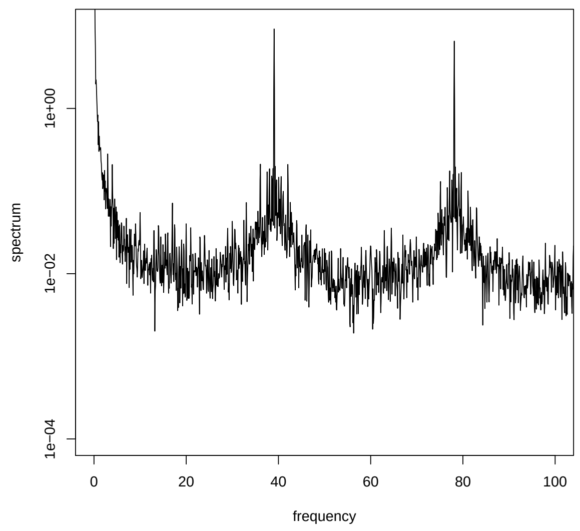

Figure 4.13: FFT @ 100 Hz, c89, Linux, Core 31

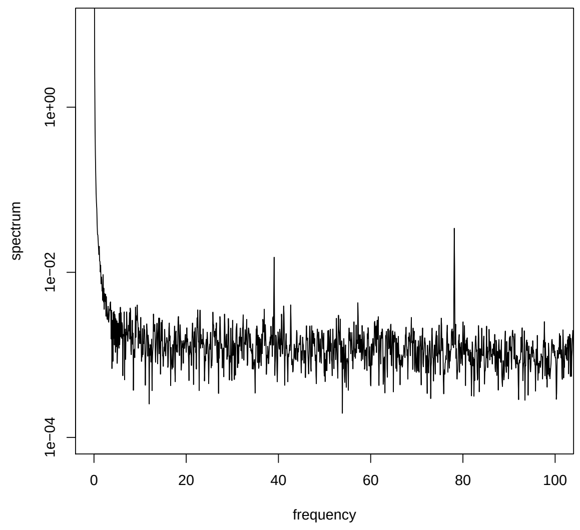

Figure 4.14: FFT @ 100 Hz, c89, Akaros, Core 31

Two traits stand out: Akaros has less signals overall, as with the previous results on Core 7, and Akaros's noise floor is significantly lower. Linux's graphs are very similar between

Core 7 and 31; moving to another socket did not achieve much. Arguably, the Core 31 at 3000 Hz is a little better. Akaros's Core 31 graphs are mostly just shifted down, compared to Core 7. I do not know the reasons behind this behavior. One possibility is some form of cross-chip interference, such as sharing the L3 cache. On Akaros, the other cores sharing the socket with Core 31 are all halted. On Linux, those cores respond to the same sorts of timer ticks and background daemons that run on all Linux cores. Perhaps that added interference is enough to increase the overall amount of noise.

It is worth looking at the raw data for Akaros, comparing Core 7 to Core 31 in Figures 4.15 and 4.16, respectively. Both have a "thick band" of results, but Core 7 has a little hitch around sample $7.5 * 10^{10}$. Step-ups like this can be indicative of some form of warm-up; based on how long it takes to kick in, it may be a hardware feature like Turbomode, though that particular setting is disabled in the BIOS.

Although we are looking at the raw data, compare Akaros's Core 7 (Figure 4.15 to Linux's Core 7 (Figure 4.1). We cannot see any signals or patterns of interference for either OS, but we can see that Akaros's results are much tighter and more predictable. This matches the intuition from looking at the various FFTs, where Akaros has a lower noise floor, fewer spikes, and a less "bristly" FFT. The FFT is powerful, but sometimes the raw data is still useful too.

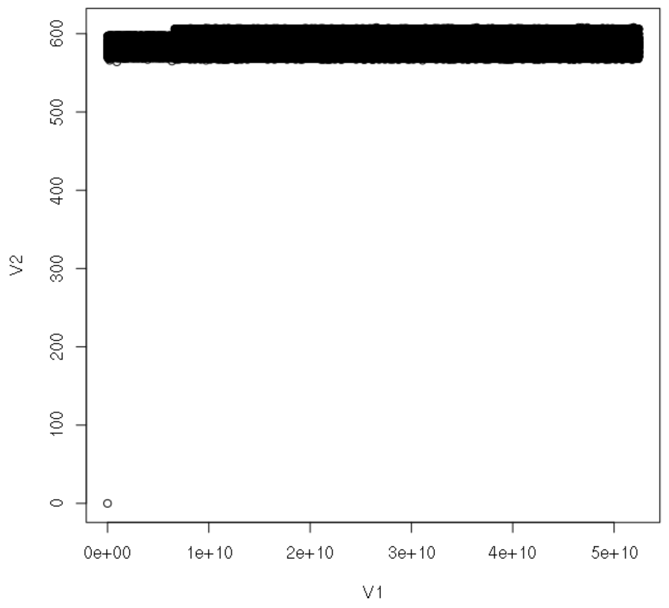

Figure 4.15: FTQ Raw Data, c89, Akaros, Core 7

Given all the platform-specific behavior exposed by FTQ, whether it is SMM or possible microarchitectural traits, I compared Akaros and Linux on another machine. The next machine, referred to as "hossin", is an Intel Nehalem Core i7-920, 2.67 GHz. It has a single

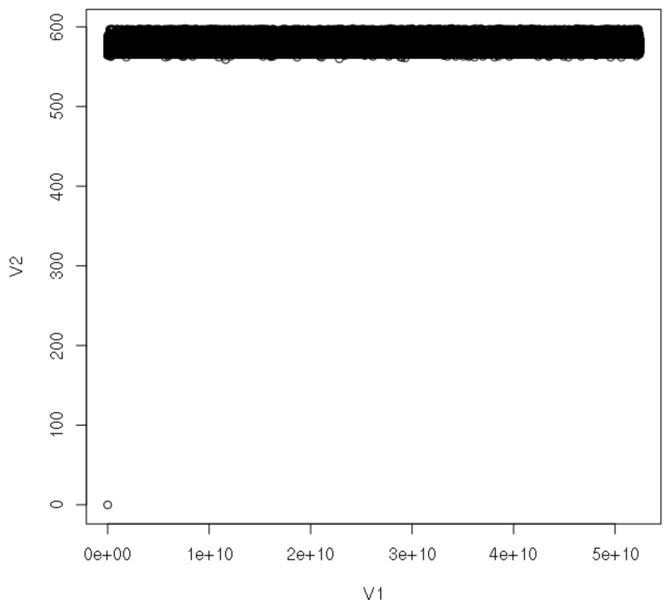

Figure 4.16: FTQ Raw Data, c89, Akaros, Core 31

socket with 4 cores, 8 hyperthreads, 8 MB L3 cache, and 3 GB RAM. The Linux kernel was 3.11.1-gentoo.

Running FTQ on Akaros on this machine required disabling all USB legacy support in the kernel, which is a normal part of initializing the USB subsystem. Akaros uses the USB legacy mass storage to boot, since we tend to boot desktops with a USB hard drive. Since the BIOS needs legacy USB support to boot us, we need to wait until the kernel runs to disable the legacy USB stack. Failing to do this results in a very strong 1 Hz signal (not shown); the constant SMIs are probably polling the USB device.

As before, I show the results for both Linux and Akaros, zooming from 3000 Hz down to 100 Hz.

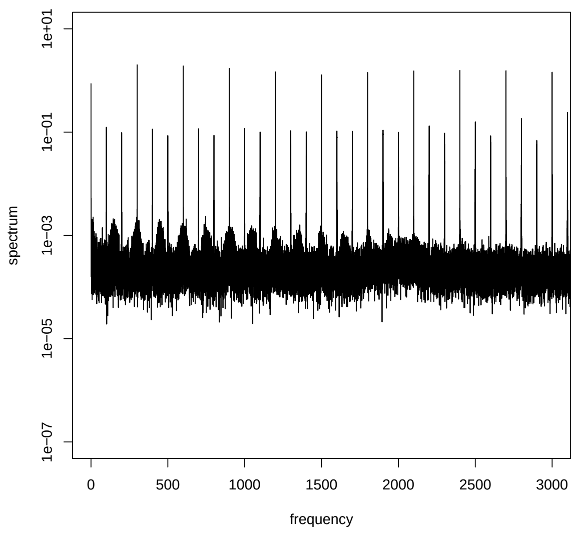

Figure 4.17: FFT @ 3000 Hz, hossin, Linux, Core 7

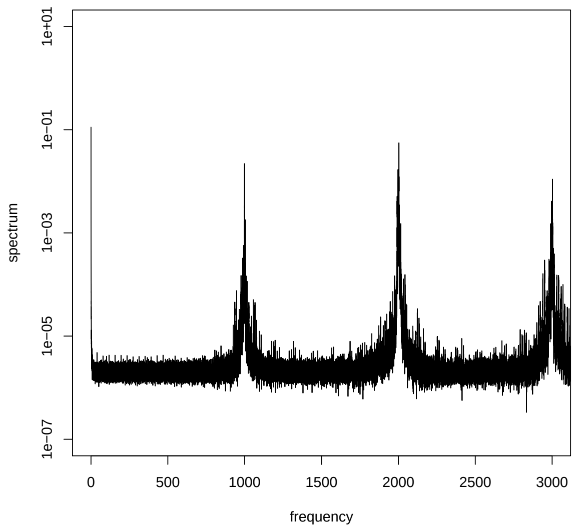

Figure 4.18: FFT @ 3000 Hz, hossin, Akaros, Core 7

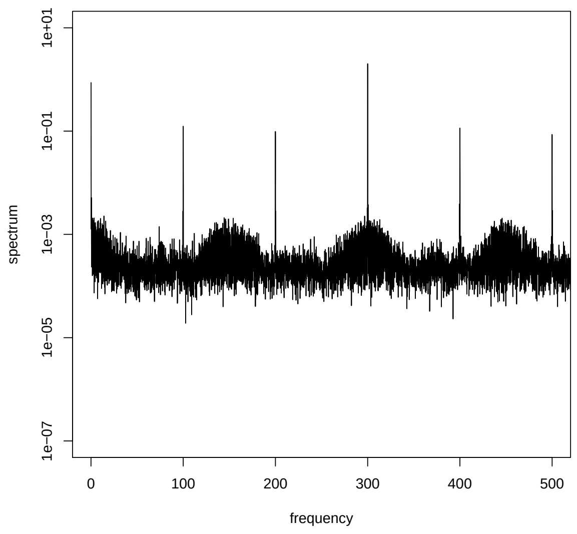

Figure 4.19: FFT @ 500 Hz, hossin, Linux, Core 7

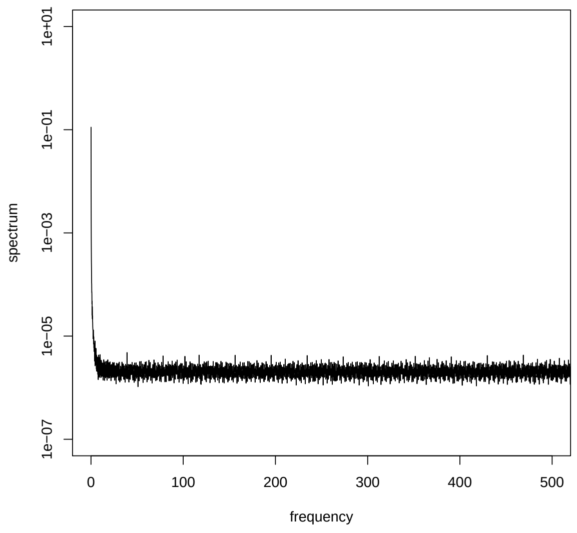

Figure 4.20: FFT @ 500 Hz, hossin, Akaros, Core 7

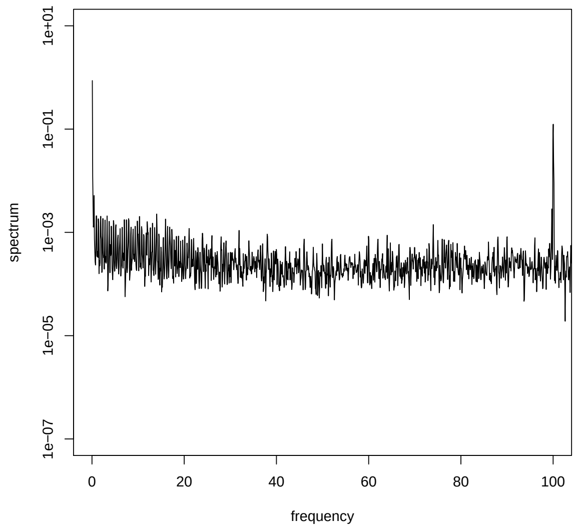

Figure 4.21: FFT @ 100 Hz, hossin, Linux, Core 7

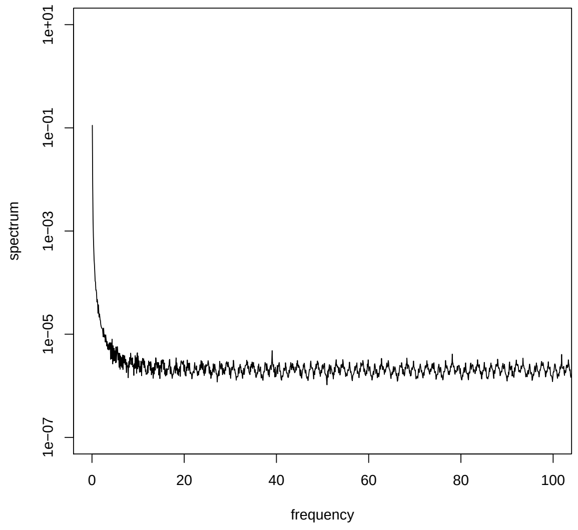

Figure 4.22: FFT @ 100 Hz, hossin, Akaros, Core 7

The first trait to note is that the noise floor on hossin is different than on c89 for both Linux and Akaros. For instance, Linux's noise on hossin was around 10^{-3}, compared to 10^{-2}

on c89. Linux has two sources of periodic interference: 100 Hz tick and 300 Hz tick. Both sources repeat at multiples of the base frequency; the clue that the sources are different is the magnitude of the 300 Hz signal. This version of the Linux kernel was configured with a 300 Hz scheduler tick. I do not know what the 100 Hz signal is for, probably another timer.

Akaros has a large signal at 1000 Hz, but otherwise both its noise floor and the amount of signal at that level (the spectrum's thickness) are very low. Akaros is below 10^{-5}, compared to Linux's below 10^{-3}. However, Akaros does have a very large 1000 Hz signal, possibly due to some hardware I did not initialize properly. The raw data in Figures 4.23 and 4.24 is interesting in this regard. For both Akaros and Linux, the usual work amount for a quantum is about 703. The deviations are around 680-690 for Akaros and worse than that for Linux. Akaros only has 12 deviations, which are quanta with any interference, compared to over 16,000 for Linux out of over 50 billion quanta. There is so little interference in Akaros that you can see it in the raw data. The 1000 Hz signal on Akaros may be very large, but that does not mean the interference is large — only that the signal itself is very clear. The lesson is that the magnitude of the signals in the FFT are not indicative of the amount of interference, only the *periodic nature* of the interference. More signals and a higher noise floor are both indicative of higher interference.

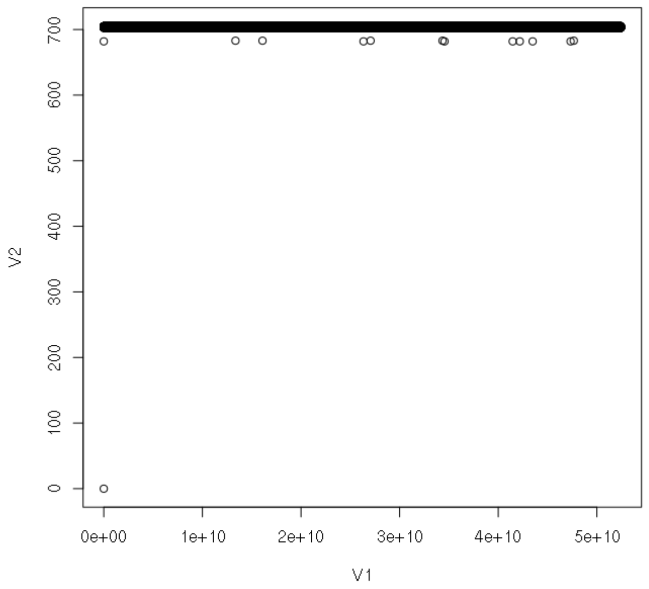

Figure 4.23: FTQ Raw Data, hossin, Akaros, Core 7

FTQ and Performance Isolation Summary

There are many sources of interference in a computer, and the operating system can only control some of them. An OS that allocates unvirtualized, physical cores to an application is

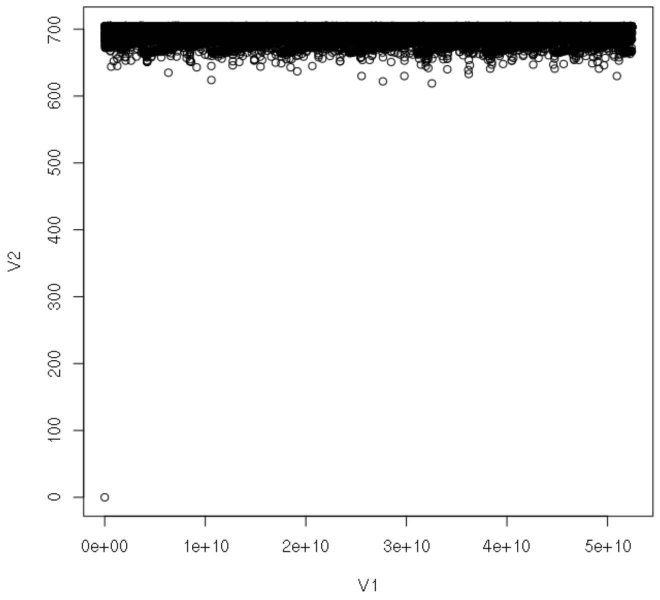

Figure 4.24: FTQ Raw Data, hossin, Linux, Core 7

necessary, but not sufficient, for performance isolation. Users need to analyze their particular platforms and deal with issues like System Management Mode, Turbo-mode, and other hardware characteristics. Although many SMM issues on Akaros are a result of uninitialized hardware, some SMM events are unavoidable — SMIs can be critical for the platform's health, such as alarms to prevent overheating. Additionally, a particular chip might have other sources of variation per quantum. For instance, the steady 703 count per quantum on hossin was very different than c89. Even with Akaros on Core 31 on c89, the counts varied in a thick band (Figure 4.16), which is a trait that OSs are unlikely to affect.

FTQ is a useful benchmark to compare the performance isolation of different OSs and various settings on a given machine. Both the raw data and the FFT are useful in different ways. The raw data exposes certain temporal behavior, such as a sudden step-up in performance, and the raw view provides insights into the bounds and patterns of the interference. The FFT extracts patterns in the raw data that humans likely cannot see, which is useful for both debugging sources of interference and comparing results from separate experiments.

In this section, I have shown that Akaros's MCP cores have an order of magnitude less noise than Linux. This improvement can be seen by comparing the noise floors from the FFTs of the FTQ output. Additionally, Akaros has fewer periodic signals — these are the spikes throughout the FFT output. Together, these two results show that MCP cores have less interference and better CPU isolation than a traditional OS's cores.

In the long term, Akaros plans to provide the same provisioning/allocation scheme for other resources as it provides to CPUs, for (future) hardware that supports isolating those resources.

Current x86 hardware allows the OS to control the CPU and RAM directly and other resources indirectly. As a proxy for controlling caches, the OS can provision and allocate all the cores that share a particular cache to the same application. Although the OS cannot cleanly prevent the cores from interfering with each other's cache accesses, at least the application only deals with interference from itself and not from other programs or the OS itself. There are various other techniques to indirectly affect the use of certain resources, such as page coloring[49]. Page coloring is based on the observation that certain physical pages of memory will be mapped to certain parts of a cache. The details depend on the associativity and size of the caches. Page coloring can be used to restrict which parts of a cache a process can access by only granting certain physical pages to the process, such that the physical pages map to only the "allocated" banks of the cache.

Akaros has had rudimentary support for page coloring for a few years, though no one uses it. There are many issues with page coloring, one of which is that the partitioning applies to all levels of caching. Though the number of "colors" differs at each level of caching, based on the cache's traits, isolating a process to use one color in the L3 can restrict the process to using that same color in the L1 and L2, even if those caches are not shared with other cores. It is also much more difficult to color the kernel's pages, compared to the user's.

Regardless of cache and other resource isolation, CPU isolation is a critical component of predictable performance, which is especially important for parallel processes. Akaros's MCPs have an order of magnitude less noise than on Linux, as well as fewer periodic signals, resulting in better CPU isolation. Ultimately, the MCP provides a collection of isolated physical cores to a process, even though the abstraction involves "virtual" cores. It is up to the application to schedule its threads and utilize its resources to the best of its ability, free from OS noise and CPU interference from other processes.

Chapter 5

Event Delivery

Processes need to communicate, both with the kernel and other processes. IPC and signal delivery are common to all operating systems, and Akaros has a variety of these mechanisms. In this chapter, I will discuss how Akaros supports event delivery for MCPs. I start with the basic overview of events, followed by particular event delivery mechanisms such as the *UCQ*, and conclude with a description of the overall event subsystem and the various techniques for ensuring message delivery.

The MCP has various requirements for event delivery:

- The user controls delivery parameters. MCPs need to control all aspects of their environment. For a certain event, the process wants to control whether or not it receives an interrupt and which vcore receives the interrupt (notification). Likewise, a process may require an event with a payload, or perhaps merely setting a bit would suffice. The simplest approach is to ask for an IPI and only set a bit for the message: equivalent to how hardware interrupts work. More complicated approaches require more programming effort from the process.

- MCPs need options for reliable delivery. The primary example of an event is the "syscall complete" message. Many second-level schedulers (2LSs) cannot afford to miss this message; otherwise the process may go to sleep, not knowing a thread can unblock. Missing messages when leaving vcore context is equivalent to Saltzer's "wakeup waiter" problem[94]. One distinction with MCP event delivery is that the "waiter" is user code, and the kernel and user must work together to solve the "wakeup waiter" problem. Not all events require reliable delivery — this depends on the specific event. Furthermore, some events should only be delivered once. For example, if the "syscall complete" message can be delivered more than once, the 2LS needs added complexity to prevent restarting the same uthread twice or dealing with messages for old syscall structs.

- Delivery should strive to be fast and scalable, yet safe for the kernel. Many fast IPC systems, such as the Xen ring buffers[7], use shared memory. In these systems, the kernel is synchronizing with a potentially malicious user and must be careful to protect itself. All serious systems, including Xen, protect the kernel in this manner; Akaros must be no different.

Akaros's event delivery provides the option for either a bit or a payload element, consisting of a small number of bytes. I will start from the bottom-up, describing how we can deliver payloads from the kernel to the user.

5.1 Concurrent Queues for Payloads

Bounded Concurrent Queues

Originally, I built a data structure for event delivery called *Bounded Concurrent Queues.* They were multi-producer/multi-consumer, fixed sized, ring buffers of elements, where the producer never trusts the consumer. They were inspired and based on techniques used by Xen's ring buffers[7], which were mapped into shared memory. They turned out to be a little lacking, so Akaros does not use them currently, though I may use them in the future.

Xen's buffers were bidirectional, power-of-two sized buffers, with two index variables tracking each end of the ring. A sender (for a given direction) increments a "private index", reserving a slot in the buffer, then copies the payload into the slot, and finally updates a shared variable that signals the receiver that the payload can be extracted. The power-of-two size buffer means that the index variables can simply be incremented, using masking as a very fast modulus operator. BCQs differed in three ways:

- Multi-producer/multi-consumer: the BCQ code had `enqueue()` macros, supporting multiple producers. Senders would compete on the private index variable with compare-and-swap. Atomic increment might not be sufficient, since it needs to handle cases where the queue fills concurrently.
- All data structure state was stored in the shared memory region. Xen had a shared region and two private bookkeeping regions, one for each end of the communication that were inaccessible to the other. BCQs put all the state in the shared ring, and the kernel simply did not trust the values. There were no pointers, and array indexes were limited to the buffer size by masking against the size - 1. The purpose of keeping the metadata in the structure is so that userspace can allocate and initialize the structures without the kernel's involvement.
- BCQs were unidirectional, which is a relatively minor point.

BCQs are relatively simple, where most of the complexity is dealing with failures by the producer:

```
struct bcq_header {
    uint32_t prod_idx;      /* next to be produced in */
    uint32_t cons_pub_idx;  /* last completely consumed */
    uint32_t cons_pvt_idx;  /* last a consumer has dibs on */
};
```

```

#define bcq_init(_bcq, _ele_type, _num_elems)
  memset((_bcq), 0, sizeof( _ele_type ) * (_num_elems))

#define BCQ_FREE_SLOTS(_p, _cp, _ne) ((_ne) - ((_p) - (_cp)))
#define BCQ_EMPTY(_p, _cp, _ne) ((_ne) == BCQ_FREE_SLOTS(_p, _cp, _ne))
#define BCQ_NO_WORK(_p, _cpv) ((_p) == (_cpv))
#define BCQ_FULL(_p, _cp, _ne) (0 == BCQ_FREE_SLOTS(_p, _cp, _ne))

#define bcq_enqueue(_bcq, _elem, _num_elems, _num_fail)
({
  uint32_t __prod, __new_prod, __cons_pub, __failctr = 0;
  int __retval = 0;
  do {
    cmb();
    if (((_num_fail)) && (__failctr++ >= (_num_fail))) {
      __retval = -EFAIL;
      break;
    }
    __prod = (_bcq)->hdr.prod_idx;
    __cons_pub = (_bcq)->hdr.cons_pub_idx;
    if (BCQ_FULL(__prod, __cons_pub, (_num_elems))) {
      __retval = -EBUSY;
      break;
    }
    __new_prod = __prod + 1;
  } while (!atomic_cas_u32(&(_bcq)->hdr.prod_idx, __prod, __new_prod));
  if (!__retval) {
    (_bcq)->wraps[__prod & ((_num_elems)-1)].elem = *(_elem);
    (_bcq)->wraps[__prod & ((_num_elems)-1)].rdy_for_cons = TRUE;
  }
  __retval;
})

#define bcq_dequeue(_bcq, _elem, _num_elems)
({
  uint32_t __prod, __cons_pvt, __new_cons_pvt, __cons_pub;
  int __retval = 0;
  do {
    cmb();
    __prod = (_bcq)->hdr.prod_idx;
    __cons_pvt = (_bcq)->hdr.cons_pvt_idx;
    if (BCQ_NO_WORK(__prod, __cons_pvt)) {
      __retval = -EBUSY;
      break;
    }
```

```
        __new_cons_pvt = (__cons_pvt + 1);
    } while (!atomic_cas_u32(&(_bcq)->hdr.cons_pvt_idx, __cons_pvt,
                             __new_cons_pvt));
    if (!__retval) {
        while (!(_bcq)->wraps[__cons_pvt & ((_num_elems)-1)].rdy_for_cons)
            cpu_relax();
        *(_elem) = (_bcq)->wraps[__cons_pvt & ((_num_elems)-1)].elem;
        (_bcq)->wraps[__cons_pvt & ((_num_elems)-1)].rdy_for_cons = FALSE;
        while ((_bcq)->hdr.cons_pub_idx != __cons_pvt)
            cpu_relax();
        (_bcq)->hdr.cons_pub_idx = __cons_pvt + 1;
    }
    __retval;
})

#define bcq_empty(_bcq)
  BCQ_NO_WORK((_bcq)->hdr.prod_idx, (_bcq)->hdr.cons_pvt_idx)
```

BCQs are a useful way to move payloads to userspace, with basic initialization, enqueue, and dequeue methods. The main problem with BCQs was that some use cases needed more from them. Enqueuing messages in BCQs can fail, due to either a misbehaving process, an unlucky circumstance, or most commonly because the queue was full. Event queues, described later and which used BCQs as a building block, would handle the failure as an "overflow" of events, and signal the process with a bit.

In the early days of using BCQs to send syscall completion events, deciding how a 2LS could deal with a missed syscall message was rather tricky. When the kernel completes a syscall, it sends a message that a particular syscall was done. Userspace needs to know exactly which one was done, under normal circumstances. With BCQs, the 2LS needs to know how to handle overflow. Handling overflow requires tracking every outstanding syscall so that it can poll to see which call completed. A global list or hash table (hashed on the syscall struct address) would work, though there is a "scalability tax" paid to manage these for every blocking syscall, since the 2LS must prepare for the possibility of any syscall completion message being lost. Per-vcore lists are a tempting solution, but when a vcore is preempted, the list needs to be dealt with.

An additional complication is that the 2LS cannot blindly run the completion handler on a syscall twice, in essence a "memory-reuse" or "stale-pointer" problem. Say completion messages were sent for syscalls A and B. A's message was delivered, and B's failed and triggered an overflow. The overflow handler polls *all* syscalls for completion, and notices A is complete. However, there is also a message delivered for A's completion, possibly handled by another vcore concurrently. Any scheme that attempts to handle syscall overflow messages must account for this possibility.

On a similar note, dealing with missing messages that inform the process of a vcore being preempted can be difficult. In that case, the process can perform an $O(n)$ scan of the vcoremap, but it also needs to know which vcores are *supposed* to be online. It is possible and correct for applications to have holes in their vcoremaps, such the list of online vcores are not consecutively numbered.

In either case, it is not impossible to deal with overflowing a BCQ and losing a message; it is just difficult. Ultimately, complicated systems could benefit from not needing to handle overflow, which led me to develop an *unbounded* variant of the concurrent queue.

Unbounded Concurrent Queues

Unbounded Concurrent Queues (UCQs) are a tool to send messages, with payloads, from the kernel to a process through shared, user-writable memory. UCQs support multiple concurrent producers and consumers, and the producers never trust the consumer. The depth of the queue is "unlimited", barring running out of user memory. "Growable" or "Expandable" would be a better name than "Unbounded"; I may change it one day. UCQs should be used in closed-loop or low-rate scenarios that require a payload or that where missing a message is unwieldy. UCQs takes advantage of the fact that the kernel is the producer, which provides certain opportunities since the consumer trusts the producer. The problems in evolving the BCQs to UCQs are running out of slots for events, being careful about the kernel using user-provided pointers, and synchronizing within the kernel (e.g. a spinlock for a linked list).

The basic idea for UCQs is to use linked, mmapped pages of arrays of event messages. By comparison, the BCQ is a circular array of event messages. As the kernel over-produces messages, it mmaps more pages into the user's address space and links them together, via a header at the beginning of the page. Userspace munmaps when it consumes all the entries in a page. To avoid excessive calls to `mmap()` and `munmap()`, we double-buffer with two pages: one current and one spare. We only need to mmap new pages when the kernel gets far ahead of the user.

The main benefit of UCQs is that they can handle spikes of messages and thus remove the need to handle overflow. This benefit comes from the kernel's ability to mmap new pages for the user on-demand, due to the user's implicit trust of the kernel. The implicit mmap solves the problem of running out of room in a fixed-size buffer by adding more buffers. The worst thing userspace can do is leak its own memory, or cause an out-of-memory condition, which the OS must deal with anyway. To be clear, any problem caused by a malicious user by triggering implicit mmaps can also be caused by normal, explicit mmaps.

Of course, the main benefit of the UCQ is also tied to its main drawback: a process can die if it the kernel runs out of memory while trying to send it messages. If the kernel sends messages faster than the process can handle them for any reason, including being swapped out, eventually the process exceeds its allowable memory allocation. UCQs are powerful, but they need to be used carefully, such that the process is not requesting an *unbounded* amount of messages at one time. Syscall completions are not unbounded: they are limited by the number of syscalls and usually uthreads, any of which take up more memory than the actual payload.

Using user-pointers is not conceptually a problem, so long as the kernel reads the pointer in and verifies the address is in userspace, and that it can handle kernel page faults on user addresses. Akaros actually cannot handle the later yet, though that is a coding problem, not a design problem. Regardless of the safety of using pointers or not, it is safer and faster to use array indices, similar to the BCQ. I only use pointers that could be controlled by userspace in limited situations, and always copy-in and verify the pointer before using it.

Swapping pages of buffers requires locking or some form of synchronization between kernel producers, but we cannot put the lock in the UCQ structure, since userspace could corrupt it. One option would be to use private data structures, as Xen's ring buffers do, and put the lock in the kernel-private structure. As with BCQs, I prefer to keep UCQ metadata in the shared memory structure so that userspace can allocate and initialize the UCQ on its own. Not only is this more flexible for the user, but it avoids the slight problem of looking up the private kernel part based on information from userspace.

Of course, any kernel locking strategy needs to use memory that is not user-writable. Instead of using private, per-UCQ memory, I lock at the process level. For scalability reasons, instead of grabbing the process lock, I grab a *hashlock*, which I developed specifically for UCQs. Hashlocks are a hash table of regular spinlocks; hashed in this case on the pointer of the UCQ. Typically, when coding in C, lock structures are embedded in larger structures, such as a file lock being in an inode struct. Hashlocks provide a means of synchronizing on objects in memory not inside the structures they protect. Additionally, there are not as many locks as objects, which can be both good and bad, where the tradeoff is between false sharing and memory usage. For UCQs, the benefit is that the memory is external from the user-writable ucq structs. In a sense, the hashlocks serve as the private structure for the UCQ. The differences are that there is not a one-to-one mapping between locks and UCQs and the kernel does not need to know about the UCQs *a priori*. The hashlocks are private for the kernel, but are not per UCQ.

How UCQs Work: Common Code

First, we should look at the structures in shared memory, which are shared between the user and the kernel. The overall structure is the UCQ, which points to a chain of pages.

```
struct ucq {
    atomic_t                    prod_idx;       /* both pg and slot nr */
    atomic_t                    cons_idx;       /* cons pg and slot nr */
    atomic_t                    spare_pg;       /* mmaped, unused page */
    atomic_t                    nr_extra_pgs;   /* nr pages mmaped */
    bool                        prod_overflow;  /* prevents wraparound */
    bool                        ucq_ready;
    /* Storage for Userspace lock for modifying the UCQ */
    uint64_t                    u_lock[2 * ARCH_CL_SIZE / 8];
};

struct ucq_page_header {
    uintptr_t                   cons_next_pg;   /* next page to consume */
    atomic_t                    nr_cons;        /* consumed so far */
};

struct msg_container {
    struct event_msg            ev_msg;
    bool                        ready;          /* kernel has written */
};

struct ucq_page {
    struct ucq_page_header      header;
    struct msg_container        msgs[];
};
```

The `struct event_msg` is the payload, coupled with a "ready" flag so the consumer knows when the message is fully copied into the storage. Each page in the linked list of pages is a `struct ucq_page`, each of which has a header that points to the next page and contains an array of messages. The UCQ has producer and consumer indexes, pointing to the next free "slot"; an index encodes both the virtual page address and the container index within that page. When producers or consumers get a slot, they need to convert it to an actual message pointer. Both the user and the kernel use the following two helper functions:

```
static bool slot_is_good(uintptr_t slot)
{
    uintptr_t counter = PGOFF(slot);
    uintptr_t pg_addr = PTE_ADDR(slot);
    return ((counter < NR_MSG_PER_PAGE) && pg_addr) ? TRUE : FALSE;
}

static inline struct msg_container *slot2msg(uintptr_t slot)
{
    return &((struct ucq_page*)PTE_ADDR(slot))->msgs[PGOFF(slot)];
}
```

Armed with the basics, next we will see how the kernel sends messages to a UCQ.

How UCQs Work: Producing

To actually send a message, the kernel calls `send_ucq_msg()`. I describe the various parts below.

```
void send_ucq_msg(struct ucq *ucq, struct proc *p, struct event_msg *msg)
{
  uintptr_t my_slot = 0;
  struct ucq_page *new_page, *old_page;
  struct msg_container *my_msg;

  assert(is_user_rwaddr(ucq, sizeof(struct ucq)));
  if (ucq->prod_overflow)
    goto grab_lock;
  my_slot = (uintptr_t)atomic_fetch_and_add(&ucq->prod_idx, 1);
  if (slot_is_good(my_slot))
    goto have_slot;
  ucq->prod_overflow = TRUE;
  if (PGOFF(my_slot) > 3000)
    warn("Abnormally high counter, there's probably something wrong!");
grab_lock:
  hash_lock_irqsave(p->ucq_hashlock, (long)ucq);
  my_slot = (uintptr_t)atomic_fetch_and_add(&ucq->prod_idx, 1);
  if (slot_is_good(my_slot))
    goto unlock_lock;
  old_page = (struct ucq_page*)PTE_ADDR(my_slot);
  if (!is_user_rwaddr(old_page, PGSIZE) || !old_page)
    goto error_addr_unlock;
  new_page = (struct ucq_page*)atomic_swap(&ucq->spare_pg, 0);
  if (!new_page) {
    new_page = (struct ucq_page*)do_mmap(p, 0, PGSIZE,
                                         PROT_READ | PROT_WRITE,
                                         MAP_ANON | MAP_POPULATE, 0, 0);
    assert(new_page);
  } else {
    if (!is_user_rwaddr(new_page, PGSIZE) || PGOFF(new_page))
      goto error_addr_unlock;
  }
  new_page->header.cons_next_pg = 0;
  new_page->header.nr_cons = 0;
  old_page->header.cons_next_pg = (uintptr_t)new_page;
  my_slot = (uintptr_t)new_page;
  atomic_set(&ucq->prod_idx, my_slot + 1);
unlock_lock:
  ucq->prod_overflow = FALSE;
  hash_unlock_irqsave(p->ucq_hashlock, (long)ucq);
have_slot:
  my_msg = slot2msg(my_slot);
  if (!is_user_rwaddr(my_msg, sizeof(struct msg_container)))
    goto error_addr;
  my_msg->ev_msg = *msg;
  wmb(); /* order ready write after *msg */
  my_msg->ready = TRUE;
  return;
error_addr_unlock:
  ucq->prod_overflow = FALSE;
  hash_unlock_irqsave(p->ucq_hashlock, (long)ucq);
error_addr:
  warn("Invalid user address, not sending a message");
}
```

Producers atomically compete with fetch-and-add calls for slots in `prod_idx` (line 10), which encodes both the page and the message number within the page. The fetch-and-add ensures that every producer gets a unique slot. In the common case, if the slot is good, we just jump to `have_slot` (line 42) and write our message in the `ev_msg`. The kernel barriers and signals the `ready` flag, is in case the consumer is trying to concurrently read the message while a write is in progress.

The slot is "good" when it points to a valid page and a slot within the range of the message array. When the slot is bad, things get interesting. To protect the linked-list invariants and prevent various madness with concurrent updates, the kernel grabs a hashlock, external to the UCQ. Whichever kernel thread grabs the lock first will be responsible for getting a new page and resetting the producer's index, which tracks which slot is next as well as the new page. All future lockers can tell if that work has been completed by examining the counter and trying to get a slot. If they get a good one, everything is okay and it is just like they got a good slot in the first place.

One minor problem is that with sufficient producers and a delayed fixer, we could overflow the `prod_idx`. Around 3900 concurrent producers could overflow the `prod_idx` into a new page on the 0th slot. To avoid overflow, the first producer to detect the slot/index is bad will set an overflow flag (line 13). All future producers will check this flag before attempting to get a slot, and if we are overflowing, they will jump to the `grab_lock` phase. In theory, there could be 3900 producers at exactly the same time all fetch and add, before any of the others have a chance to set the overflow. Not only does this require 3900 cores, but they all have to be writing to the exact same UCQ. If this happens, I have a warning that will go off.

The next part is getting a new page and linking it with the old one (line 24). The kernel simply does an `atomic_swap()` on the `spare_pg`. If there is a spare already mmaped, the kernel uses it. Otherwise, we need to mmap a new page. The spare page keeps the amount of `mmap()` and `munmap()` calls to a minimum; they are only needed when we grow the UCQ beyond two pages. Either way, we link the old page to follow to the new page. We set the index to a good value, then unlock and clear the overflow flag.

When we set the counter, we set it to 1 (line 38), instead of 0, thereby reserving slot 0 for the kernel thread performing the operations. Aside from being slightly more efficient, we prevent a denial of service from the user. The user could muck with the `prod_idx` endlessly, in a way that seems benign. By reserving a slot, we make sure that any time the kernel grabs the hashlock that it gets a good slot. Whenever we fail to get a slot we lock, and whenever we lock we get a good slot.

How UCQs Work: Consuming

Consuming is similar to producing, but with slightly different concerns. Although the kernel could not spin or wait on the user, the user can wait on the kernel. But the user needs to be careful of spinning on other users, which may be preempted.

The consumer calls `get_ucq_msg()` to copy-out the next message from the UCQ, if one is available. If none are available, this does not block to wait for one.

```
int get_ucq_msg(struct ucq *ucq, struct event_msg *msg)
{
  uintptr_t my_idx;
  struct ucq_page *old_page, *other_page;
  struct msg_container *my_msg;
  struct spin_pdr_lock *ucq_lock = (struct spin_pdr_lock*)(&ucq->u_lock);

  do {
loop_top:
    cmb();
    my_idx = atomic_read(&ucq->cons_idx);
    if (my_idx == atomic_read(&ucq->prod_idx))
      return -1;
    if (slot_is_good(my_idx))
      goto claim_slot;
    spin_pdr_lock(ucq_lock);
    my_idx = atomic_read(&ucq->cons_idx);
    if (slot_is_good(my_idx)) {
      spin_pdr_unlock(ucq_lock);
      if (my_idx == atomic_read(&ucq->prod_idx))
        return -1;
      goto claim_slot;
    }
    old_page = (struct ucq_page*)PTE_ADDR(my_idx);
    while (!old_page->header.cons_next_pg)
      cpu_relax();
    atomic_set(&ucq->cons_idx, old_page->header.cons_next_pg);
    while (atomic_read(&old_page->header.nr_cons) < NR_MSG_PER_PAGE) {
      cpu_relax_vc(vcore_id()); /* pass in self to check everyone else*/
    }
    old_page->header.cons_next_pg = 0;
    atomic_set(&old_page->header.nr_cons, 0);
    other_page = (struct ucq_page*)atomic_swap(&ucq->spare_pg,
                                               (long)old_page);
    if (other_page) {
      munmap(other_page, PGSIZE);
      atomic_dec(&ucq->nr_extra_pgs);
    }
    spin_pdr_unlock(ucq_lock);
    goto loop_top;
claim_slot:
    cmb();
  } while (!atomic_cas(&ucq->cons_idx, my_idx, my_idx + 1));
  assert(slot_is_good(my_idx));
  my_msg = slot2msg(my_idx);
  while (!my_msg->ready)
    cpu_relax();
48   rmb();  /* order the ready read before the contents */
49   *msg = my_msg->ev_msg;
50   my_msg->ready = FALSE;
51   wmb();  /* post the ready write before incrementing */
52   atomic_inc(&((struct ucq_page*)PTE_ADDR(my_idx))->header.nr_cons);
53   return 0;
54 }
```

The first thing to note is that, unlike with producing, consumers compare-and-swap (CAS) to claim a slot (line 43), instead of fetch-and-add. I need to use CAS, since we do not want to advance the index unless the UCQ is not empty. The kernel could blindly advance, since it was the producer and would advance the index regardless.

Emptiness is defined as the `prod_idx == cons_idx` (line 12). Both the producer's and the consumer's "next" slot is the same, so there are no items to be consumed. In that case, the UCQ is empty, so we simply return.

In the common case of a non-empty UCQ, the next available slot is good. The consumer immediately tries to claim the slot with a CAS (line 41), which atomically advances the consumer's index. Once the consumer has secured a good slot, and it must spin until the kernel has loaded the memory with a message. Then we just copy-out the message and increment the "number consumed" counter. The copy-out is important since as soon as we incremented the "number consumed" counter (line 52), the page containing the message could be freed.

As with the producer's side, things get interesting when the slots are bad. If the `cons_idx` is bad, it means the user needs to go to the next page, and to do so it must lock first. As in the kernel, once a thread grabs the lock, it checks to see if the locker's work has been completed, and if so it simply restarts. In this case, if the `cons_idx` slot is good again, the work is complete (line 18). We use a PDR lock (line 16), which is a type of lock that can detect and recover from preemption. I discuss PDR locks in Section 6.2. It is important that userspace does not busy wait on code on another vcore without somehow handling preemption, in accordance with the rules for vcore context outlined in Section 4.4.

Once the lock is held, the consumer thread needs to set the consumer index such that it tracks the first slot in the next page in the linked list. But first, it must wait until the kernel has posted the new page. It is possible that the kernel is in the process of adding a new page to the linked list. This information is stored at the top of the current page of slots (line 25). Once the kernel posts the next page, we can set up the `cons_idx` so consumers can grab slots. The consumer does not need to reserve a slot for itself, since there is no DoS risk. Note that as soon as a valid `cons_idx` is written to the ucq struct, new consumers can come in and immediately consume new items, even though an "older" consumer is still holding the lock.

After setting up the counter, the lock-holder's next job is to free the old page of consumed items. Before discarding the old page, the consumer needs to be sure there are no other consumers still accessing the old page. Even though we hold the lock, that lock only protects threads mucking with the list metadata — a previous consumer could have just claimed the last slot and is still copying out the message. We statically know how many items there are on the page, and the per-page variable `nr_cons` tracks how many messages were consumed, similar to a reference count. Only when all the messages were consumed can the consumer proceed to free the page (line 28). When the consumer thread waits here, it has no idea which vcore it may be waiting on, so `cpu_relax_vc()` must be given the `vcore_id` of the consumer, as discussed in Section 6.2.

Similar to the kernel's side, the consumer simply does an `atomic_swap` the old page with the `spare_pg`, making the old one the spare. If there already was a spare, then we have too many pages and need to `munmap()` the old spare (line 36). The consumer only calls `munmap()` when we are shrinking a UCQ that was greater than two pages. Finally, after unlocking, the consumer thread tries again from the top (line 40). Recall that a new consumer could have arrived and consumed any remaining messages, and the queue could be empty.

Notes on UCQs

There are a few other subtle points involved with the synchronization and maintenance of the page lists.

The consumer only attempts to claim good slots, never bad ones. Yet while claiming the good slot, the `cons_idx` could be advanced to a bad slot, if the slot claimed was the last good slot on a page. It is also possible the kernel is in the process of adding an item, and has set the `prod_idx` equal to the same bad slot value as the `cons_idx`. If the two indexes are equal, the consumer thinks the UCQ is empty, and will return without waiting or otherwise trying to get a message. The producer is in progress, but it has not completed yet. There is no problem here; merely some potentially interesting situations where multiple consumers may see the UCQ at different states. There is no guarantee that a consumer will see a message before it is completely posted.

For both the producer and the consumer, once the corresponding index (`prod_idx` and `cons_idx`) are set to valid entries, any new producers and consumers will take the fast path and claim slots, regardless of whether or not the original thread is still holding a lock. It is important to both set the page and set the slot as one atomic unit; this is why the two values are mixed into one `atomic_t` (a `long`). These index variables are special: they are both protected by the lock and atomically incremented. The lock protects who resets them, with the understanding that it should only be done when everyone agrees the slot is bad, but the normal atomic operations happen lock-free.

When slots are bad, kernel producers need to wait while another producer fixes up the state of the UCQ. The only acceptable place to spin is on the process's hashlock. Any other scheme will probably fail, even if all you want to do is swap the spare page and reset the counter, since those steps need to happen atomically too. The hashlock is what ensures that setting the spare page and resetting the counter happen atomically.

Userspace will `mmap()` a large section of memory and then initialize multiple UCQs with pages from that range. When a UCQ grows and shrinks, that mmapped region may fragment slightly. Additionally, the `mmap()` and `munmap()` calls add in overhead. One optimization that could be added to UCQs would be to expand the `spare_page` to be an array of spare pages. We would only need memory mapping operations when the array was empty or full, thereby amortizing the costs of those system calls. In essence, having n spare pages allows the UCQ to only require mmaps when growing or shrinking beyond $n + 1$ pages.

Regardless of the number of spare pages or the lack of contention, the user and the kernel need to lock every time they switch the active page. These swaps occur every 127 messages currently; the messages are 24 bytes, but with the added `bool` in the message container, the unpacked struct is 32 bytes. That can be optimized if the need arises.

It is possible for userspace from any process to *produce* events, if it has access to the memory of the UCQ. If the UCQ's current page is not full, any code can follow the common-case path where slots are good. If the slot is not good, the program would need to make a syscall to have the kernel send the event.

A final note on UCQs is that they can be used in any operating system, not just Akaros. All that is required is a shared memory region and `mmap()`. Of course, the UCQs are part of a larger system: any OS that uses them needs to do something with the events once they are delivered.

5.2 The Event Delivery Subsystem

UCQs are one part of a much larger event delivery subsystem in Akaros. Akaros events are very flexible, allowing userspace control over which events are received, how they are received, and on which core they are received. The primary abstraction is the *event queue*, which encapsulates both the UCQ and the various configuration parameters. For any activity that userspace wants to receive an event, it merely provides a pointer to a `struct event_queue` to the kernel. Userspace can make as many or as few event queues as it likes: a single event queue can be used by multiple producers and consumers.

In this section, I outline the basics of the event queues and how userspace can use them.

Event Queues

To get started, let us start with the structures.

```
struct event_msg {
    uint16_t                    ev_type;
    uint16_t                    ev_arg1;
    uint32_t                    ev_arg2;
    void                        *ev_arg3;
    uint64_t                    ev_arg4;
};

struct event_mbox {
    struct ucq                  ev_msgs;
    bool                        ev_check_bits;
    uint8_t                     ev_bitmap[(MAX_NR_EVENT - 1) / 8 + 1];
};

struct event_queue {
    struct event_mbox           *ev_mbox;
    int                         ev_flags;
    bool                        ev_alert_pending;
    uint32_t                    ev_vcore;
    void                        (*ev_handler)(struct event_queue *);
};

struct event_queue_big {
    struct event_mbox           *ev_mbox;
    int                         ev_flags;
    bool                        ev_alert_pending;
    uint32_t                    ev_vcore;
    void                        (*ev_handler)(struct event_queue *);
    struct event_mbox            ev_imbox;
};
```

An event queue is made up of an event mailbox and various configuration parameters. A mailbox is a UCQ and a bitmap with one bit per event type. For any event queue, a process can request either a payload, which is sent to the UCQ, or a bit, which is set in the bitmap. Whether or not a bit is set is controlled by a flag in `ev_flags`.

The mailbox is only ever accessed through the `ev_mbox` pointer. The mailbox itself can be inside the event queue struct, as is done with the "big" version, or it can be somewhere else in memory. Multiple event queues can actually point to the same mailbox, but with different flags and other settings. Recall from Section 4.2 the per-vcore "preemption data" structure (known as the VCPD):

```
struct preempt_data {
    struct user_context         vcore_ctx;
    struct ancillary_state      preempt_anc;
    struct user_context         uthread_ctx;
    uintptr_t                   transition_stack;
    uintptr_t                   vcore_tls_desc;
    atomic_t                    flags;
    int                         rflags;
    bool                        notif_disabled;
    bool                        notif_pending;
    struct event_mbox           ev_mbox_public;
    struct event_mbox           ev_mbox_private;
};
```

Note that there are two mailboxes per vcore for various messages. When setting up event queues to deliver events to these mailboxes, userspace simply sets the `ev_mbox` pointer to one of those addresses. When a 2LS calls `handle_events()`, Parlib code handles the events in both of these mailboxes. I will discuss the distinction between the two types of mailboxes shortly.

The other event queue setting of important is the `vcore` field. Certain event queue options take a vcore as a parameter, such as which vcore gets notified (IPI) when an event is delivered to the event queue. One thing to note is that if an event queue's mailbox pointer is clear, the kernel will send the event to one of the VCPD mailboxes.

Every event message has a type field (`ev_type`). When userspace handles the events in an event queue, it runs a series of functions from a table of function pointers indexed by the event type. Applications using Parlib can register multiple event handlers per event type; when an event fires, each handler will run with both the payload from the event and a `void *arg` set up during handler registration. This setup is very similar to interrupt handlers in an operating system. One very nice feature of having multiple event handlers per event is that event tracing becomes simply registering a debugging event handler.

Event Queue Options

There are a variety of flags for the event queue that the kernel takes into account when sending events. Some are trivial, and others expose important details about events and MCPs. I will discuss the important points below.

- `EVENT_IPI`: The user wants the kernel to notify a vcore after sending the message to the mailbox.
- `EVENT_NOMSG`: The user does not want a full message (with payload); the kernel sets a bit instead.
- `EVENT_VCORE_APPRO`: If sending an IPI, send it to where the kernel thinks most appropriate, currently only used for certain preemption scenarios.
- `EVENT_VCORE_MUST_RUN`: Only send to a vcore that is either running or will run the next time the process starts up.
- `EVENT_INDIR`: Post the event to the event queue, but then send an *indirection event* (indir) to the vcore's mailbox that points to the original event queue. These are useful for messages that must be unique, since multiple indirs can be sent for a single message.

- `EVENT_SPAM_PUBLIC`: Send the original message to at least one vcore guaranteed to either run, or have its messages checked by another vcore, ensuring the process reads the message.
- `EVENT_SPAM_INDIR`: If the kernel is sending an indir, this flag ensures the kernel will spam the indir so that the process is guaranteed to see the message.
- `EVENT_NOTHROTTLE`: When sending indirs, normally the kernel will not send multiple indirs for the same event queue, to cut down on message traffic. This flag disables that optimization.
- `EVENT_VCORE_PRIVATE`: If the kernel is sending to a VCPD mailbox, send it to the private mailbox.

The distinction between public and private mailboxes for a vcore is about which vcores may handle the messages in the mailbox. The private mailboxes will only be handled by code running on the vcore that owns the mailbox. The public mailboxes *may* be handled by other vcores — a common occurrence when recovering from a preempted vcore. When userspace chooses which type of mailbox to use for an event queue, the developer should ask whether or not the event must be handled by the target vcore. Examples of events appropriate for the private mailbox include messages to reschedule the current thread, run another thread, or to profile the current thread. All of these are very particular to which vcore runs the handler. Events sent to the public mailbox include various process-wide notices, such as "vcore 6 was preempted" — this can be handled by any vcore.

Indirection events (indirs) are important for scenarios where a message can only be handled once. For example, the message that a syscall completed should only be handled once; otherwise the job of the 2LS is much more difficult. In this case, the original message is posted to the mailbox pointed to by the event queue, but then another message of type `EV_EVENT` with a payload containing the event queue pointer is sent to the vcore. The original message exists only once, but multiple indirs can be sent that direct vcores to check the event queue. Exactly one vcore will extract and handle the message; subsequent consumers will not find the message in the UCQ.

Typically, there will not be multiple indirs for a message, but the level of indirection is important in certain circumstances when the kernel is trying to make sure userspace received the indir. Indirs are another example of the design tactic of "safely poking". It is harmless to send an indir to a vcore to look at another event queue, since that event queue ensures a single copy of a message exists.

Spamming messages is when the kernel guarantees that the process will receive the message and will wake up sleeping processes so they handle the message. Multiple vcores may receive these messages, typically in the public mailboxes. The particulars of how spamming works are described below in Section 5.3. The important part for now is that the message will be handled at least once, regardless of the process concurrently yielding vcores or being preempted. However, the message may be handled multiple times, and in certain circumstances, old, stale messages may linger in mailboxes of offline vcores.

Note that spamming only relates to the mailboxes where events are sent; as with all cases, IPIs are an additional option. If the `EVENT_IPI` flag is set, every time the kernel sends an event to either an event queue or a vcore's mailbox, it will attempt to notify the appropriate vcore. A spammed message will be seen at some point in the future, specifically the next time the targeted vcore enters vcore context and handles events. Notifying a vcore is the tool to make sure that happens immediately, since notification puts the vcore into vcore context. Otherwise, a vcore may be running a uthread for a long period of time. There are two things of note here. Regardless of the `IPI` flag, if the kernel sends a message to a vcore (spammed or an indir), it will set `notif_pending` to ensure the vcore sees the message before leaving vcore context. Additionally, a vcore must go through vcore context to yield to the kernel, and the kernel will ensure that the vcore does not yield while `notif_pending` is set.

The `SPAM_INDIR` option applies to sending indirs. It is an instruction to the kernel to spam indirs to vcores, guaranteeing a message is received. It is not clear to me whether or not an application would ever want an indir without the guaranteed reception. If a message can be handled only once, is its nature such that it must be handled exactly once? Supporting both spammed and un-spammed indirs comes at no cost, so I have kept the option in place until we have more experience with event queues.

An important detail to note is that spamming and notifications (IPIs) leverage an Akaros design principle: make spurious messages harmless. First discussed in Section 3.2, safe spurious events, sometimes referred to as "safe poking", provide a layer of indirection that allows me to avoid certain race conditions and grabbing locks. In the case of event delivery, it allows me to send multiple messages to different mailboxes to *poke* vcores to check the real underlying event queue. Indirs and any spammed message are spurious. Similarly, even the notification code can send spurious IPIs. Within the kernel, the kernel does not even bother locking the process; instead it sets `notif_pending` and makes an unlocked read to determine the physical core id, and sends the `__notify()` kernel message. If the message was sent to a stale core, the handler notices and ignores the message. Likewise, the `notif_disabled` flag could be unset at the time of sending the message, but then turned on by the time the IPI traps the remote core. Again, the kernel message handler makes the final check before carrying out any actions. Finally, vcore context itself is designed spuriously; other than performance, there is no harm in dropping a vcore into vcore context. Making spurious signals harmless greatly simplifies the system.

Guidelines for Using Event Queues

Event queues have a variety of options and use cases. A few examples and guidelines will help clarify their usage.

First off, how does userspace actually use event queues? Any event queue pointer can be passed to the kernel for a particular event. The kernel maintains a rudimentary array of event queue pointers, one for each type of event, in procdata. A process can register an event queue for any of these at will. The kernel event table is not used for very much at this point. Uthread initialization code sets up event queues for preemption handling and interprocess POSIX signals, the latter of which sends the signal number as part of the payload.

The real power of the event queue pointer is that they can be passed to the kernel in any context. The kernel event table is just one place where userspace can sign up for an event. Any subsystem, and any *future* system, that wants to send an event merely needs a way to get an event queue pointer from userspace into the kernel. Furthermore, the same event queue can be used for multiple event sources.

The most common place to drop an event queue is in a syscall struct. Recall from Section 3.2 that all syscalls are asynchronous, with the arguments and return values in a `struct syscall` in user memory. One of the fields of the struct is `struct event_queue *ev_q`. When a 2LS wants to block a uthread on a system call, it writes the `ev_q` pointer and sets a flag in the syscall struct, as described in Section 4.4. When the kernel finishes the system call, it checks the flag and sends an event to the event queue.

Another case where userspace passes event queue pointers to the kernel is the alarm service, called "#A" in the Plan 9 namespace. Kernel-implemented devices are all named with a "#" and exist in a global namespace. Userspace can request the kernel to fire an event after a certain amount of time. It does so by creating an alarm and writing the event queue pointer to a control file.

Now that we know how to register for events, what settings should be used for the event queue flags? First, is it important to receive the message, even during yields, preemptions, and if asleep? For instance, generic syscall completion events may fit this mold, depending on the 2LS. If so, then the flags must include a spamming flag: either `EVENT_SPAM_PUBLIC` or `EVENT_SPAM_INDIR`.

Next, can the process handle receiving more than one copy of the message? If not and it is spamming, then it must use an indir. If spamming is not requested at all, then the kernel will not attempt to send multiple messages anyways. The rule is that if a message must be unique, then it cannot be spammed. If a message (or an indir) must be guaranteed, then it must be spammed. It is possible for an application or 2LS to build their own guarantees for their own event queues by hooking into the preemption handler.

If it uses an indir and the handler might not return, then `EVENT_NO_THROTTLE` must be set. The reason is that throttling tells the kernel to not send an indir for every message in the main event queue. In this case, the kernel may notice the event queue has not been checked and then, as an optimization, not send an indir, knowing that some vcore still needs to handle its indir. The assumption is that once a vcore starts handling messages from an event queue pointed to by an indir, it will do so until that queue is empty. If a handler does not return, this assumption is broken.

Next, does the event need to be handled immediately? If so, set `EVENT_IPI`, so that the kernel notifies a vcore. This question is about control flow, compared to the two previous questions which are about to whom and how many events are delivered.

Finally, does the event pertain to "vcore business", meaning it is only appropriate for the target vcore to receive the message? If so, set `EVENT_VCORE_PRIVATE`.

Looking at a few event queues in particular:

- Pthread syscall completion: `INDIR`, `IPI`, and `SPAM_INDIR`. The 2LS can only handle one message per syscall (`INDIR`), and it must receive the message (`SPAM_INDIR`). The IPI is so that the 2LS can immediately notice when a thread unblocks and request additional vcores if necessary.

- #A alarm service: `IPI` and `SPAM_PUBLIC`. The alarm service can handle multiple messages. The messages are actually just a "poke" to check for any expired alarms, since multiple alarms in userspace are multiplexed over a single kernel alarm. Any extra messages will just check the list again. The spamming is necessary to wake up the process or otherwise make sure we do not miss an alarm.

- Per-vcore alarms: `IPI` and `VCORE_PRIVATE`. Kevin Klues built a per-vcore alarm service, initially used for profiling, using #A. This alarm system only runs when the vcore runs, hence the lack of spamming options. The alarm is particular to the vcore, and not the process in general, so the private flag is set. The private flag is incompatible with spamming. Note that both the per-vcore alarms and the process-wide alarms both use the underlying #A service — the only difference is the flags to the event queues.

- Early Single-Core Process (SCP) syscall handling: `IPI` and `NO_MSG`. Even SCPs can handle events, though since they are not MCPs, many of the options related to vcores are meaningless. In general, SCPs are treated as if they run on vcore 0. In the case of early syscalls before Parlib initializes, userspace will not actually handle their events, and sending no message payload is appropriate.

There are a few other things programmers that use event queues should know.

For one, SCPs can receive events, with payloads. Very early on in the life of an SCP, while the program is still running glibc code, such as ld.so, events are not available. As soon as the Parlib code initializes, regular SCPs can receive events. They actually use vcore context for Vcore 0, but they are not pinned to a core. Using vcore context in this manner is similar to using a signal handling context in other Unix systems, and it benefits from sharing code with the more common MCP operations. From the kernel's perspective, very little event code changes to support SCPs.

Under certain circumstances, the kernel may send events without an event queue. One example is the `sys_self_notify()` syscall, whereby a process can notify one of its vcores. In these cases, the kernel sends directly to a vcore's mailbox, since otherwise the kernel would not know to which mailbox to post the event. I considered reflecting unhandled traps and faults in a similar manner, as an event without an event queue, delivered to the vcore that had a faulting uthread. This approach is inappropriate and buggy; the fault is a property of the thread itself and should stay with the thread. There are scenarios where the event could be handled when the thread that triggered the fault is no longer the `current_uthread`, possibly tricking the 2LS into thinking a different thread faulted.

There are two parts to event queues that provide levels of indirection: the indirs and the `ev_mbox` pointer, and both can be used in different ways. For instance, a 2LS could set up a big event queue, with the mailbox pointing to the storage space in the event queue, set the flag to `EVENT_IPI`, and then poll that event queue in the scheduler-specific `vcore_entry()`. Alternatively, the 2LS could set up a small event queue and point `ev_mbox` at a vcore's private mailbox and set the IPI flag. These approaches are almost equivalent; the difference is that all vcores would poll and could receive the messages in the former case, while only one vcore could receive messages in the latter. Either way, both the indirs and the mailbox pointer are tools for redirection, and can be used any way a programmer wants.

I also built application-specific event handling into the userspace event subsystem. An application can place a handler function pointer in the event queue (`ev_handler()`) and set an aptly named flag `EVENT_JUSTHANDLEIT`, and the handler will run in vcore context. This approach differs from the usual table of event handlers in that the `ev_type` of the event does not matter — this particular handler will run regardless of the message that is sent to the event queue. Another option exists (`EVENT_THREAD`) where the event code will ask the 2LS to spawn a thread to run that handler. The 2LS does not actually have to spawn a thread; it can wake up a sleeping thread or do whatever it wants, but the intent is that the handler is executed in a thread's context. Handling events in threads is safer since uthreads can block and fault, while if a vcore faults the process dies.

No existing 2LS uses these capabilities yet, but they are examples of the sort of thing that can be done. I am sure some of these options will be removed and others will be added as the system matures. There are many options available, and the changes required amount to a very small amount of code.

As one example of a future feature, I could have a 2LS in which uthreads block on event queues directly. This approach would be very similar to the existing `EVENT_THREAD` option, in which a 2LS could have a per-vcore pool of uthreads to service events in response to the `spawn_thread()` upcall. The slight difference is the programming of the uthreads: instead of exiting to sit in a pool, they could block on a specific event queue. Blocking on an event queue requires more bookkeeping, but it is a little more powerful in that the queue can be used to hold work until a uthread is ready to handle it. In contrast, the `spawn_thread()` upcall expects the event to be handled right away; if the per-vcore uthread pool is empty, the 2LS needs to store the event until a uthread becomes available.

Imagine the uthread blocks on an empty event queue, attempting to extract a message. When an event arrives, the uthread wakes up and receives the message as a return value to the blocking call, handles the event, then reblocks. An application could ship work to vcores in this manner, where the per-vcore scheduler simply checks its per-vcore event queues and runs uthreads until they block again. No IPIs would be needed, since the vcore checks its queue whenever it runs out of work to do. Regardless of whether uthreads block directly on an empty event queue or if the 2LS maintains a pool of generic uthreads for event handling, the event queue can be used as a means of shipping work.

Not only could the application ship work between its cores in this manner, but the kernel could also ship work to userspace, without syscalls. A common scenario involves an application, such as a webserver, listening on a socket for incoming calls. With minor changes to the kernel, we could modify the `listen` syscall to send an event containing the file descriptor of the incoming connection instead of returning via the usual syscall paths. (Actually, there is no `listen` syscall; to listen in Plan 9, you simply open a file such as `/net/tcp/123/listen`, but that is a minor point.) The kernel sends the event directly to userspace; it can send events to remote cores or queue them up until the kernel returns to userspace. In the latter case, the application could have requested to receive the network interrupt on one of its cores for cache locality reasons. As the workload changes, userspace can increase the pool of uthreads or request more vcores if the event queue's length grows too long.

It should be noted that UCQs may not be suited to unlimited inbound connections on a socket. Although the kernel would run out of memory in the networking stack before a process ran out of memory in the UCQ, it may be nice to have some form of back-pressure from the application. Clearly we can have a parameter to listen that limits the amount of new connections in an opening phase. A more elegant solution could be to use BCQs instead of UCQs for this particular event handler. The event system is designed such that BCQs, or any other type of delivery system, could be used inside an event mailbox. Of course the BCQs could overflow and are not suitable for all event types, but in the case of the listening socket, the kernel could simply drop the connection.

Ultimately, there are many options for event queues and many opportunities for future growth. Most new features will be developed in particular second level schedulers. Those features that are sufficiently powerful or general to all 2LSs will be implemented in Parlib so that all schedulers can use them.

5.3 Never Miss an Important Event

An important aspect of event delivery is the ability to guarantee the process sees specific events. When an event queue has the `EVENT_SPAM_PUBLIC` flag or the `EVENT_SPAM_INDIR` flag with an indir, the kernel and user work together to make sure the process will handle the event.

Missing events is a more difficult problem for M:N, core-based systems than 1:1 thread-based systems. For 1:1 systems, the kernel just sends the event to the kernel task, and the next time that thread runs it will receive the message. Internally, the kernel deals with Saltzer's "wakeup waiter" problem[94] — the race between a waiter that sleeps and a waker that signals the waiter. Unix signals, for instance, interrupts the thread and runs a signal handler on its next execution. With process groups in 1:1 systems, the kernel just needs to pick a particular kernel task and deliver the signal.

With MCPs, the difficulty lies in the fact that a vcore may stop running and *might* never run again. With 1:1 models, threads always run again, until they are dead. The underlying issue is that MCPs manage their own vcores, and with these capabilities comes difficulties and responsibilities. One of those responsibilities is to work with the kernel when yielding a vcore to not miss a waking event. Event delivery is powerful and flexible; this power came from decoupling the event delivery from the vcore's execution and control flow. Now we need to deal with the cases where the application may want to recouple those by basing its control flow decisions on a particular event.

For an example of why it is important to not miss events, consider an MCP with one vcore that issued a blocking system call. The system call could be a read on a socket, or some other blocking call. Until the syscall completes, the process may have no work to do and want to yield. If the last vcore starts to yield right as the event fires, it is possible the vcore does not see the event and yields forever. Contrast that scenario with synchronous syscalls on a 1:1 threading system; in the latter case, when the syscall completes the thread just restarts — the kernel handled everything. In the 1:1 system, userspace did not have the opportunity to perform work while the syscall was blocked, but it also did not need to deal with missing events.

The *Pause* Technique

Now is a good opportunity to discuss a technique I use for analyzing synchronization schemes for concurrency bugs. The *pause* technique is very simple and can prove a scheme has a bug by clearly expressing a situation where the scheme would fail. Given multiple, concurrent streams of instructions, simply pause one of the streams at an opportune moment and execute as many of the other streams as you can. The *pause* technique gets to the heart of concurrency: instruction streams can interleave in any manner. As simple as it sounds, I have found many bugs with this method. More importantly, I can convince other people or myself that a scheme will fail in a clear manner.

In the case of missing events, the vcore code is trying to check for events and yield (Listing 5.1) while the kernel is trying to send an event (Listing 5.2). As an added complication, the kernel may be preempting vcores in addition to any voluntary yielding caused by userspace (not shown here). This code is clearly flawed, and is not what Akaros does, but shows what we would like to do. A scheduler checks for events, then yields if there are no events. If it does not receive the event, it may never wake up again — clearly a problem. Additionally, the faulty yield code serves as a basic example of the pause technique. It can find bugs, but it cannot prove bugs do not exist.

```c
void some_scheduler()
{
  handle_events();
  /* no events, decides to sleep */

  /* PAUSE */

  yield_vcore_to_kernel();
  /* never run again */
}
```

Listing 5.1: Vcore Yielding

```c
void syscall_completion_event()
{


  vc = find_online_vcore();
  send_event(vc, syscall_done);
  confirm_vc_still_online();


}
```

Listing 5.2: Kernel Sending Event

As a side note, debugging code in virtual machines can expose "pause based" race conditions that one would never expect on real hardware. Due to the nature of virtual machines, physical cores in the VM can be taken offline for a large quantum, based on the underlying host OS, creating an abnormally long pause in an instruction scheme. At the other extreme, many bugs require running on hardware to be exposed, since the nature of those races may require high speeds or true parallelism. When debugging, both hardware and virtual machines have their uses.

Yield with a Little Help from the Kernel

As an alternative to using the `EVENT_SPAM_INDIR` flag, when yielding a vcore, userspace could manually change the `ev_vcore` setting in every event queue to point to another vcore. That way, if the vcore yields, a message will still make it through to the process. First, userspace needs to pick another vcore that is not concurrently yielding. Second, the 2LS must change all event queues for the yielding vcore to the new target — including event queues set up by the application. Finally, it must handle any messages in each of those event queues. This approach, while correct, is a little problematic. Implied in these steps is that the 2LS must maintain a per-vcore list of event queues or some other means of finding the vcores. If the yielding vcore ever comes back, the 2LS will probably want to reassign the event queues back to the vcore, which can get complicated. The vcores must also work together when a single vcore wants to yield — once a vcore is picked as the target of event queues, it probably should not yield until the transfer is complete. Lastly, the 2LS needs to special-case the scenario where the last vcore yields; there is no other vcore that can receive event queues. For all of these reasons, the "spamming" solution I use involves both the kernel and the user. With a little help from the kernel, yielding and guaranteeing event delivery is much simpler.

One of the properties of vcore context is that so long as the per-vcore flag `notif_pending` is set, a vcore will not be allowed to leave vcore context. Both userspace and the kernel work together to maintain this property, as described in Section 4.2. To reiterate, `notif_pending` means that a vcore has missed something and must recheck everything it normally checks in vcore context. The exact meaning of "recheck everything" depends on who or what originally attempted to *notify* the vcore and the nature of the 2LS.

In the case of event delivery, whenever a message is delivered specifically to a vcore's mailbox, the `notif_pending` flag is set. These direct messages happen as part of the "spamming" process, which applies to `EVENT_SPAM_PUBLIC` and `EVENT_SPAM_INDIR` messages. As a side note, the kernel does not set `notif_pending` if a vcore's mailbox was incidentally delivered to, such as when an event queue happens to use a vcore's mbox for the `ev_mbox` pointer.

Userspace's role in not leaving vcore context while `notif_pending` is set involves checking the flag after reenabling `notif_disabled`. See Section 4.2 for more details. The kernel's role is a little easier — do not let the vcore yield while `notif_pending` is set. The semantics of `vcore_yield()` are thus slightly different than `sched_yield()` on other operating systems: yield can fail.

As I have mentioned before, the problem of vcores missing messages is analogous to Saltzer's "wakeup waiter" problem[94]. The kernel's event delivery system is the wakeup code and the waiter is vcore context. Saltzer solved the problem with a "wakeup waiting" switch, described as follows. The "wakeup waiting" switch was a variable that was cleared by the waiter. The waiter clears the switch, checks its work queue, then blocks. The kernel's block routine would abort and wakeup if the "wakeup waiting" switch was set. The wakeup code puts a task into the work queue of the waiter, then wakes the waiter — which turns on the switch and wakes the waiter if it was blocked[94].

When vcore context is the waiter, there are a few additional issues, but the overall pattern is the same. To complete the analogy, `notif_pending` is the "wakeup waiting" switch. One distinction is that there are two separate ways for vcore context to "wait." Vcore context could pop to a uthread or it can yield to the kernel, either of which could result in a missed message. Both exit paths from vcore context must clear `notif_pending`, check for events, and then abort the exit if `notif_pending` is set. Another complication is that the waiter code is in userspace and the wakeup code is in the kernel. Likewise, kernel code enforces that a yield cannot happen when `notif_pending` is set. The "wakeup waiting" switch is a shared-memory technique. By placing `notif_pending` in procdata, user-level code can easily participate in the protocol, as opposed to the kernel providing a syscall to clear `notif_pending`.

A final distinction arises in how the kernel performs its duties in handling the vcore's yield syscall and in delivering wakeup events. The kernel's role in yielding is made more complicated by one other goal: we want to avoid locking while sending events and to be sure an event was delivered. Ultimately, there is a race within the kernel where some code sets `notif_pending` at the same time as a yield. It is impossible to guarantee that `notif_pending` will not be set after a yield, for instance. Instead, the guarantee is that for a successful yield there was a moment when vcore was no longer online and `notif_pending` was not set.

As a brief aside, the kernel maintains three linked lists of `struct vcore`: online, bulk-preempted, and inactive. As far as the overall process is concerned, a vcore is online if it is on the online list, even though there may be a slight delay before the vcore is actually running on a specific pcore (the kernel message may be in flight at the time). The bulk-preempted list is used when the kernel preempts the entire process at once; the invariant is that at least one of the online and bulk-preempted list must be empty. The inactive list is simply all remaining vcore structs. These lists are important for lockless event delivery.

```
 1 void yield_path()
 2 {
 3   /* in user */
 4   vc->notif_pending = FALSE;
 5   wrmb(); /* barrier */
 6   handle_vc_mboxes();
 7   yield_vcore_to_kernel();
 8 
 9   /* in kernel now */
10   if (vc->notif_pending)
11     return; /* abort */
12   lock_for_lists();
13   remove_from_active_list();
14   wrmb(); /* barrier */
15   if (vc->notif_pending) {
16     add_to_actives();
17     unlock_for_lists();
18     return; /* abort */
19   }
20   unlock_for_lists();
21   finish_yield();
22 }
```

Listing 5.3: Improved Vcore Yielding

```
23 void event_delivery()
24 {
25   /* in kernel */
26   vc = first_active_vc();
27   while (vc) {
28     send_vc_msg(vc);
29     wmb(); /* barrier */
30     vc->notif_pending = TRUE;
31     wrmb(); /* barrier */
32     if (vc == first_active_vc())
33       break;
34     vc = first_active_vc();
35   }
36 }
```

Listing 5.4: Kernel Spamming Event

The result of using `notif_pending` is shown in pseudocode in Listings 5.3 and 5.4. This is not the actual code used in Akaros, but is more of a stepping stone to understand the issues in event delivery, as well as to explain how the kernel deals with `notif_pending`. In the case of the `notif_pending` flag being 0, it can only be disabled at line 4. If userspace clears the flag, set at line 30, then it must have received the message sent at line 28.

In the case we are interested in, `notif_pending` is set true at some point after line 4. The kernel's yield path uses the "check, signal, check again" pattern that I frequently use for shared-memory synchronization. The first check at line 10 is an optimization to avoid grabbing a lock and performing heavier synchronization. The real protection comes from the "signal" at line 13, whereby the yield path tells the event deliverer that the vcore is unavailable, and the "check again" at line 15. For its part, the delivery code signals at line 30 and checks at line 32. If the check at line 32 succeeds, then the `notif_pending` write happened before the vcore was removed from the list, with a caveat described below. In that case, the yield path will see it at line 15 and abort the yield. If the check at line 32 fails, then the yield may or may not be aborted, and the sender must attempt to send to another vcore. The repeated attempts at sending messages are why I call this "spamming" and why messages that are spammed must be able to be received multiple times.

Note that the event deliverer looks at the active vcore list without grabbing a lock at lines 26, 32, and 34. The lock protects the list from modification. It is safe to look at the first element, since we do not follow the links (which could result in a cycle) and since the memory for the vcore structs are never freed so long as we hold a reference on the process. Of course, as soon as code reads the list head pointer, it could be changed.

A related issue is an apparent assumption that since the vcore is still at the head of the list that the list did not change. This is the classic "ABA" problem[43], where pointer equality is confused with "no state change." I do not assume that the list did not change, though it may appear that I do so. It is possible for a long *pause* at line 28 where the entire kernel yield path executes from line 8 through 21, without userspace having received the message. However, since the vcore is now back on the head of the active list, that means the vcore has since restarted and will either check the message or yield *again*. On this later yield, `notif_pending` will be set and the message will be received. I do not care about a *particular* yield racing with the event delivery; I care about the *final* yield. Note that the ABA issue has nothing to do with the successful yield guarantee that there was a moment when vcore was no longer online and `notif_pending` was not set. The issue I described was about event delivery code thinking it succeeded where it may have failed, which it did not.

There are a few other points of interest regarding the use of the first element of the active list. First, being the first member of the active list is a weaker statement than being "online". A vcore could be deeper in the list and the algorithm would be just as correct. However, the only item I can look at safely, without a lock, is the first item of the list.

More importantly, the lists are actually mainly used to *find* a vcore. If a sender already knew who to send a message to, it could dispense with checking the first list and check procinfo to see if a vcore was mapped to a pcore or not. I check the active list only when the original target of an event queue (the `ev_vcore` field) was unreachable (described later). Additionally, I do not only use the active list: I also check the bulk-preempted and inactive lists.

Finally, while the `notif_pending` check is a powerful means of making sure userspace does not miss a message, I actually have another flag I use for event delivery that simplifies certain scenarios. Every vcore has a flag in procdata in the VCPD called `VC_CAN_RCV_MSG`. When this flag is set, userspace will make sure it handles that vcore's mailboxes, which a more limited and specific statement than `notif_pending`'s "do whatever checks are needed upon entering vcore context."

All of the notif_pending checks mentioned above are in fact done — they are the basis for not missing notifications, but `VC_CAN_RCV_MSG` helps in a few ways. `VC_CAN_RCV_MSG` tells us clearly if a message delivered to a specific vcore will be handled. We do not always want to check vcore lists to confirm "onlineness," and for almost all event delivery, the first step is to try and send a message to a specific vcore. `VC_CAN_RCV_MSG` is similar the "mapped" flag in procinfo, but is under the control of userspace. The flag differs from being mapped in that some vcores may be offline but still able to receive messages. One such scenarios includes preemption recovery. Preempted vcores are not yielded: the kernel yanks them away. Eventually some other core will handle the preemption, including dealing with its messages. This includes bulk preempted vcores. The final benefit of `VC_CAN_RCV_MSG` is that in some situations the kernel determines a vcore is messageable and sets the flag. Once the flag is set, the kernel can simply send future events.

For the most part, userspace owns `VC_CAN_RCV_MSG`, but the kernel will turn it on under certain circumstances. When a vcore starts up, the kernel knows it will start checking messages, so it sets this flag to help out userspace. The flag does not mean a vcore is online, only that its mailboxes will be checked. Additionally, when trying to deliver a spammed message, the kernel may need to grab locks and be absolutely certain a vcore is either running or will run soon; in those cases the kernel will set the flag as well.Userspace toggles the flag in a few circumstances. Most notably, it turns off the `VC_CAN_RCV_MSG` when yielding, before checking messages; it would be around line 3 in Listing 5.3 (not shown). If a yield fails due to a missed notification or a received message, userspace turns the flag back on. Also, userspace will turn off the flag for *another* vcore in the process of handling preemptions, discussed in further detail in Section 6.1.

To sum up the overall process, the event delivery spamming function, `spam_public_msg()`, will first try the target vcore. If its `VC_CAN_RCV_MSG` is set and the `EVENT_VCORE_MUST_RUN` is not set, then the message will be handled. Otherwise, the code hunts for a vcore to which it can send the message.It checks the first member of the actives, then the bulk preempts, then even the inactives (under certain circumstances). When spamming list members for the actives or bulk preempted, we rely on list membership and `notif_pending` as described above. The point here is that `VC_CAN_RCV_MSG` is useful for confirming that a message sent to the vcore will be handled, but since we went through the effort of hunting for a vcore that we know will run, we can leverage that extra work and simply be done with message delivery. Finally, if all of those options fail, the code grabs the process's lock to ensure that no vcores are yielding, being granted, or being preempted. At that point, knowing there are no concurrent changes, it is fairly easy to find a vcore, deliver the message, and set `VC_CAN_RCV_MSG`.

Ultimately, userspace and the kernel work together to make sure events can be delivered to the process, with a guarantee that the process will actually handle the message, regardless of concurrent yields or preemptions. In the common case vcores are online and easily messageable. Of course, Akaros needs to be able to handle the rare corner cases. Flags like `notif_pending`, with its ability to force vcores to remain in vcore context, and `VC_CAN_RCV_MSG`, with its ability to help the kernel easily determine if a core is messageable, enable the system to work through those corner cases.

Dealing with these scenarios is trickier in the MCP environment — the process has control of its vcores, and likewise it must perform its part in event delivery. Most applications and even 2LSs will not need to worry about the intricacies of event delivery, which is dealt with in the Parlib library. Using the correct flags on event queues and following the Event Queue Guidelines is enough to ensure that Akaros does the right thing. The most important guideline is that spammed messages may be handled multiple times: if this is not acceptable, then use an indir. Many aspects of Akaros are built on a similar principle of "safe poking", where spurious events happen in rare situations. One source of these rare situations as well as a major client of the reliable event delivery subsystem is when the kernel preempts vcores, as we will see shortly.

Chapter 6

Virtual Core Preemption

Eventually, any real system with multiple applications will need to deal with revoked resources. There is a progression of "system needs", starting from the trivial and leading to the more complex and versatile:

- Giving cores to applications is relatively easy. When making a system where processes schedule their own threads on dedicated cores, the first thing a developer tries is simply popping into userspace and never doing anything else.
- Of course, many applications want to do more than just execute and require some form of I/O. Thus the kernel is involved, either directly by executing I/O syscalls or indirectly by enabling IPC, such as for a microkernel. Still, with asynchronous syscalls, the application can handle blocking syscalls and maintain control of its cores and threads without serious hassles. Page faults require work, but it's nothing that vcore context cannot handle.
- The next complexity arises when applications want to yield their cores. After all, many applications have a varying demand for CPU time, and to maximize system throughput, we want those applications to yield. Yielding presents a few issues with event delivery where it is important to not yield and miss an event, as I discussed in the previous chapter.
- The final step in the increasingly complex progression, and the subject of this chapter, is handling the *preemption* of cores.

Ideally, an application would have uninterrupted access to the entire machine. If there is only a single application on a system, then clearly we will never preempt resources. If there are multiple applications with static resource requests and sufficient resources to run all applications in parallel, then, again, there is no reason to preempt resources. In those cases where this is not the case, we are forced to schedule and allocate a limited amount of resources among competing processes. Akaros needs to be able to handle this wider variety of situations. Applications will have changing workloads and resource requirements, as well as changing priorities. The kernel scheduler will want to backfill idle resources with low-priority MCPs. Resource preemption and revocation is a concern once we have competing applications and fluctuating priorities and resource requests.

The Akaros scheduler currently will preempt cores from processes if those cores are provisioned to other processes that have requested more cores, as I discuss in Section 3.2. Otherwise, the current scheduler is first come, first served. Future kernel schedulers are expected to adhere to the provisioning guarantees, but they are also free to implement other policies. One such policy is a round-robin scheduling of blocks of cores, with long time quanta. Another policy could attempt to optimize each application's efficiency, comparing the performance given the amount of allocated resources. Those other schedulers can preempt cores from processes for whatever reason they want; provisioning is just one such reason. The Akaros MCP code and user-level libraries are designed to be able to handle preemptions under any circumstance.

Preemption is notably different than yielding and blocking syscalls. The latter situations are under the control of the application, to some extent. Preemptions are under the control of the kernel. A user can affect preemptions by adjusting the kernel's policy, such as through provisioning, but from the perspective of application code, preemptions are uncontrollable. Of course, the kernel can warn about an impending preemption, but that does not *prevent* a preemption.

The involuntary aspect of preemption makes handling preemption more difficult, especially so, given that applications are expected to control their un-preempted cores. With the propensity for user-level spinlocks and synchronization, preempted cores raise the issue of deadlock. Additionally, guaranteed event delivery, such as with the event queue options `EVENT_INDIR` and `EVENT_SPAM_PUBLIC`, becomes more difficult. The primary goal for dealing with preemption is correctness: no deadlocks and no missed messages. A preemption can come at any line of code in userspace, and a preempted core might never be returned. Even worse, the preemption handler itself could be preempted. The secondary goal is performance. The application or 2LS needs to know about the changed amount of resources so that it can make scheduling decisions. Ideally, we also want to recover from preemption quickly, without grabbing an undue amount of locks or requiring an excessive amount of system calls.

Dealing with preemption has always been an issue in systems that provide user-level locking and scheduling. Recall from our related work in Section 2.3, both Psyche and Scheduler Activations attempted to solve the preemption problem, but were unable to handle all of the cases. Psyche provided a "two-minute warning" and assumed that if a process had not yielded by then that it would eventually run again. Scheduler Activations could not handle rare corner cases, due to the coupling of events to activations. The systems that dealt with preemption and user-level locking had a variety of issues as well; see Section 2.4 for a refresher. As a reminder, the lessons from prior work related to preempted cores are:

- Preemptions will happen, and we need a clean way to detect preemption when it does.
- There are a variety of locking strategies, and one size does not fit all. We do not want to build a particular locking strategy into the kernel interface.

- Detecting critical sections is difficult: unless a thread atomically grabs the lock and marks itself as a lock-holder, there will be false positives or false negatives.
- User-level schedulers make preemption recovery more difficult: if they choose the wrong thread to run, they could deadlock. Additionally, the preemption handling code itself could be preempted.
- Be prepared to handle changing resource allocations. The kernel needs to expose the details of which physical resources are allocated to userspace. Likewise, userspace need to be prepared to adapt and perform well as these resources change.

In the rest of this chapter, I will start by explaining how the Akaros kernel preempts cores and describing a basic system call userspace can use to help recover from preemption. Then I will present user-level techniques for spinlocks that provide preemption detection and recovery, as well as adapting those techniques to custom busy-waiting synchronization. Finally, I will describe how second-level schedulers using the uthread library recover from the preemption of a vcore under arbitrary circumstances.

6.1 Preemption Basics

How the Kernel Preempts Vcores

At a high level, when the kernel wants to preempt a physical core, it first determines the virtual core, and optionally sends a warning message. Userspace controls whether or not it receives a warning event. Regardless, the kernel sets the per-vcore field `preempt_pending` in procinfo, which serves as a read-only timestamp of when the vcore will be preempted. The current userspace code uses the field as a flag that a preemption is coming, and does not use the value to decide whether or not to yield.

The time between warning and the actual preemption can be determined by the kernel scheduler. Presumably the scheduler is preempting for a reason, and there is a policy governing how long before a core is preempted so that it can be allocated to its provisionee. The `preempt_pending` field and the optional event can be used in conjunction with the kernel scheduler to implement a "two-minute warning" policy, as in Psyche. The current kernel scheduler does not care and immediately preempts cores.

As a reminder, there are two primary shared memory structures involved with preemption, per virtual core. The struct vcore is read-only, in procinfo. The preemption data (VCPD) is read-write, in procdata. Slightly edited versions of the structs are show below:

```
struct vcore {
    uint32_t            pcoreid;
    bool                valid;
    uint32_t            nr_preempts_sent;
    uint32_t            nr_preempts_done;
    uint64_t            preempt_pending;
};

struct preempt_data {
    struct user_context         vcore_ctx;
    struct ancillary_state      preempt_anc;
    struct user_context         uthread_ctx;
    uintptr_t                   transition_stack;
    uintptr_t                   vcore_tls_desc;
    atomic_t                    flags;
    int                         rflags;
    bool                        notif_disabled;
    bool                        notif_pending;
    struct event_mbox           ev_mbox_public;
    struct event_mbox           ev_mbox_private;
};
```

After sending warnings to userspace, the kernel finally revokes, unmaps, and changes the list membership of the vcore. All vcore structs are on one of three per-process linked lists. The vcores themselves sit in the vcoremap in procinfo (an array of vcore structs); they are not dynamically allocated. A vcore is on exactly one of three lists: *online* (mapped and running vcores), *bulk_preempt* (was online when the process was bulk preempted), and *inactive* (yielded, singly-preempted, or has not started yet). During preemption, the kernel moves vcores from the online list to either the inactive or the bulk-preempted lists, depending on the overall nature of the preemption. Bulk preemption is used when all vcores for a process are preempted at once: a time-slice of a spatial partition. When the process's `proc_lock` is released, either the online list or the bulk_preempt list should be empty.

To actually revoke a core, the kernel sets the `VC_K_LOCK` flag in the vcore's preemption data structure in procdata to signify the start of the current preemption, increments the `nr_preempts_sent` field, and sends the `__preempt()` kernel message to the physical core. The kernel-message handler saves the currently running user context into the appropriate slot in the VCPD: in `vcore_ctx` if notifications were disabled, and in `uthread_ctx` otherwise. The kernel only cares about the `notif_disabled` flag when it determines if a vcore was in vcore context or not. The kernel does not care about what stack or TLS the context uses — those values only make sense for userspace. If a uthread disables notifications, then as far as the kernel (and user code on other cores) is concerned, that core is in vcore context. Typically, the kernel also saves floating point state. Since a preemption is involuntary, userspace may be in the middle of instructions that rely on the FPU's state, unlike in when uthread yield and create software contexts, as discussed in Section 4.4.

Note that when the kernel message interrupts the physical core, the core might have been executing kernel code, either due to another interrupt or a system call. The current user context is not necessarily the one interrupted by the IPI; the kernel looks to the per-cpu variable `current_ctx` for the value of the user's context. There could be many nested kernel interrupt contexts, but the `__preempt()` handler does not care — its job is to save the user's context.

After the kernel saves the user context, it sets the `VC_PREEMPTED` flag and clears the `VC_K_LOCK` flag. The purpose of `VC_K_LOCK` is to prevent userspace from concurrently accessing fields that the handler updates, such as the uthread context. Without this flag, userspace could start copying the uthread context field while the kernel is still loading it, resulting in a corrupted uthread context. There are similar races with the FPU state. Both of these cases are discussed further in Section 6.3 below.

`VC_PREEMPTED` deserves special mention, since its meaning and usage is subtle. The `__preempt()` handler sets the flag, and the flag is cleared in `__startcore()`. Userspace reads the flag, but does not modify it (though it can), even after handling preemption. The purpose of the `VC_PREEMPTED` flag is to tell us that a vcore is offline, due to preemption, and that it has not come back online again. Userspace uses this flag in two ways: determining if a vcore was preempted and needs recovering, and making sure a vcore has not come back online yet. There are various situations with preemption recovery where a remote vcore wants to conditionally perform recovery only if a vcore has not already restarted.

It is possible for `VC_PREEMPTED` to be set, but for the preemption to have been handled by userspace. In short, Akaros's preemption handler can recover from some preemption scenarios solely through shared memory without actually starting up the vcore again. If a vcore *needs* to be handled after a preemption, then `VC_PREEMPTED` is set. The converse is not true; `VC_PREEMPTED` could be set, but the vcore already had been recovered — spurious signals are safe. If `VC_PREEMPTED` is not set, then we know the vcore is not preempted, and handlers can simply abort. `VC_PREEMPTED` is a subset of being offline; it is possible for a vcore to be preempted, restart, and then yield, at which point it is offline but not `VC_PREEMPTED`.

At the end of the `__preempt()` handler, the kernel increments the `nr_preempts_done` field. The two fields, `nr_preempts_sent` and `nr_preempts_done`, are used by the kernel to maintain an ordering between the process management kernel messages. The process management kernel messages, such as `__startcore()` and `__preempt()`, are sent while holding the `proc_lock` and based on the existing state of the process. For instance, the kernel will not send a `__startcore()` if a vcore is already online. Likewise, a `__preempt()` will be sent to an online vcore. The ordering on the sending of messages is managed by the kernel when it holds the lock. However, there is no ordering guarantee between the *execution* of the kernel messages on separate physical cores. As a reminder, kernel messages of the same type sent to a particular physical core will execute in order. The `nr_preempts_sent` and `nr_preempts_done` fields are used to establish an ordering between these messages sent to separate cores.

To see potential issues with message ordering, consider a process with a vcore, say vcore 6, running on physical core 3. The kernel sends a `__preempt()` to pcore 3, then immediately sends a `__startcore()` for vcore 6 to physical core 4. These are the steps a kernel scheduler would perform to migrate a vcore to another pcore. The `__preempt()` copies the running context into the VCPD, and `__startcore()` uses the context from the VCPD to restart the vcore. Thus, the `__startcore()` must not begin to execute until the `__preempt()` completes.

At a first glance, the previous scenario can be prevented by having `__startcore()` spin, waiting until the `VC_K_LOCK` field is clear. Aside the vulnerability associated with that flag being in user-writable space, a simple flag is insufficient. Consider a `__startcore()` for vcore 6 on pcore 3, quickly followed by a `__preempt()` to pcore 3 and another `__startcore()` for vcore 6 to a new pcore 4. The `__preempt()` executes after the first `__startcore()` on pcore 3, due to the in-order delivery guarantee of kernel messages. However, the `__startcore()` on pcore 4 could execute before or concurrent with the `__startcore()` on pcore 3, which could result in various races and corrupted contexts.

The ordering of messages provided by `nr_preempts_sent` and `nr_preempts_done` prevents these and other scenarios. Specifically, when the kernel sends a `__startcore()`, it also sends the current `nr_preempts_sent`. The `__startcore()` handler will spin until `nr_preempts_sent == nr_preempts_done`, ensuring the `__startcore()` happens after the previous `__preempt()`, regardless of which physical core the new `__startcore()` is sent to. Likewise, the `__preempt()` messages are always handled after the `__startcore()` which they are preempting, since kernel messages sent to the same physical core execute in the order delivered. Thus all of the `__startcore()` and `__preempt()` messages for a process execute in order. As a side note, messages like `__notify()` are not ordered; spurious messages are designed to be safe.

It is tempting to use `nr_preempts_sent` and `nr_preempts_done` as a signal to userspace that preemptions are in progress, instead of using the `VC_K_LOCK` flag in the VCPD. The reason userspace does not do so is because it actually performs a compare-and-swap on other flags, so long as the preemption is not in progress. This technique is a slight variant of "CAS-ing with the Kernel," first described in Section 4.4. The difference is that with the VCPD flags, the kernel does not actually care about the flag's value, and the `VC_K_LOCK` is designed to prevent userspace's compare-and-swap from succeeding. The central requirement of needing to change a flag conditioned on the absence of another flag remains the same, hence the requirement that `VC_K_LOCK` shares the same atomically changeable memory as the other VCPD flags. The main point is that `nr_preempts_sent` and `VC_K_LOCK` may appear similar, but serve separate purposes.

Finally, after the kernel preempts a vcore, it sends an event to the process, assuming the process has registered an event queue. The exact time of delivery depends on whether or not the process is being *bulk preempted*. Bulk preemption applies when all online vcores are preempted at the same time: a time-slicing of the spatial partition. For a typical, non-bulk preemption, the kernel immediately sends an event. For bulk-preempted vcores, the kernel delays sending the events until the process has at least one vcore online again. When the kernel allocates new cores to process that was bulk preempted, the first vcores that start up are taken from the bulk preempted list. Once the kernel has started all of the cores given to the process, it will send preemption messages for any vcores remaining on the bulk preempted list, and move those vcores to the inactive list. The purpose of bulk preemption is to minimize the amount of events sent to the process. In the future, Akaros may have an event handling mechanism where MCPs can run short handlers on LL cores, meaning they can handle events when they have no physical cores. In this case, after a bulk preemption, the kernel will send a single event to alert the process that it was time-sliced.

Changing Vcores: sys_change_vcore()

The kernel's role in preemption is merely to preempt and to tell the process a vcore was preempted. Userspace recovers from preemption. It is possible, but not guaranteed, that the kernel will grant new cores before userspace gets a chance to recover. One of the tools for preemption recovery is a syscall, `sys_change_vcore()`, that stops the calling vcore and starts another vcore on the same physical core. It is a kernel-assisted context-switch. For example, vcore 6 on pcore 3 could change to vcore 4, still on pcore 3.

It may seem counter-intuitive that the kernel is necessary for a context switch, given that one of the purposed of the MCP is fast user-level context switches. The distinction is that the kernel is switching *vcores*, and the user switches *threads*. The kernel does not know about threads, but it does know about vcores and must update the vcoremap, vcore lists, and other state.

There are two arguments to `sys_change_vcore()`: the vcore to change to and a flag, specifying whether or not the calling vcore needs to be restarted or not. If the flag, called `enable_my_notif`, is set, then the calling vcores `notif_disabled` flag will be clear. The next time the vcore starts up, the kernel will start a fresh vcore at the `_start` entry point at the top of the vcore's stack. In essence, the context switch is one-way: the calling context will be destroyed. If the `enable_my_notif` argument is clear, the calling vcore wants to be restarted where it left off, which is a more traditional context switch. In this case, `sys_change_vcore()` is similar to Mach's Handoff Scheduling[11] and Google's `switch_to`[108] primitives.

A critical part of `sys_change_vcore()` is notifying the process that the calling vcore has gone offline. It may seem unnecessary to do so, since userspace initiated the call. The critical detail is that `sys_change_vcore()` can fail, and when it succeeds, the caller will not run again for a while. One obvious failure case is if the target vcore is already online. Thus detecting the success of the syscall is not trivial from userspace.

The type of completion event that the kernel sends differs based on the `enable_my_notif` flag. When clear, which is when the calling vcore expects to have its context resumed in the future, the kernel sends a preemption event. In this case, the calling vcore is preempted, and another vcore will need to handle it. One way to look at it is that the kernel preempted a vcore, then the process uses `sys_change_vcore()` so that a *different* vcore is preempted.

When `enable_my_notif` is set, meaning the calling context will not run again, the kernel sends an `EV_CHECK_MESSAGES` event. This event instructs the process to check the public mailbox of the calling vcore. Sending this event ensures that no messages are lost and eases the event handling code. Recall from Section 5.3 that the kernel will deliver messages to an online vcore's public mailbox as part of its guarantee that the process will receive "spammed" messages. The kernel may have successfully delivered a message to a vcore's mailbox. In that mailbox, there could be an event or other action that triggers a `sys_change_vcore()`, before the vcore gets to the spammed message. Since the vcore never returns, it would never handle the message that was successfully delivered. One option to avoid this scenario would be to handle all messages before executing the `sys_change_vcore()`, but that is extremely unwieldy — that strategy places unnecessary limitations on the use of the syscall. The easy alternative is for the kernel to send an `EV_CHECK_MESSAGES`, which is sort of like an indirection event, so that userspace handles the vcores mailbox and receives the message whose delivery was previously guaranteed.

The handler for `EV_CHECK_MESSAGES` actually has two jobs: handle the messages and turn off `VC_CAN_RCV_MSG` for the *remote* vcore. Recall from Section 5.3 that when the vcore's `VC_CAN_RCV_MSG` flag is set, the vcore is a valid target for guaranteed message delivery (unless `EVENT_VCORE_MUST_RUN` is set). `VC_CAN_RCV_MSG` should not be modified by remote cores if the target vcore is online. For this reason, the handler conditionally clears `VC_CAN_RCV_MSG` only if `VC_PREEMPTED` with a compare-and-swap.

There are a few other technical details about `sys_change_vcore()`. For one, it is a mix of a `__preempt()` and a `__startcore()`, though it runs from syscall context — it is not a kernel message. It has similar issues as `__startcore()` regarding the ordering of operations. Internally, `sys_change_vcore()` will abort if the calling vcore has a concurrent preemption and performs the bulk of its `__startcore()` operations in a kernel message sent to itself, similar to using self-IPIs in other operating systems. Thus the ordering of the critical parts of `sys_change_vcore()` happen according to the same ordering rules as `__preempt()` and `__startcore()`, as discussed above.

The `sys_change_vcore()` syscall is not a complete solution to preemption; it is useful tool used by higher-level preemption handling functions. It is designed to be callable from vcore context in any situation. The exact details of the situation determine whether or not it is safe to set `enable_my_notif`. In the upcoming sections, I will describe the various preemption handling functions and how they use `sys_change_vcore()`.

6.2 Preemption Detection and Recovery Locks

One of the major concerns with revoking cores is that the kernel preempts a lock-holder. Ideally, parallel programs would avoid locking when possible, but there will be scenarios where a spinlock is the best or easiest approach to a problem. Many user-level schedulers and runtimes attempted to handle preempted lock-holders, and Akaros is no different. I outlined many of these systems and their shortcomings in Section 2.4.

In this section, I will describe Akaros's approach to preempted lock-holders, called *PDR* locks (Preemption Detection and Recovery). I have developed two variants of the PDR lock: the *Spin-PDR* lock and the *MCS-PDR* lock. My primary goal is correctness; since vcores may stop running permanently, a preempted lock-holder must not deadlock the system. Additionally, PDR locks should not be significantly more expensive than their non-PDR counterparts. PDR locks must be able to handle nested locking — some systems from prior work could not handle nested locks, which are quite common in programs. Finally, PDR locks should not require any algorithm-specific support from the kernel. There are a variety of locking strategies, and one size does not fit all. I do not want to build a particular locking algorithm into the kernel interface.

The guiding principles behind PDR locks are to use the idle time of contexts spinning on a locked lock to ensure that the lock-holder is not preempted, and that lock-holders must atomically grab a lock and advertise themselves as the lock-holder. The performance impact is relatively low, since we use time that would be otherwise spent busy waiting on a locked byte. PDR locks only attempt to recover from preemption when a lock-holder is preempted, which is the critical component of deadlock. More specifically, if the process has only one vcore, the preempted lock-holder could be waiting on the spinning context to yield, which is a circular wait condition. Recovering from preemption in more generic scenarios is the responsibility of a higher-level function, described in Section 6.3. There are a few requirements for PDR locks:

- Userspace needs to be able to detect if another vcore is preempted — the `VC_PREEMPTED` flag, exposed through shared memory.
- There must be a primitive to change to the lock-holder: `sys_change_vcore()`.
- The contexts attempting to grab a lock must be able to identify the lock-holder or otherwise find a way to make the lock-holder run. Typically, this requires atomically grabbing the lock and advertising itself as the lock-holder.

The requirements for PDR locking are not specific to a particular locking algorithm or even an operating system. Akaros provides the ability to detect preemptions and change to a preempted vcore, which suffices. Other systems may be able to provide the same with suitable modifications. More importantly, PDR locks provide guidelines for converting locking strategies into preemption-safe versions, if possible. The key requirement for locking algorithms is to find some way to atomically grab the lock and provide the spinning contexts a way to identify the lock-holder.

On Akaros, PDR locks are used when in vcore context, with the vcoreid used as the identifying variable for the lock-holder. Uthreads are able to use PDR locks by first pinning to a vcore and disabling notifications, as described previously in Section 4.4. Once a uthread has disabled notifications, it appears to be in vcore context to the kernel and user code running on other cores. At that point, it can use a PDR lock. All PDR locks in Akaros provide helpers to manage this setup so that they can be called from any context.

Spin-PDR Locks

As a reminder, spinlocks typically involve an atomic swap operation on a small amount of atomically addressable memory. The lock is a word of memory and has an unlocked value, say 0, and a locked value, say 1. Multiple threads try to atomically swap a 1 to the lock, and whichever thread receives a 0 from the swap succeeded in grabbing the lock. To unlock, the lock-holder simply writes a 0 to the lock variable. As an optimization, before attempting to swap the 1 for a 0, threads can first read the lock to make sure it is a 0. This cuts down on cache line write traffic.

The most basic PDR lock is the Spin-PDR lock. The Spin-PDR lock is like a spin lock; its unlocked value is -1 and its locked value is the vcoreid of the lock-holder. A simple swap is insufficient, since a failed lock attempt would incorrectly overwrite a previous lock-holder value. However, compare-and-swap does the trick:

```
void __spin_pdr_lock(struct spin_pdr_lock *pdr_lock)
{
    uint32_t lock_val;
    do {
        while ((lock_val = pdr_lock->lock) != SPINPDR_UNLOCKED) {
            ensure_vcore_runs(lock_val);
            cmb(); /* force re-read of lock_val */
        }
    } while (!atomic_cas_u32(&pdr_lock->lock, lock_val, vcore_id()));
    cmb();
}

void __spin_pdr_unlock(struct spin_pdr_lock *pdr_lock)
{
    mb();
    pdr_lock->lock = SPINPDR_UNLOCKED;
}
```

The function `ensure_vcore_runs()` will check if the vcoreid stored in `lock_val` is preempted, and if so it will perform a `sys_change_vcore()`. When changing vcores, the `enable_my_notif` flag is cleared; the calling vcore expects to run again from where it left off. The PDR lock code knows nothing about the caller and must assume that it requires a full context switch. For instance, the code could already hold another lock, and if it did not return, then the system would deadlock.

There are a variety of potential outcomes from `ensure_vcore_runs()`. The common case is that the vcore is not preempted, and we just spin like in a normal spinlock. If the vcore was preempted and there are several other vcores checking, then potentially all of them will attempt to change to the lock-holder. The kernel enforces that only one vcore will succeed in the `sys_change_vcore()` call; the others will fail and start the loop again — this time the lock-holder is online.

A slightly nastier situation can result in the vcore attempting to grab the lock to change to a vcore that is no longer the lock-holder and possibly not even online. Once the caller determines that the lock-holder is preempted, the caller could be preempted or otherwise delayed long enough for the lock-holder to restart, unlock, and do just about anything. If that vcore is offline for any reason, such as a preemption or yielding, the caller will `sys_change_vcore()` to the target. Although this scenario is tricky, it is safe. A vcore can `sys_change_vcore()` to any other vcore at any time without loss of correctness. The target vcore, or perhaps another vcore, will notice the caller was preempted and deal with the preemption accordingly. In this manner, spurious calls to `sys_change_vcore()` are similar to cooperative yielding or time-slicing frequently. This aspect of `sys_change_vcore()` is yet another example of the Akaros principle of making spurious events safe. It would be much more difficult to design a PDR lock where the caller atomically changes to the lock-holder only if they still hold the lock.

MCS-PDR Locks

The general principle of PDR locking can be applied to MCS locks, though the corner cases are more difficult. The one issue MCS-PDR locks need to overcome is that callers are not guaranteed to know the vcoreid of the lock-holder. They are, however, guaranteed to know the vcoreid of the vcore ahead of them in the queue, and by induction, the set of spinning vcores knows the lock-holder. In general, MCS-PDR locks use a refined version of the requirement to identify the lock-holder: vcores spinning must be able to identify another vcore to change to, so that the spinning does not result in a deadlock.

As a refresher, a basic MCS lock is a linked list of nodes, one per thread. Each node is the size (and alignment) of a cache line and has a lock byte that a single thread busy waits on. The lock is passed down the list, in order, with a minimum of cache line contention.

The basic MCS lock is built around a qnode structure:

```
struct mcs_lock_qnode
{
    struct mcs_lock_qnode *next;
    int locked;
}__attribute__((aligned(ARCH_CL_SIZE)));

struct mcs_lock
{
    struct mcs_lock_qnode *lock;
};
```

Basic MCS locks in Akaros have the following lock and unlock methods:

```
void mcs_lock_lock(struct mcs_lock *lock, struct mcs_lock_qnode *qnode)
{
    qnode->next = 0;
    cmb();
    mcs_lock_qnode_t *predecessor = atomic_swap(&lock->lock, qnode);
    if (predecessor) {
        qnode->locked = 1;
        wmb();
        predecessor->next = qnode;
        while (qnode->locked)
            cpu_relax();
    }
    cmb();
}

void mcs_lock_unlock(struct mcs_lock *lock, struct mcs_lock_qnode *qnode)
{
    if (qnode->next == 0) {
        cmb();
        if (atomic_cas_ptr((void**)&lock->lock, qnode, 0))
            return;
        while (qnode->next == 0)
            cpu_relax();
        qnode->next->locked = 0;
    } else {
        mb();
        qnode->next->locked = 0;
    }
}
```

The algorithm involves threads atomically swapping to put themselves at the end of the MCS queue. If they were at the front of the line, they hold the locks; otherwise they tell the predecessor who they are, and spin until the predecessor unlocks. The predecessor, when it unlocks, tries to remove itself from the list, conditioned on it being the only one in the queue. If it fails, it must wait on the its successor to advertise itself before unlocking. There are also variants of the MCS unlock that do not require compare-and-swap; see Mellor-Crummey and Scott's original work[69] for more details. Note that in the MCS locking algorithm, when the lock-holder unlocks, it will busy wait on an unknown thread; this is a slight challenge for the PDR locks. I deal with it in the same way as the threads attempting to grab the lock: ensure someone else runs, who will eventually ensure the target runs.

MCS locks are particularly susceptible to preemption, due to the FIFO nature of the queue. Preemption interferes with spinlocks only when a lock-holder is preempted; preemption of a thread *attempting* to grab a lock has little effect on other participants. If a thread in an MCS list is preempted, other threads behind it in the queue will be delayed. The reason is that a thread in an MCS list can be given the lock passively, while the thread is offline. Many systems tried to customize their preemption handling scenarios to get around the MCS queue, as I discussed in Section 2.4. The MCS-PDR locks do not suffer from any of the drawbacks of those systems.

I developed and tinkered with a few versions of MCS-PDR locks. First I will present the original version, called the MCS-PDRO lock. The original version suffers under extreme load in some cases, leading me to develop a safer, slightly more expensive variant, which I will present later.

The basic structures of the PDRO lock are the same as the MCS lock, with the addition of the vcoreid to the qnode:

```
struct mcs_pdro_qnode
{
    struct mcs_pdro_qnode *next;
    int locked;
    uint32_t vcoreid;
}__attribute__((aligned(ARCH_CL_SIZE)));

struct mcs_pdro_lock
{
    struct mcs_pdro_qnode *lock;
};
```

All PDR locks are wrapped with calls to `uth_disable_notifs()`, which ensure that the calling context drops into vcore context, as far as any external observers are concerned. The helper also pins the uthread to its vcore, which allows uthread code to call `vcore_id()` and save it in the qnode. By saving the vcoreid in the qnode, any thread with a pointer to a qnode can identify the vcore owning that qnode.

```

void mcs_pdro_lock(struct mcs_pdro_lock *lock, struct mcs_pdro_qnode *qnode)
{
    uth_disable_notifs();
    qnode->vcoreid = vcore_id();
    __mcs_pdro_lock(lock, qnode);
}

void mcs_pdro_unlock(struct mcs_pdro_lock *lock, struct mcs_pdro_qnode *qnode)
{
    __mcs_pdro_unlock(lock, qnode);
    uth_enable_notifs();
}

void __mcs_pdro_lock(struct mcs_pdro_lock *lock, struct mcs_pdro_qnode *qnode)
{
    struct mcs_pdro_qnode *predecessor;
    uint32_t pred_vcoreid;
    qnode->next = 0;
    cmb();
    predecessor = atomic_swap_ptr((void**)&lock->lock, qnode);
    if (predecessor) {
        qnode->locked = 1;
        pred_vcoreid = ACCESS_ONCE(predecessor->vcoreid);
        wmb();
        predecessor->next = qnode;
        while (qnode->locked) {
            ensure_vcore_runs(pred_vcoreid);
            cpu_relax();
        }
    }
}

void __mcs_pdro_unlock(struct mcs_pdro_lock *lock,
35                            struct mcs_pdro_qnode *qnode)
36 {
37     uint32_t a_tail_vcoreid;
38     if (qnode->next == 0) {
39         cmb();
40         if (atomic_cas_ptr((void**)&lock->lock, qnode, 0))
41             return;
42         a_tail_vcoreid = ACCESS_ONCE(lock->lock->vcoreid);
43         while (qnode->next == 0) {
44             ensure_vcore_runs(a_tail_vcoreid);
45             cpu_relax();
46         }
47         qnode->next->locked = 0;
48     } else {
49         mb();
50         qnode->next->locked = 0;
51     }
52 }
```

As with all PDR locks, the MCS-PDRO locks center around the busy waiting `while` loops. These are the places where the code can deadlock, as well as the source for idle cycles to make sure the lock-holder is not preempted. Unlike spinlocks, MCS locks have two places where spinning occurs: when waiting for the lock (line 27) and when handing off the lock to the next thread (line 43).

Threads spinning while waiting for the lock do not know the identity of the lock-holder. However, they do know the identity of their predecessor. Each predecessor makes sure its predecessor runs, up to the actual lock-holder. The trick is that threads atomically enter the lock and determine their predecessor, instead of the lock-holder. The predecessor is sufficient to make sure the lock-holder runs.

When waiting to hand off the lock to the next thread, the current lock-holder does not know who its successor will be. However, it does know the identity of the *tail* of the MCS queue: this is the last qnode, stored in the `lock` variable (line 42). The successor to the current lock-holder is at the head of the MCS queue. By ensuring the tail runs, which will ensure its predecessor runs and so on, eventually we ensure that the successor runs.

The way I have described MCS-PDR locks, it sounds like all of the preempted vcores would percolate to the tail of the MCS queue. Picture ten vcores in a queue and five of them are preempted. Eventually, the lock-holder and the next four vcores in line will be online, and the last five would be offline. This is the end result of the algorithm, but it only happens if the lock is held for a long time, compared to the time it takes to perform a `sys_change_vcore()`.

However, when a lock is held for a short duration, all of the vcores quickly lock, unlock, then reenter the queue behind the offline vcores — all in the time it takes to perform a syscall. Imagine a preempted vcore in the middle of the queue, and it is the only preempted vcore in the system. By the time its successor changes to it, all of the vcores ahead of it have grabbed and released the lock, and the lock has been passed to the preempted vcore for quite a long time. The `sys_change_vcore()` calls are slowly moving the preemption "bubbles" towards the tail of the queue, but the locking and unlocking can add qnodes to the end of the queue at a higher rate. The end result is that based on the time the lock is held and the think time between lock acquisition, a preemption bubble will stay in the MCS queue. It will not be the same vcore each time; that will change as the vcores cycle through the system. Ultimately, at least one syscall is made to hand off the lock — hardly ideal.

The preempt storm I describe happens under particular circumstances, but is easily shown with a microbenchmark (Section 6.2). It is unclear whether or not this scenario is realistic or not; all vcores need to grab and quickly release the lock. Regardless, to explore the issue further, I developed another version of the MCS-PDR lock. The next variant of the MCS-PDR lock stores the lock-holder vcoreid in the lock structure. Most of the other differences between the MCS-PDR locks follow from this addition. The first to note is that the qnodes are stored in the lock, instead of on the stack:

```
struct mcs_pdr_qnode
{
    struct mcs_pdr_qnode *next;
    int locked;
}__attribute__((aligned(ARCH_CL_SIZE)));

struct mcs_pdr_lock
{
    struct mcs_pdr_qnode *lock __attribute__((aligned(ARCH_CL_SIZE)));
    uint32_t lockholder_vcoreid __attribute__((aligned(ARCH_CL_SIZE)));
    struct mcs_pdr_qnode *qnodes __attribute__((aligned(ARCH_CL_SIZE)));
};
```

The qnodes look the same as for normal MCS locks, but the lock itself stores the qnodes. At lock initialization time, the `qnodes` array is malloced. The important part is that given a qnode, we must be able to determine the vcoreid. The newer MCS-PDR locks use pointer arithmetic on the qnode array to do so. By contrast, the older MCS-PDRO locks store the vcoreid in the qnode struct.

```
void mcs_pdr_lock(struct mcs_pdr_lock *lock)
{
    uth_disable_notifs();
    __mcs_pdr_lock(lock, &lock->qnodes[vcore_id()]);
}

void mcs_pdr_unlock(struct mcs_pdr_lock *lock)
{
    __mcs_pdr_unlock(lock, &lock->qnodes[vcore_id()]);
    uth_enable_notifs();
}

```

```
void __mcs_pdr_lock(struct mcs_pdr_lock *lock, struct mcs_pdr_qnode *qnode)
{
    struct mcs_pdr_qnode *predecessor;
    uint32_t pred_vcoreid;
    struct mcs_pdr_qnode *qnode0 = qnode - vcore_id();
    seq_ctr_t seq;
    qnode->next = 0;
    cmb();
    predecessor = atomic_swap_ptr((void**)&lock->lock, qnode);
    if (predecessor) {
        qnode->locked = 1;
        pred_vcoreid = predecessor - qnode0;
        wmb();
        predecessor->next = qnode;
        seq = ACCESS_ONCE(__procinfo.coremap_seqctr);
        while (qnode->locked) {
            if (vcore_is_preempted(pred_vcoreid) ||
                seq != __procinfo.coremap_seqctr) {
                if (lock->lockholder_vcoreid == MCSPDR_NO_LOCKHOLDER ||
                    lock->lockholder_vcoreid == vcore_id())
                    ensure_vcore_runs(pred_vcoreid);
                else
                    ensure_vcore_runs(lock->lockholder_vcoreid);
            }
            cpu_relax();
        }
    } else {
        lock->lockholder_vcoreid = vcore_id();
    }
}

void __mcs_pdr_unlock(struct mcs_pdr_lock *lock, struct mcs_pdr_qnode *qnode)
{
    uint32_t a_tail_vcoreid;
    struct mcs_pdr_qnode *qnode0 = qnode - vcore_id();
    if (qnode->next == 0) {
        cmb();
        if (atomic_cas_ptr((void**)&lock->lock, qnode, 0)) {
            lock->lockholder_vcoreid = MCSPDR_NO_LOCKHOLDER;
            return;
        }
        a_tail_vcoreid = lock->lock - qnode0;
        while (qnode->next == 0) {
            ensure_vcore_runs(a_tail_vcoreid);
            cpu_relax();
        }
        lock->lockholder_vcoreid = qnode->next - qnode0;
        wmb();
        qnode->next->locked = 0;
    } else {
```

```
63            lock->lockholder_vcoreid = qnode->next - qnode0;
64            mb();
65            qnode->next->locked = 0;
66        }
67    }
```

The newer MCS-PDR lock is fairly similar to the MCS-PDRO lock. The main additions are the `lockholder_vcoreid` and extra logic in the busy waiting loops.

When waiting for the lock, the use of the `lockholder_vcoreid` variable is an optimization — it is not guaranteed to be set. It is possible that a lock-holder gets preempted after acquiring the lock (line 22), but before setting `lockholder_vcoreid` (line 40). If the lock-holder is known, the spinner can just ensure the lock-holder runs (line 35). If the lock-holder is not known, the spinner ensures its predecessor runs.

There is a brief period where `lockholder_vcoreid` points to the successor, but the successor is still locked (between lines 59 and 61). It is possible the lock-holder was preempted at line 60, but all the other vcores will think the successor should run. However, the successor will see that it is the `lockholder_vcoreid` and know that means its predecessor should run (line 32).

The `lockholder_vcoreid` is not needed for correctness; it is a performance optimization. For correctness, the only thing needed in the `while` loop at line 28 is that we ensure the predecessor runs (line 33). The main benefit of the lock-holder optimization comes under heavy contention for a shortly-held lock.

To break out of the `sys_change_vcore()` storm, which was the motivation for the newer MCS-PDR locks, we need a vcore to handle the preemption of the vcores in the "bubble". If vcores only spin and check their predecessor, as in MCS-PDRO locks, then it is extremely unlikely that an event handler will deal with the preemption first — polling is simply faster. By using the `lockholder_vcoreid`, vcores do not spin on their predecessor unless they need to, which increases the chances that another vcore will deal with the preempted vcore. Typically, this is the lock-holder after it unlocks, re-enables notifications, and checks messages.

Tracking the `lockholder_vcoreid` is not particularly expensive. There is an extra cache line write. Having the lock-holder write `lockholder_vcoreid` on behalf of the successor helps a little. A final tidbit about the `lockholder_vcoreid` is that there is a race when a lock-holder unlocks (line 50) and sets the `lockholder_vcoreid` to none (line 51). In between those lines, a new lock-holder could have taken over and advertised itself. This situation is not a problem; if vcores see there is no `lockholder_vcoreid`, they fall back to using the predecessor.

Although *tracking* the `lockholder_vcoreid` is not very expensive, having every spinning thread *polling* the `lockholder_vcoreid` variable is expensive. It is a heavily contended cache line that gets written on every lock handoff. To cut down on the reads to `lockholder_vcoreid`, the spinners only try to recover when either its predecessor is preempted (line 29) or there has been a preemption since it started spinning (line 30). When there are no preemptions, the checks at lines 29 and 30 are reads of shared cache lines.

A final discussion point, mentioned earlier, is where to put the qnode structures. The MCS-PDRO locks and the basic MCS locks takes a pointer to a qnode, and the caller allocates the qnode on the stack before calling the lock method. The benefit of each thread providing their own qnode struct is that the number of qnodes in the system grows naturally with the number of threads. The benefit of storing the qnode on the stack are that the memory is likely to be in the cache of the caller.

For the newer MCS-PDR locks, I opted to store an array of qnodes in the actual lock, one for each vcore. The array wastes memory, but since uthreads pin themselves to vcores, the lock's memory only grows for every vcore, not every uthread. Otherwise, it would be infeasible to store per-thread qnodes inside the lock. There are two benefits to using the qnode array that are particular to MCS-PDR locks.

First, given a qnode pointer, we need to determine the vcoreid of that qnode. In the stack-based qnode style, I stored the vcoreid in the qnode. Reading the vcoreid from the qnode required a memory reference. With the array of qnodes, we can determine the vcoreid with pointer arithmetic (lines 24 and 59).

The second reason has little to do with performance; all MCS-PDR locks must be careful when dereferencing qnode pointers. If the vcoreid is stored in the qnode, then when a thread attempts to read the qnode of its predecessor to determine the predecessor's vcoreid, there is a chance that the predecessor has already unlocked. If it unlocked, the qnode memory could have been freed, and the thread's read could page fault. It is actually safe for the memory to be filled with garbage; the worst case is the locking thread ensures an arbitrary vcore is not preempted. The original rule I used for MCS-PDR locks was that the memory used for the qnodes cannot be freed during the lifetime of the lock. This is not completely unreasonable for stack-based locks — the vcore stacks are never freed, for instance. Regardless, it was a little simpler to put the qnodes in the lock.

Ultimately, it is unclear at this time which variant of MCS-PDR locks is better. They both have their trade-offs, as we will see below.

Locking Microbenchmarks

In this section, I compare the performance of various locking algorithms. The tests were run on the machine "c89", which is an Intel Xeon E5-2670, 2.6GHz, with two processors, each with 16 hyper-threaded cores. The test is a microbenchmark named `lock_test`, which has various options, such as the number of threads, the number of lock acquisitions per thread, and type of lock. It runs on both Akaros and Linux, and for every lock acquisition, the benchmark records timestamps before attempting to lock, after grabbing the lock, and after unlocking the lock.

First, let us look at the family of spin locks. I adjusted the amount of loops for each run so that the test ran for at least 20 msec. All measurements are in TSC ticks, unless otherwise stated. The TSC frequency of c89 is 2.59 ticks per nanosecond. The overhead for reading the TSC is 32 ticks. The primary metric of interest is the lock acquisition time, which is the $time\_after\_acquire - time\_before\_acquire - TSC\_overhead$.

Table 6.1 shows spinlock acquisition times, measure in ticks, for basic spinlocks on Linux and Akaros, as well as Akaros's Spin-PDR locks. The tests with one thread show the locking overhead for uncontended locks. Four threads shows the performance under light contention, and 31 threads shows the performance under heavy contention. For the four-thread runs, all threads are pinned to the same socket. The 31-thread runs involve all cores across two sockets. These do not show any particular effects of preemption; the systems are unloaded and no special steps were taken to take away cores.

Table 6.1: Spinlock Acquisition Latency (in ticks)

						Quantiles				
		Avg.	**Std.**	**Min**	**Max**	**50%**	**75%**	**90%**	**99%**	**99.9%**
Linux	**1 Thr**	**32.85**	96.3	24	42780	28	28	32	268	336
	4 Thr	1853	**12245**	24	844184	364	440	1092	**56543**	**167629**
	31 Thr	19058	**154476**	28	8989312	528	1520	2700	**592223**	**2309883**
Akaros	**1 Thr**	**32.50**	50.8	28	20676	28	28	32	36	508
	4 Thr	1000	**42317**	28	5244320	108	488	1148	2792	4853
	31 Thr	30301	**135596**	28	2834032	1272	2584	11581	706436	1600882
PDR	**1 Thr**	**46.06**	47.9	40	20668	44	44	44	52	500
	4 Thr	995	**2980**	40	74216	296	500	1344	15932	33624
	31 Thr	**40301**	**55451**	40	807172	18320	58276	111792	249555	389737

The first thing to note is that contended spinlocks are terrible for tail latency, on any operating system. The standard deviation, maximum, and large quantiles are several orders of magnitude greater than the median. Figure 6.1 provides a look at the distribution of the lock latency for regular spinlocks on Akaros, with four threads. For the x-axis, I zoomed in enough to see the graphs features — the extreme outliers dominate the tail otherwise.

Spinlocks are inherently unfair; the winner of the lock depends on the caching system. A nice way to see this is by analyzing the timestamps of each lock acquisition. Figure 6.2 plots a dot every time a thread grabbed the lock. Ideally, the lines would be solid. The gaps represent long stretches of time when a thread was unable to win the race to grab the lock. The solid lines represent numerous acquisitions within a short time frame — enough that we cannot see a gap. This effect is not abysmal for an application. Presumably when *any* thread grabs a lock, application work is performed. However, unlucky threads can be delayed for long stretches of time.

Akaros's uncontended spinlocks perform slightly better than Linux's, especially at the 99%. This result is likely a proxy for the FTQ results from Chapter 4 — there is more interference on Linux, with a slight effect on this microbenchmark.

The performance of the PDR locks are of more interest than the regular spinlocks. Spin-PDR locks are about 16 cycles more expensive than their regular spin-based counterparts, which is nearly 50%. Most of this cost is due to dropping into vcore context (pinning and disabling notifications), which involves a function call, various increments, and a branch or

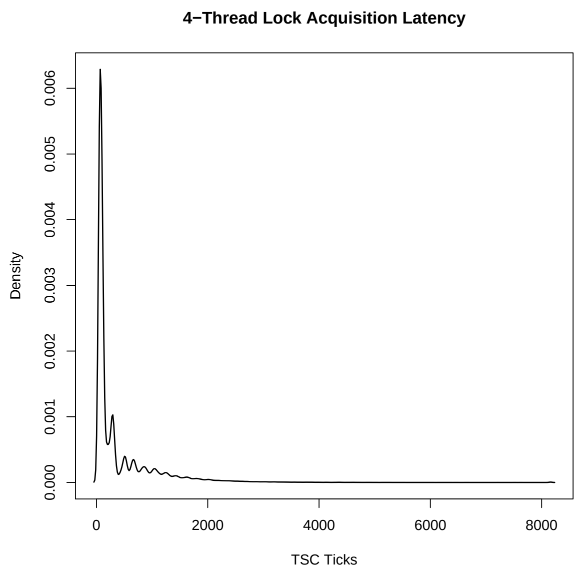

Figure 6.1: Akaros Spinlocks, 4 Threads, Acquisition Latency

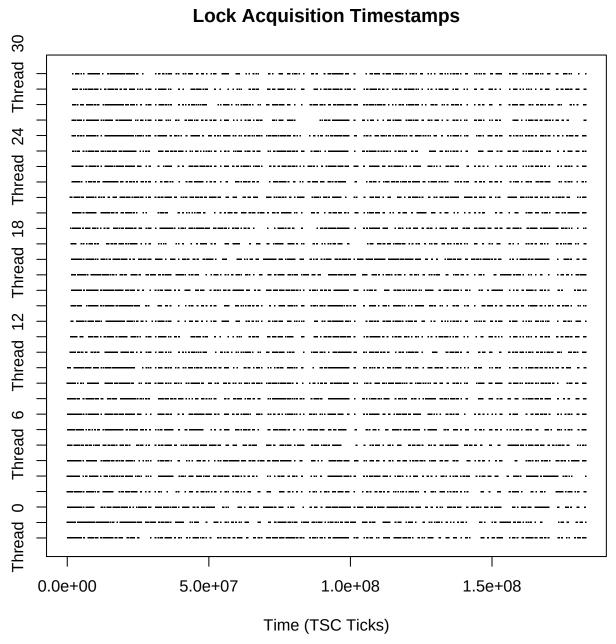

Figure 6.2: Linux Spinlocks, 31 Threads, Acquisition Times

two. When instructing the benchmark to run from vcore context, a one-thread PDR lock takes 33 cycles on average, with similar quantiles. Incidentally, the lock's *held* time is greatly increased, since enabling notifications is more expensive than disabling them.

Another performance issue may be due to using compare-and-swap instead of a simple atomic swap. Note the Spin-PDR's *average* performance with 31 threads; it is three orders of magnitude higher than a single thread. This is the case regardless of whether the test fakes vcore context or not. Spin-PDR locks are basically the same as regular "test, test and set" spinlocks, with the addition of ensuring the target vcore runs, and using compare-and-swap instead of atomic swap. Just for testing, removing the call to `ensure_vcore_runs()` made little difference to the lock's performance, leaving compare-and-swap as a likely culprit. Regardless, spinlocks are the wrong tool for a heavily contended lock with a large number of threads.

Spin-PDR locks are reasonably fast at low scale, and they can recover from preemption. Of course, nothing is free — the ability to recover from preemption comes at a cost. Let us look at MCS locks and see how the different variations compare in Table 6.2. When analyzing MCS locks, keep in mind that they are queue locks, and the lock acquisition time naturally increases as a factor of the number of threads in the system. Once a thread joins the end of the queue, it must wait for $O(n)$ locks and releases until it acquires the lock.

Table 6.2: MCS Lock Acquisition Latency (in ticks)

						Quantiles				
		Avg.	**Std.**	**Min**	**Max**	**50%**	**75%**	**90%**	**99%**	**99.9%**
Linux	**1 Thr**	**38.46**	92.3	32	21876	32	36	36	272	372
	4 Thr	919.4	273.5	32	61488	908	920	936	1288	1984
	31 Thr	16303	430868	32	81945846	12572	13764	14940	16632	**32699**
Akaros	**1 Thr**	**35.46**	27.2	32	2272	32	36	36	36	300
	4 Thr	841.0	88.0	32	20916	832	864	888	944	1408
	31 Thr	10718	831	36	20592	10576	11236	11864	13000	14388
PDRO	**1 Thr**	**52.36**	66.8	40	20196	48	48	52	52	600
	4 Thr	836.5	134.1	56	17716	848	896	944	1140	1648
	31 Thr	16520	1585	64	30268	16488	17452	18756	19836	20328
PDRN	**1 Thr**	**58.4**	58.6	48	19388	56	56	56	56	548
	4 Thr	870.6	113.4	48	21268	864	920	976	1124	1480
	31 Thr	17595	2491	364	37924	17504	19264	20880	23768	25692

As expected, basic MCS locks on Akaros perform better than on Linux in all categories, due to the lack of interference. Linux's MCS locks have a particularly long tail. The PDR locks, as expected, are more expensive in the uncontended (one thread) case. PDR locks add a few extra instructions or cache line fetches to the common case. For functions that are relatively small and simple like locks, those additions affect the overall performance. These extra overheads are the price of preemption detection and recovery. Under a light load (four threads), the PDR overhead is lost in the noise.

Figure 6.3 shows the approximate distribution of lock latency for all four locks with 31 threads. The long tails are not visible, since the density is so low, so I zoomed in the graph on the main body of the distributions. This graph tells a similar story as Table 6.2. One additional note is that the new MCS-PDR locks (PDRN) have a very normal looking distribution. This feature has been a constant for the PDRN locks throughout numerous test runs.

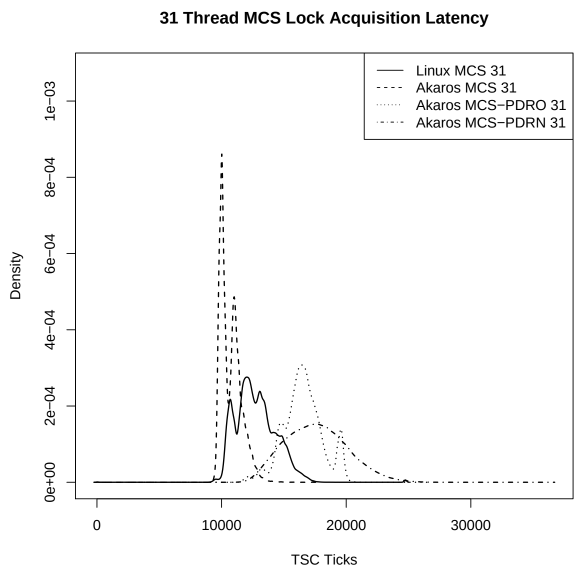

Figure 6.3: MCS locks, 31 Threads, Acquisition Times

Despite the regularity of the PDRN locks, they have a greater overhead than the PDRO locks. As described earlier, PDRO locks suffered under heavy lock contention when vcores were preempted. To test the various locks under preemption, I ran the lock benchmark while periodically preempting two vcores for 10 milliseconds at a time.

The most effective way to see the impact of preemption is to look at the lock throughput. Figure 6.4 shows the basic MCS lock's throughput, under preemption. It is no surprise that the lock throughput drops to 0 when a vcore is preempted. If the preempted vcore was in the MCS queue, no other core can make progress once the preempted vcore is granted the lock. Much like with barriers, the entire system waits on one thread that is not running. Any system using MCS locks must be concerned with this sort of interference, which MCS-PDR locks are designed to avert.

What is more interesting is Figure 6.5, which is the behavior of the PDRO lock while being preempted. The throughput drops very low, to around 250 locks per msec. The system

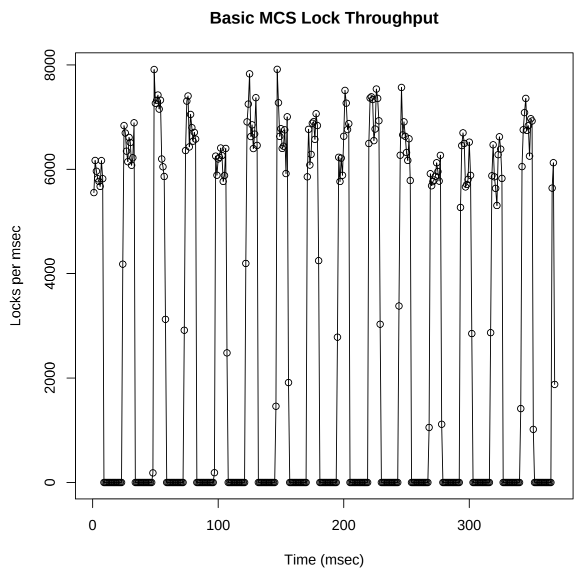

Figure 6.4: Akaros, Basic MCS locks, 31 Threads, Lock Throughput

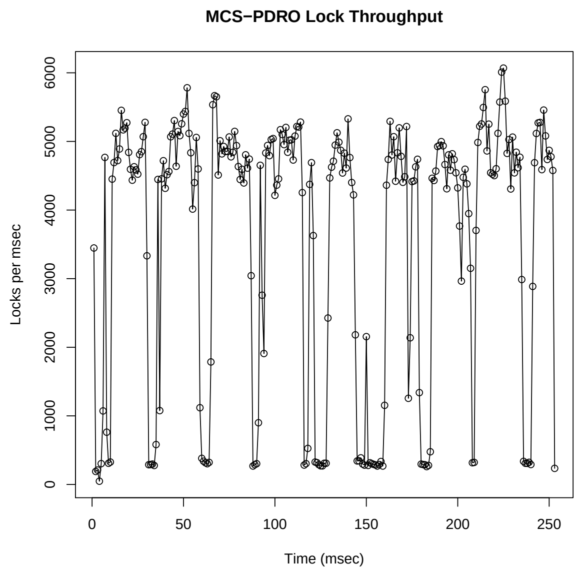

Figure 6.5: Akaros, MCS-PDRO locks, 31 Threads, Lock Throughput

is making progress, albeit at a slow rate. This anomaly is the "preemption storm" described above, which was the motivation for the PDRN locks, shown in Figure 6.6. Under heavy

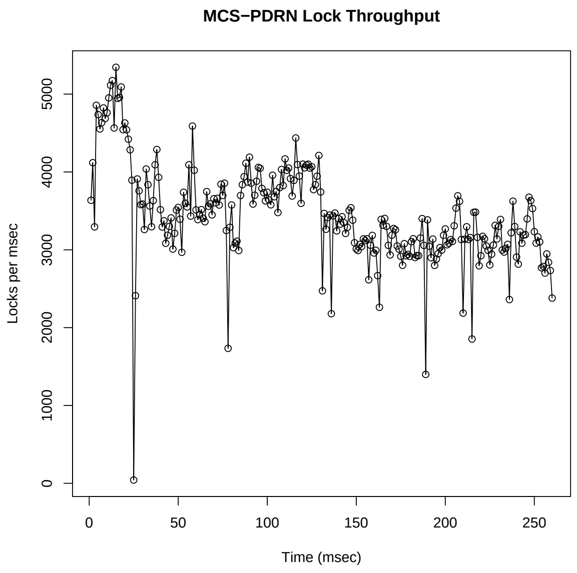

Figure 6.6: Akaros, MCS-PDRN locks, 31 Threads, Lock Throughput

load, the PDRO locks are unable to fully deal with preemption and stabilize on a lower number of vcores, and each lock handoff involves at least one syscall. If we run `lock_test` with PDRO locks, preempt one vcore, and never allocate another core to the benchmark, the PDRO locks are very unlikely to recover.

Another way to analyze the anomaly is by looking at the number of `EV_VCORE_PREEMPT` messages sent. Every call to `sys_change_vcore()` triggers either a `EV_VCORE_PREEMPT` or a `EV_CHECK_MSGS` event. In the case of PDR locks, the `sys_change_vcore()` calls will trigger a `EV_VCORE_PREEMPT` event. The benchmark instruments the event handlers for these messages by registering extra event handlers, which track the count of events. The PDRN locks had 420 preempt messages, which includes both the `sys_change_vcore()` calls and the original preemption message. The PDRO locks had 20,893 preempt messages — a factor of 50 more than PDRN.

Ultimately, both variants of the MCS-PDR lock, as well as the Spin-PDR lock, enable userspace to recover from the preemption of a lock-holder. It is up to the programmer to choose the appropriate lock for a given workload. For lightly-contended locks, Spin-PDR locks are the simplest and most effective choice. For more heavily-contended locks, the choice is between MCS-PDRO and the MCS-PDRN locks. The current Akaros libraries default to the PDRN lock, though the correct choice depends on both the lock contention and the expected amounts of preemption. Finally, in extreme situations, an application could use the non-PDR locks, so long as that application is never preempted. If an application will be run only with provisioned cores, then at compile time the developer can choose the faster locks, with the understanding that the program will need to be recompiled to be safe from preemption.

Custom Synchronization

Application and library writers can use the basics of PDR locks to build their own synchronization primitives. Just recall one of the vcore context rules: never busy wait on user-level code without handling preemption in some manner. Otherwise, there is a chance of deadlock.

Certain synchronization algorithms are not good fits for vanilla PDR locking. For instance, Ticket Locks[22] are a FIFO spinlock, where each thread increments a counter to join the locking group, similar to taking a number at a bakery. When the lock-holder unlocks, it increments the "next" counter. Both counters are stored in the same, atomically addressable memory word. Unfortunately, there is no obvious way for threads to atomically join the group and announce their vcoreid.

One approach that would usually work well for Ticket Locks and any other locking protocol would be to lock like normal, and add a `lockholder_vcoreid` field. Lockers set the field immediately after locking and before unlocking. Of course, there is a slight window where the variable is unassigned — although rare, it must be dealt with. In these cases, the spinning code can check *every* vcore to make sure it is not preempted. The performance could be lousy and result in excessive `sys_change_vcore()` calls, but given the rarity of the event the overhead might not be a problem.

To make busy waiting easier for user code, the Parlib library has a few helpers for making sure other cores are not preempted. The PDR locks use `ensure_vc_runs(int vcoreid)`, which changes to the target vcore if it is preempted. If the vcoreid is the vcoreid of the caller, it will ensure all *other* vcores are not preempted, from 0 to `max_vcores()`. Eventually, it will recover and change to the lock-holder. Keep in mind that `ensure_vc_runs()` checks the `VC_PREEMPTED` flag, so while the scan is $O(n)$, it will not usually need to change to each vcore.

Blindly calling `ensure_vc_runs()` on the entire process is not a great idea when the lock-holder is not preempted. Even the $O(n)$ scan takes time, and that could be time wasted while holding a lock. Unless there is a strong suspicion that a target is preempted, the `cpu_relax_vc()` helper is more appropriate. It will only ensure all other vcores run every 1000 calls, which is a common technique for busy waiting algorithms: spin for a bit before taking more advanced steps.

Ultimately, any user-level code that busy waits must have a plan for dealing with preemption. Most applications and schedulers will use a PDR lock from the Parlib library. The goal of these PDR locks was to avoid deadlock with a minimum of overhead. PDR locks do not solve the overall problem of a preempted vcore. Consider an application that simply uses PDR locks, with ten uthreads on ten vcores, and half of those vcores get preempted. At some point, we want to get back to normal processing and not rely on `sys_change_vcore()`. Spinlocks are a tool, and PDR locks make sure that tool does not break during preemption. Fully recovering from preemption will require more steps.

6.3 Handling Preemption

The Uthread library provides routines to handle the preemption of vcores. Virtually all applications that use uthreads will use these event handlers.

Broad Plan

The broad plan for handling preemption is quite simple:

- If the preempted vcore was in uthread context, remotely copy out the uthread's context, hand it to the 2LS, and check the vcore's mailbox.
- If the preempted vcore was in vcore context, we have no choice but to change to it. Halt *our* current uthread, hand it to the 2LS, and change to the preempted vcore.

For the former case, when the vcore was executing a uthread, preemption is fairly simple since the code that was running does not interact with scheduler code. Continuing the analogy between vcore context and interrupt context in operating systems, recovering from a preempted uthread is like an IRQ handler interrupting a syscall: relatively simple. We need to check the vcore's mailbox in case the preempted vcore was the destination of a spammed message that was guaranteed to be delivered. Given that most applications perform their work in uthread context, most preemptions will happen in this case. The common case will involve no system calls and minimal interaction between the user and the kernel. Since the uthread's context and the vcore's floating point state are in shared memory (the VCPD), the handler can copy the context without assistance.

For the latter case, the preempted vcore could have been in the middle of a delicate operation and must continue. As a simple example, a 2LS could have been about to run a uthread, and if we do not run that vcore, then that uthread would be lost. Attempting to recover a preempted vcore context is like an NMI (non-maskable interrupt) handler interrupting arbitrary kernel code: be very careful. I opted to not bother with attempting to deduce what the vcore was doing and just context switched to it (`sys_change_vcore()`).

The initial problem with simply calling `sys_change_vcore()`) is that the caller sacrifices itself and then it is the preempted vcore. The trick to preventing a perpetual ping-ponging is that before changing vcores, the *caller* prepares itself for preemption: it packages up its uthread, hands it to the 2LS, and takes steps to make sure its messages are read. Then, the vcore handling the preemption can change to the preempted vcore with `enable_my_notifs` set, meaning that it does not need to run ever again.

In a sense, this approach to preemption delays the work done in response to a warning until *after* a preemption occurs. Consider Psyche's two-minute warning: presumably a process, when informed of impending preemption, must finalize its work, prepare to be preempted, and perhaps yield its core to the operating system. The problem with the warnings is that the process may not respond in time, or at all. In Akaros, this preparation is done after the preemption and on a different vcore — we get the time from the future, on another vcore, without delaying the preemption. That being said, it is still useful to tell the application in advance about upcoming preemptions, but it is not necessary.

When viewed in this light, the approach to handling preemption for vcores that were in vcore context is sufficient for handling *all* preemption scenarios. Simply perform the “preparation” for the handling vcore to be preempted, then change to the preempted vcore, regardless of its state. The former case, where a uthread is preempted, is an optimization for the common case.

Details

The heavy lifting in preemption handling is performed by an event handler and its helper functions.

```
static void handle_vc_preempt(struct event_msg *ev_msg, unsigned int ev_type,
                              void *data)
{
    uint32_t vcoreid = vcore_id();
    struct preempt_data *vcpd = vcpd_of(vcoreid);
    uint32_t rem_vcoreid = ev_msg->ev_arg2;
    struct preempt_data *rem_vcpd = vcpd_of(rem_vcoreid);
    struct uthread *uthread_to_steal = 0;
    struct uthread **rem_cur_uth;

    if (rem_vcoreid == vcoreid)
        return;
    while (atomic_read(&rem_vcpd->flags) & VC_K_LOCK)
        cpu_relax();
    if (!(atomic_read(&rem_vcpd->flags) & VC_PREEMPTED))
        return;
    if (rem_vcpd->notif_disabled) {
        change_to_vcore(vcpd, rem_vcoreid);
        return;
    }
    if (!start_uth_stealing(rem_vcpd))
        return;
    if (!(atomic_read(&rem_vcpd->flags) & VC_PREEMPTED))
        goto out_stealing;
    if (rem_vcpd->notif_disabled)
        goto out_stealing;
    rem_cur_uth = get_cur_uth_addr(rem_vcoreid);
    uthread_to_steal = *rem_cur_uth;
    if (uthread_to_steal) {
        if (uthread_to_steal->flags & UTHREAD_PINNED) {
            stop_uth_stealing(rem_vcpd);
            change_to_vcore(vcpd, rem_vcoreid);
            return;
        } else {
            *rem_cur_uth = 0;
36                __uthread_pause(rem_vcpd, uthread_to_steal, FALSE);
37                wmb();
38            }
39        }
40  out_stealing:
41        stop_uth_stealing(rem_vcpd);
42        handle_indirs(rem_vcoreid);
43  }
```

There are a few complications with preemption handling that roughly fit into three categories:

- Concurrency: the preemption handler can be run by multiple vcores, while the kernel is still performing the preemption, or while the preempted vcore restarts.
- Difficulties with `current_uthread`: there are some cases where a vcore cannot package its state to change to another vcore.
- Missing messages: we must not miss messages that were guaranteed to be delivered to the application.

Concurrent Operations

Preemption handling must wait until a preemption is complete. The kernel sets `VC_K_LOCK` when the preemption starts and clears it after the vcore's context is saved into the VCPD. The handler spins until the kernel unlocks (line 14). Note the handler is busy waiting without using `cpu_relax_vc()`; this is fine since the code waits for the kernel, not for userspace.

There could be multiple vcores handling the preemption for the same vcore concurrently. Preemption messages are spammed to make sure some vcore handles them; multiple copies of the message must be safe. The synchronization between vcores handling the event happens in two places. The first is in the kernel when there are competing `sys_change_vcore()` calls — the kernel ensures only one of them wins. The second is in fighting to set a VCPD flag claiming the right to steal a uthread (line 22). In `start_uth_stealing()`, threads compete to set `VC_UTHREAD_STEALING` in the preempted vcore's VCPD. If the kernel is mucking with the flags (`VC_K_LOCK` is set), userspace spins. If the flag is already set, it returns failure. Failure means the handler can simply abort (line 23); another vcore is handling the preemption.

Competing vcores are not the only ones who we need to worry about: the target vcore could start back up. As discussed above, if `VC_PREEMPTED` is set, then the target vcore has not come back online. A handler cannot access or modify another vcore's `current_uthread` while that vcore is online. `VC_UTHREAD_STEALING` is used to prevent *any* accesses to the vcore's uthread — from other vcores or from the vcore itself. In the early stages of `vcore_entry()`, which is the first code run when a vcore starts, if `VC_UTHREAD_STEALING` is set, the vcore will busy wait until it is cleared (checking its mailbox for preemption messages in the meantime). Of course, once a vcore has passed the `VC_UTHREAD_STEALING` check, nothing will stop it from modifying its `current_uthread`. The handling vcore can detect this state since the target vcore will have its notifications disabled, which is the defining feature of vcore context.

As a side note, older versions of Akaros would restart a vcore exactly where it left off. If it was running a uthread, then that uthread would be started directly by the kernel. The problem with that approach is there is no way for a vcore to steal another vcore's uthread safely, since the preempted vcore's uthread would start directly, without the chance to check `VC_UTHREAD_STEALING`.

In short, the handler needs to set `VC_UTHREAD_STEALING` so long as the target vcore is preempted and not in vcore context. It can aggressively set the flag, so long as the flag is written before the checks are performed. Note the use of the "check (lines 16 and 18), signal (line 22), check again (lines 24 and 26)" pattern.

There are a few other esoteric race conditions involved here. Imagine a vcore that was preempted in uthread context. The handler gets to line 21, then the vcore starts up and restarts its uthread. Then the handler sets `VC_UTHREAD_STEALING` at line 22. The check at line 26 will fail, since the target vcore is no longer in vcore context. The check at line 24 would notice, and prevent the handler from stealing a running uthread, whose context is invariable different than what is in VCPD. The check at line 26 is necessary if a vcore was preempted once in uthread context so that the handler could get past line 18, then the vcore starts up and gets preempted again, this time in vcore context.

Difficulties with `current_uthread`

Concurrent operations from other vcores can be solved with shared memory synchronization. Another class of problems arises from situations where we cannot package up the uthread, either on the preempted vcore or the handling vcore. The two sources of these problems are `VC_UTHREAD_STEALING` and `UTHREAD_PINNED`.

Recall from Section 4.4 that uthread's have a flag, `UTHREAD_PINNED`, that means the uthread is not allowed to leave its current vcore. This flag is not set lightly; it is only used when a uthread requires that it stays on a vcore, usually as part of the process of disabling notifications. Without the flag, a uthread cannot be sure of its vcoreid, and thus unable to disable its vcore's notifications. During the brief moment after the uthread is pinned, but before notifications are disabled, we treat the vcore as if it was in vcore context since it will be in vcore context very soon. This special treatment for pinned uthreads applies when starting uthreads, as well as during preemption. When starting pinned uthreads, the Uthread scheduler code temporarily violates the rule that we never leave vcore context when there is a `notif_pending`. It will pop to uthread context, ignoring `notif_pending`, because any code that clears `UTHREAD_PINNED` must check `notif_pending` and reenter vcore context if it was set. In essence, pinning a uthread acts like a wrapper around vcore context.

For the purposes of handling preemption, if a vcore has a pinned uthread, then it is stuck in vcore context and cannot package up its state to yield completely. This statement is true for both the preempted vcore and for the vcore handling the preemption. If a preempted vcore had a pinned uthread, we must change to it (line 33). If the *handling* vcore has a pinned uthread, then it needs to be careful if it must change to the preempted vcore. To explain further, we need to look at the helper `change_to_vcore()`.

```
44 static void change_to_vcore(struct preempt_data *vcpd, uint32_t rem_vcoreid)
45 {
46     bool were_handling_remotes;
47     if (atomic_read(&vcpd->flags) & VC_UTHREAD_STEALING) {
48         __change_vcore(rem_vcoreid, FALSE); /* returns on success */
49         return;
50     }
51     cmb();
52     if (!current_uthread) {
53         were_handling_remotes = ev_might_not_return();
54         __change_vcore(rem_vcoreid, TRUE);  /* noreturn on success */
55         goto out_we_returned;
56     }
57     if (current_uthread->flags & UTHREAD_PINNED) {
58         __change_vcore(rem_vcoreid, FALSE); /* returns on success */
59         return;
60     }
61     assert(current_uthread);
62     __uthread_pause(vcpd, current_uthread, TRUE);
63     current_uthread = 0;
64     were_handling_remotes = ev_might_not_return();
65     __change_vcore(rem_vcoreid, TRUE);      /* noreturn on success */
66 out_we_returned:
67     ev_we_returned(were_handling_remotes);
68 }
```

Whenever a handler determines that it must change to another vcore (lines 19 and 33 in `handle_vc_preempt()`), it calls `change_to_vcore()`. Ideally, the handler would completely clean up its state (lines 63-65) and perform a "one-way" context switch (line 66). A one-way context switch calls `sys_change_vcore()` with an argument of `TRUE` and will never return. If the handling vcore has a pinned uthread, then it must return from its `change_to_vcore()` (line 59). Likewise, if the handling vcore's `current_uthread` is being stolen, then it cannot clean up its state since it cannot touch its `current_uthread`.

The overall plan to handle preemption relies on the handling vcore cleaning up its state, and then changing to the preempted vcore. The two cases preventing the clean-up are `VC_UTHREAD_STEALING` and `UTHREAD_PINNED`. One concern is that through a pathological series of events, multiple vcores could be stuck and unable to make progress due to a stealing in progress or a pinned uthread. Perhaps two vcores, both with pinned uthreads, change back and forth to each other. The preemption handler and the scheduler entry function work together so that these livelocks will not occur. When analyzing these scenarios, note that all vcores will be in vcore context. It is possible that a handler may think another vcore was in uthread context, but in any such deadlock, all vcores will eventually agree that all of them are in vcore context.

First, a vcore will never set another vcore's `VC_UTHREAD_STEALING` and then itself call `change_to_vcore()`. So for multiple vcores to be stuck in a cycle of stealing each other's uthreads, the kernel would need to preempt a vcore that was in the middle of a handler. There is nothing wrong with this scenario. If a vcore was preempted in the middle of its handler and had set another vcore's `VC_UTHREAD_STEALING`, the next time it ran it would see the other vcore was in vcore context and clear `VC_UTHREAD_STEALING`, averting this deadlock.

Second, the only time a vcore handles an event while its own uthread is pinned is when another vcore was briefly stealing its uthread. The vcore does this because it is spinning, waiting on another vcore to stop stealing its uthread. In this case, the stealer will soon unset `VC_UTHREAD_STEALING`, either because it noticed we were in vcore context or because it stole the uthread. In any event, the next time the vcore ran with a pinned uthread, it would not check events right away. It would wait until the uthread became unpinned, at which point it would handle any outstanding preemption events — this time in a state where it can cleanly perform a one-way change to another vcore.

In general, vcore context code only handles events in clearly marked places. Vcore context is decoupled from event handling, unlike with both Scheduler Activations and Psyche. In Scheduler Activations, an activation was a vehicle for an event. In Psyche, software interrupts were executed just like hardware interrupts, with interrupt vectors and handlers. In Akaros, events are just one of many things doable in vcore context. By choosing when to handle events, we can avoid handling events at inopportune moments. Additionally, event handlers do not need to return — those one-way context switches to other vcores never return, and the vcore starts up with a fresh context the next time it runs. The flexibility of this model is one of the benefits of Akaros's vcore context over Scheduler Activations and Psyche.

The details behind `VC_UTHREAD_STEALING` and `UTHREAD_PINNED` can be a bit overwhelming. In fact, every issue caused by `VC_UTHREAD_STEALING` can be removed if I change the preemption handling system to not attempt to copy uthreads remotely. I built the remote recovery to see if and how it could be done, since it seemed possible with shared memory. Perhaps profiling data from real-world use cases will show a negligible improvements from remote recovery, and that the complexity in the code is not worth the hassle. The main point is that there are some occasions where we cannot blindly change to another vcore, but the system is designed to handle them.

Never Miss a Message

The final complications stem from guaranteed event delivery. If a vcore is online or has its `VC_CAN_RCV_MSG` set, it is a valid destination for a spammed message, as discussed in Section 5.3. Part of the preemption handler's job is to turn off this flag in a preempted vcore, then check the public mailbox for messages. That process is fairly straightforward — the only complication involves using compare-and-swap to change the flag only if the preempted vcore is still preempted. The major difficulty associated with missing messages involves the *handler's* vcore, due to the scenarios where the handling vcore does a one-way context switch, never returning from the event handler.

There are three situations the preemption handler must consider:

- There are still messages in the handling vcore's public mailbox. Dealing with this is simple: when a `sys_change_vcore()` succeeds, the kernel sends an `EV_CHECK_MSGS` event. The uthread library set up an event queue with the `EVENT_SPAM_PUBLIC` and `EVENT_VCORE_MUST_RUN` flags set, so the kernel will ensure an online vcore will get the message. The vcore receiving the message will usually be the one we are changing to, which is guaranteed to be online. The handler for `EV_CHECK_MSGS` will call `handle_vcpd_mbox()`, which deals with the details of handling message from another vcore's public mailbox.

- The preemption event came from an `EVENT_INDIR` event queue that is throttled. This means that only one indir message was sent to userspace to look at the event queue, though there may be many events remaining in the queue. If the handler does not return, then the process will not check that event queue until the next message, which may never come. The rule for event queues is that if a handler might not return, such as the preemption handler, then any event queues with `EVENT_INDIR` must also have `EVENT_NO_THROTTLE`. Actually, the preemption event queue is not an indir — it uses spammed public messages, but this rule still applies to all other event queues.

- The handling vcore was in the process of handling a remote vcore's public mailbox. Similarly to throttled indir event queues, there may be only a single message sent to the process to handle another vcore's mailbox. If a handler does not return, then any other messages in that mailbox could be missed. The solution is for the handler to send a `EV_CHECK_MSGS` to its own public mailbox.

  Specifically, any event handler that might not return calls `ev_might_not_return()`, which works together with `handle_vcpd_mbox()` to send an `EV_CHECK_MSGS` only if the caller is processing a remote vcore's mailbox. Likewise, any time a vcore encounters a `EV_CHECK_MSGS` while processing another vcore's mbox, it sends that message to its own public mailbox. In essence, a vcore uses its public mailbox as a "to-do" list to get us out of trouble.

Final Thoughts on Preemption

Handling preemption is complicated, but the Uthread library manages its complexities. Several of those complications arise from the handler stealing uthreads, which may or may not be truly necessary.

From the perspective of second-level scheduler writers, the main impacts of preemption are:

- A call to `handle_events()` might not return. Plan accordingly.

- The 2LS upcall `thread_paused()` will usually be called when there is a preemption. This is the 2LS's opportunity to reevaluate scheduling or do whatever it wants. Note that the calling core might be about to change to another vcore, so `thread_paused()` must return and the 2LS must not assume the current vcore will continue to run.

Likewise, 2LS or application developers can override the uthread preemption handler if they choose, but only after reading this chapter. More likely, applications will use the existing handlers, and use the `thread_paused()` method to simply add uthreads to runnable lists. If future 2LSs have particular needs that are not suited by the existing upcalls, then I can easily extend the interface.

The main takeaway from preemption handling is that it can be done safely, without deadlocking, from any situation. The key insight is to prepare for preemption from the handler, *after* the original vcore was preempted, and then change to the preempted vcore. This technique, as well as many others used in preemption handling, was enabled by separating event handling from vcore context. In this manner, the kernel can preempt a core immediately from a process so that it can reallocate the physical core to a higher priority process.

Chapter 7

Applications

In the previous chapters, I described the mechanisms underlying the MCP, event delivery, and preemption handling. In this chapter, I will show how applications can utilize Akaros's mechanisms, including adapting to preemption and customizing their second-level schedulers. Finally, I will show the kernel preempting cores from a low-priority application to allocate them to a high-priority application with provisioned resources.

7.1 Fluidanimate

Given the FTQ results from Section 4.5, a user-level, compute-bound task will run better on Akaros than Linux. Arguably, FTQ is such a hyper-sensitive compute-bound task. This is not a mystery: the Akaros kernel does not interfere with a user's core, and the Linux kernel does. Of course, Akaros's processor isolation is not a free lunch: the cost is at least one core dedicated to handling interrupts, system management, and single-core processes. The result of this tradeoff is that a system with n cores can only dedicate $n - 1$ to an MCP.

Instead of testing a basic matrix multiplication kernel, I looked for an HPC application with more interesting characteristics to better explore the pros and cons of the Akaros model. Fluidanimate, a particle simulator from the PARSEC benchmarking suite[10], is one such application. Fluidanimate is a user-level, compute-bound task with a requirement of having a power-of-two number of workers. That requirement is due to how fluidanimate decomposes its problem; a power-of-two is very convenient for the application. The application computes in phases, with periodic barriers across all workers. Ideally, fluidanimate would run on a power-of-two number of physical cores, with each worker thread pinned to a particular core.

Of course, on Akaros, we cannot allocate all of the cores of a machine, so fluidanimate's performance on a machine with 32 cores should suffer. Likewise, on any operating system, we may not have a power-of-two number of cores available. The machine might not have a power-of-two number of cores at all. Additionally, there may be multiple applications on a machine, and fluidanimate might be allocated less cores than it desires. One simple tool to deal with the imbalance between the number of cores and the number of workers is the thread. With fast, user-level threading, an application can have as many worker threads as it likes, independent of the underlying parallelism available.

Table 7.1: Fluidanimate Runtime (seconds)

	32 Thr	**64 Thr**	**128 Thr**	**256 Thr**	**512 Thr**
Linux 32 Cores	**66.95**	89.91	133.66	170.18	221.00
Linux 31 Cores	75.80	93.56	143.52	179.28	226.77
Linux 24 Cores	89.07	107.70	153.18	181.91	234.16
Akaros 31 Cores	70.24	**68.55**	68.59	68.64	71.73
Akaros 24 Cores	95.95	**82.14**	84.17	84.68	88.98

Table 7.1 shows the runtime of fluidanimate on c89 for 1000 frames of the `in_500K.fluid` data set, varying the number of cores and pthreads. Increasing the thread count does not help fluidanimate's performance on Linux, regardless of the number of cores. In fact, the oversubscribed threads greatly interfere with each other; increasing the thread count by a factor of four decreases performance by a factor of two. For Akaros, 64 threads performs the best, and additional threads do not significantly hurt performance.

Overall, Akaros's best case (64 threads on 31 cores) is worse than Linux's best case (32 threads on 32 cores). However, the gap is quite narrow — only about 2%. Older, 32-bit versions of fluidanimate performed slightly better on Akaros by around the same margin, even with only 31 cores.

Fluidanimate on Akaros uses the default pthread scheduler, which is first-come, first-served (FCFS). Since there are more threads than cores on Akaros, each phase of fluidanimate's computation goes as follows. 31 threads run until they individually reach the next barrier — typically some threads finish faster than others due to how the particle simulation works. When a thread finishes, a remaining thread runs. On an unlucky phase, the entire application could be held up by one or two long-running threads that happened to run late in the phase. A smarter second-level scheduler, closely coupled to the application and that knows about a thread's progress, could quickly time-slice these threads so that they finish around the same time. Hofmeyr et al. pursued this idea with their Juggle scheduler[42], which proactively load balances threads in HPC applications. With close coordination between the scheduler and the application, as well as fast user-level context switches, a Juggle scheduler on Akaros should perform quite well.

As shown in Table 7.1, fluidanimate performs better on Akaros than Linux with 31 and 24 cores. In short, performance on Linux declines if the number of cores is not ideal, due to either kernel scheduling or interference. Due to Akaros's inexpensive user-level threads and synchronization, fluidanimate was able to spawn significantly more threads than cores and be more resilient to preemption.

To see this effect more clearly, I measured fluidanimate's throughput over time as cores were preempted. First, to set the baseline performance, Figure 7.1 shows fluidanimate's framerate over time with no interference on c89. Clearly, the throughput varies over time — this is simply an artifact of the application.

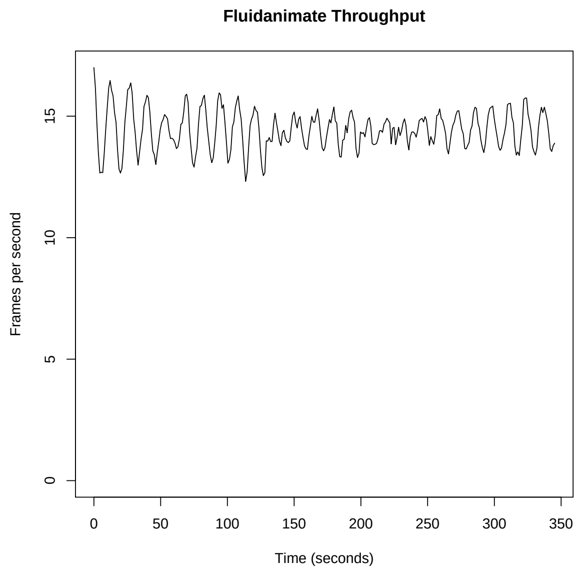

Figure 7.1: Fluidanimate, Akaros, 64 Threads, 31 Cores

On both Linux and Akaros, I ran fluidanimate for 5000 frames, which takes about 5 minutes under normal circumstances. Every ten seconds, I preempted a core. To preempt cores on Linux, I used `taskset`. To preempt cores on Akaros, I provisioned them to an idle process that simply halts the core. After ten seconds with one core, I gave all the of available cores to the application (32 on Linux, 31 on Akaros). Figure 7.2 shows the application's performance with preemption on Akaros and Linux. Figure 7.3 shows Akaros's performance *relative* to Linux. When the application has fewer cores than threads, fluidanimate's performance on Akaros is higher — enough so that the application finishes earlier. Greater resilience to changing amounts of resources is one of the features enabled by fast user-level threading. Also note that fluidanimate did not deadlock on Akaros; the default preemption handlers did their job.

In this test, the cause of preemption was synthetic: either an idle process or `taskset` directives to the kernel. In the next section, fluidanimate will run as a background application, where the machine is provisioned to a high-priority webserver.

7.2 Webserver

The MCP was designed with the desire to allow high-performance applications to manage their resources. The purpose of user-level scheduling is to allow an application to cus-

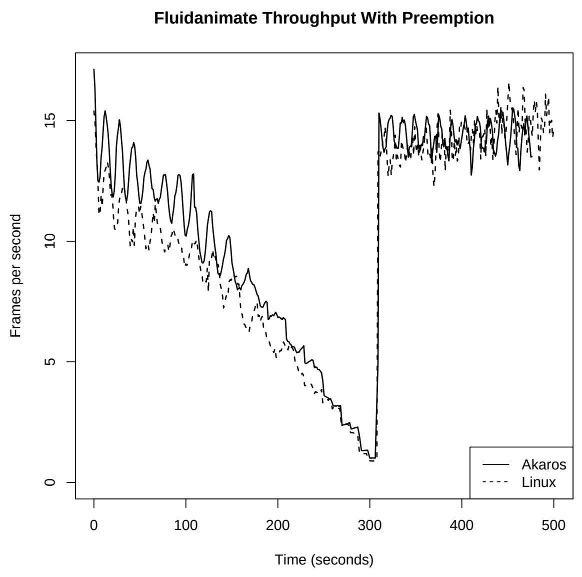

Figure 7.2: Fluidanimate Performance on Akaros and Linux with Preemption

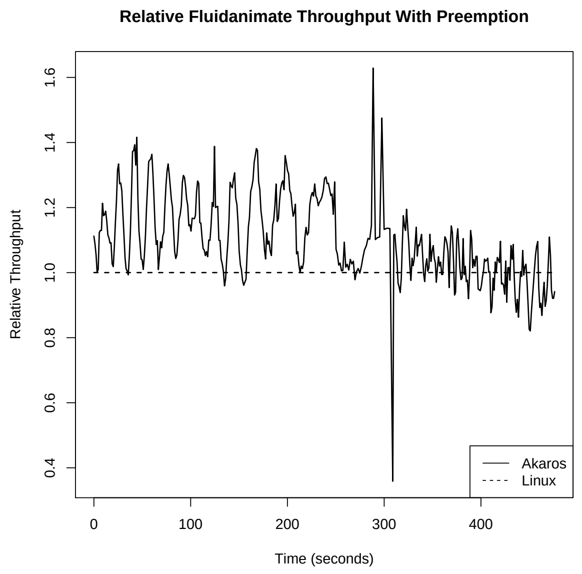

Figure 7.3: Fluidanimate Relative Performance on Akaros and Linux with Preemption

tomize its scheduler to maximize performance given its set of allocated resources. Likewise, Akaros's distinction between resource `provisioning` and `allocation` is driven by a desire to utilize idle resources while still satisfying a high-priority application's needs, as I discussed in Section 3.2. The distinction between `provisioning` and `allocation` only matters for applications with changing resource demands — otherwise the allocated amount is the provisioned amount. An application with a smart scheduler can manage its threads and adjust the amount of resources used based on its workload. In this section, I will discuss one such application, a webserver, and how it interacts with Akaros.

Kevin Klues wrote a basic, threaded webserver in C called *kweb*. It only serves static files and is easily modifiable. We ported kweb to use Akaros's pthread library.

One interesting feature of kweb is its handling of bursts of requests. HTTP persistent sessions batch several requests across a single connection to amortize the cost of establishing a TCP connection. These requests are referred to as a *burst*. Some webservers pursue a "thread per connection" strategy. By contrast, kweb's strategy is a "thread per request." By default, kweb threads dequeue connections from a global connection queue. The threads extract a request, then requeue the connection in the global queue, and finally processes the request. All threads writing responses via the same connection serialize on a per-connection mutex. Kweb's intra-burst concurrency enables certain scheduling policies and allows the system to stay busy if there are not enough connections, at the expense of complexity.

To measure kweb's performance, I used httperf[76] with slight modifications by Kevin and I to handle a large number of open file descriptors and to track various events. Httperf is capable of numerous types of tests with a wide array of options. Due to the relatively immature networking stack of Akaros, I used a throughput test with a limited number of connections. Httperf launches 100 connections, each of which sends bursts of 100 requests for a total of 100,000 requests per connection. The total number of requests is not particularly important — it is mostly a knob to control how long the test runs.

The benefit of this type of test compared to one where httperf creates numerous connections is that performance degrades gracefully under load. The Akaros networking stack has scalability problems with a large number of connections, which is beyond the scope of my dissertation. If kweb falls behind with a typical "connections per second" benchmark on Akaros, networking performance drops drastically. With a static set of connections, queueing within the network stack and the application layer provides back-pressure such that the system handles as many requests as it can without being overloaded. For this reason, I often refer to this test as a *throughput plateau* test, since it represents the plateau of an ideal throughput curve.

For a frame of reference, kweb on Linux with 50 threads handles around 165,600 queries per second (QPS) on c89 for the throughput plateau test. Kweb on both Linux and Akaros performed best on this benchmark with fifty threads. Likewise, throughput for both varied about the average. Table 7.2 shows the throughput for kweb on Linux and Akaros for a variety of settings, explained below.

Kweb on Linux did not noticeably benefit from any scheduling policies, such as restricting the application to a single socket or to only using one hyperthread in a pair of sibling cores. However, kweb on Akaros definitely benefits from restricting the webserver to the first socket and from using only one core out of each pair of hyperthreaded siblings.

Table 7.2: Kweb Throughput, 100 connections of 100,000 calls each

	Avg.	**Min.**	**Max.**	**Std.**
Linux	165558	159103	167703	2395
Akaros, Pthreads, 6 VCs, Hyperthreads	134676	133121	135567	635
Akaros, Pthreads, 6 VCs, No-Hype	162885	160488	166035	1661
Akaros, Custom 2LS, 6 Worker Cores	170315	168094	172045	766

The reason for the former is that Akaros's current networking stack uses physical core 0 to execute most of the networking stack, including the reception of all interrupts. By contrast, Linux routes interrupts to several cores. Ideally, Akaros would allow the application to request that certain IRQs from the network card are routed to its cores — that is a subject of future work. For now, the best performance on Akaros results from keeping the application on the same socket where the network stack executes, to share the last-level cache.

Kweb on Akaros also benefits from running on only one core in each pair of hyperthreaded siblings. The reason for this may be that the pthread scheduler, as well as all custom schedulers for kweb, poll for work in vcore context. If an application chooses to be nice and yield its unused resources, then that is the decision of the scheduler. For the highest and most predictable performance, typically I instruct the pthread scheduler to not yield its vcores for lack of a smarter policy. The difference in performance can be seen in Table 7.2. Akaros with pthreads and using hyperthreaded cores handled only 134676 QPS, compared to 162885 QPS when the kernel scheduler handed out non-hyperthreaded cores.

Additionally, the webserver performed best on Akaros with only six worker vcores, regardless of the second-level scheduler. At first, I thought this was due to the contention in user-space for the global run queue in the pthread scheduler. However, kweb's performance is actually heavily limited by the kernel's performance. Despite Akaros's relatively poor networking stack, there are still steps the 2LS can take to optimize its performance, as I discuss below.

Customizing the User-level Scheduler

As I often say, the problem with user-level scheduling is that someone needs to write a user-level scheduler. I wrote a primitive scheduler for kweb, called *wthreads*, and incorporated several optimizations, some of which improved performance while others harmed performance. Overall, with a customized 2LS, kweb on Akaros beat even kweb's performance on Linux, as shown in Table 7.2.

On Akaros, the biggest performance improvement comes from using specific physical cores, notably disabling hyperthreading and limiting the application to a single socket. For the pthreads version of kweb, I hacked the kernel scheduler to only hand out even-numbered cores so that all odd-numbered cores are halted — effectively disabling hyperthreading. This approach is not particularly manageable, so one of the customizations to the 2LS is to have the webserver turn off all of its odd-numbered cores. Putting the control of hyperthreading into the hands of the application has two benefits: the application can choose whether or not to do so, and the application is actually allocated the (halted) sibling cores. The latter benefit is that the application is billed for the use of those cores. It just so happens to want the cores to remain halted. Not all applications will want hyperthreading disabled — fluidanimate did not benefit from disabling hyperthreading.

The current Akaros kernel interface for requesting and allocating cores does not make a distinction between particular cores or between sibling pairs. The applications request is only for a quantity of cores. To allocate specific cores, such as a single socket, the user must provision those specific cores. The kernel will allocate provisioned cores before any others. In the future, the kernel will need to accept requests for specific cores, possibly a bitmap of desired cores. Whether or not the kernel interface should explicitly support sibling pairs is unclear. To minimize CPU interference, an MCP would need both siblings in a hyperthreaded pair. However, perhaps allocating siblings could be done with an explicit request for each core, instead of building a sibling pair abstraction.

Regardless, the application's scheduler will be involved in managing hyperthreading, regardless of the details of the kernel interface. The application can determine which physical cores it runs on by looking at the vcoremap in *procinfo*, a memory region shared with the kernel.

The webserver also runs best when limited to a single socket. Not only are interrupts routed strictly to physical core 0, but the networking stack does not scale well to large numbers of cores. When the webserver utilizes cores on the second socket, the kernel's locks and memory references force cross-socket cache misses, which are expensive.

Similarly, kweb's performance depends on the number of physical cores, as well as the location. Figure 7.4 shows the throughput of kweb as we increase the number of workers. For each worker vcore, kweb actually has a pair of siblings with the odd-numbered vcore halted. The application also has allocated and halted physical core 1, so as to limit interference to core 0. When the application has eight workers, it actually has physical cores 1-17, and cores 16 and 17 are on the second socket. Clearly the performance is worse in this situation. However, regardless of issues with multiple sockets, more cores is not necessarily better. Kweb performed better with six workers than seven, even though seven workers is still all in the first socket. The large variation of kweb's performance with seven workers is a sign of poor scalability.

I pursued several other improvements to the webserver's scheduler. One beneficial change was how the 2LS deals with blocking syscalls. Since the vcores busy wait in their idle loops, there is no reason to receive IPIs from the kernel when syscalls complete. Changing this was as simple as changing a few flags in the event queue that the 2LS uses when an asynchronous syscall is not complete.

Overall, I had high ambitions for a customized 2LS. The default pthread scheduler used a global runqueue and made no concessions to the nature of the webserver. I built a scheduler with per-vcore run queues, with the policy that all threads working on a given connection stay on the same vcore. Each vcore is responsible for many connection, and each connection has exactly one vcore. The intent of this policy is to scale to large numbers of cores by partitioning

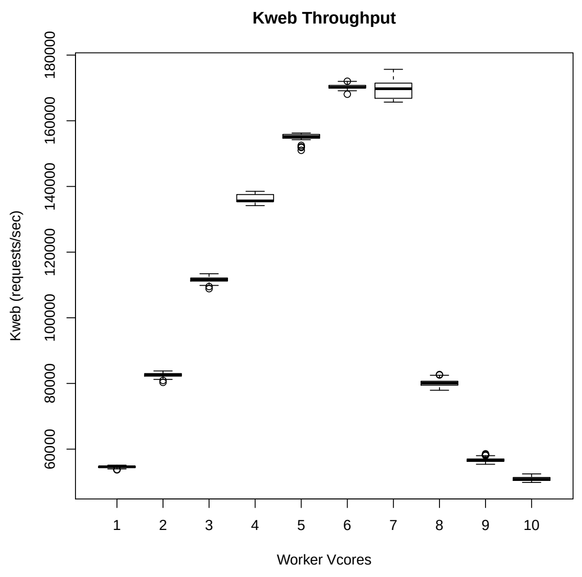

Figure 7.4: Akaros Kweb, Custom-2LS, Throughput vs. Workers

the connections. The latest trend in scalable networking stacks is partitioning connections across cores, from the queues within the NIC all the way up to the application. Megapipe[33] developed *partitioned listen sockets*, which allow an application to listen for connections for a particular queue/core combination, and FreeBSD is pursuing similar developments[16]. Although Akaros's network stack does not have these modifications, partitioning connections among cores should result in some benefits from locality and lack of contention on per-connection structures and locks.

Since the Akaros kernel cannot dispatch new connections directly to vcores, I dedicated a single vcore to handling the `accept()` call, and had that vcore dispatch new file descriptors (FDs) to the worker vcores. I used BCQs (Section 5.1) to dispatch new FDs, so that the accept loop can detect back-pressure and hangup connections if the system is overloaded.

For their part, the worker vcores would prioritize existing threads working on older requests. When there were no runnable threads, it would create threads on demand to handle any new requests or connections. Old threads were recycled in per-vcore, zombie-thread queues. Whenever a thread blocks on a syscall, if there are more requests, another thread will quickly process the next request and issue another syscall. Optionally, the 2LS can limit the number of outstanding syscalls, and choose to idle or busy-wait.

Although all of these changes sound reasonable, the overall performance of the system is only around 140,000 QPS — far less than with a naive global scheduler. It appears that the 2LS was doing its job quite well; the application would have syscalls submitted to the kernel for every connection. The problem was that the kernel itself cannot handle a heavy workload. In fact, the system performed best with a naive scheduler since it happened to not issue too many concurrent requests to the kernel. In this case, it is unfortunate that the Akaros kernel cannot handle the workload, but at least the application was able to adjust its behavior to suit its situation. Optimizing for a lousy kernel is better than not being able to optimize at all, and when the kernel's network stack gets upgraded, the application can be retuned to take advantage of any improvements. It is possible that the kernel is not to blame for the poor performance; Akaros needs better profiling tools to determine the root cause of these issues. As it stands now, the results from Table 7.2 are for a custom scheduler that disables hyperthreading on its cores, uses a global runqueue, and limits the amount of outstanding syscalls per vcore. Although it is not ideal, it is clearly an improvement over the default pthread scheduler and even kweb on Linux.

Running with Background Tasks

The webserver is an example of a high-priority task. The provisioning and allocation kernel scheduling policies were built to allow the webserver to adjust its vcores based on its workload. The kernel scheduler can backfill unused resources to low-priority applications. When demand increases, those resources are preempted from the low-priority application and allocated to the high-priority application.

To test these mechanisms, I ran kweb with the throughput httperf test and fluidanimate as a background application. In lieu of a smart scheduler that could determine its workload based on a connection rate or other metrics, I manually told kweb to increase or decrease its amount of vcores to simulate those decisions. The actual policy to make those decisions is beyond the scope of my dissertation.

Figure 7.5 shows the performance of kweb and fluidanimate where kweb adjusts its desired number of cores over time. Kweb uses the customized 2LS, where it allocates both cores in a sibling pair and halts the odd-numbered physical core. Neither fluidanimate nor any other application can use those cores while they are allocated to kweb. The test was structured as follows: kweb ran alone for 30 seconds with one worker and three vcores (physical cores one and three were halted). After 30 seconds, fluidanimate started and requested all remaining cores Every 30 seconds kweb requested two sibling cores. Those cores were preempted from fluidanimate and reallocated to kweb. At 240 seconds, to simulate a drop in workload, a script instructed kweb to yield three sibling pairs, which the kernel then allocated to fluidanimate.

As expected, kweb's performance increases and fluidanimate's decreases as cores are preempted from fluidanimate and given to kweb. Two other features of Figure 7.5 are worth mentioning. First, when fluidanimate starts up at 30 seconds, we see a slight drop in kweb's throughput. Even though kweb has exclusive access to its cores, fluidanimate shares a last-level cache on the same socket as kweb. Fluidanimate's memory accesses degrade kweb's performance. The Akaros kernel does not treat cache space or memory bandwidth as an isolatable resource yet. Second, at 210 seconds, kweb starts up another worker, but its

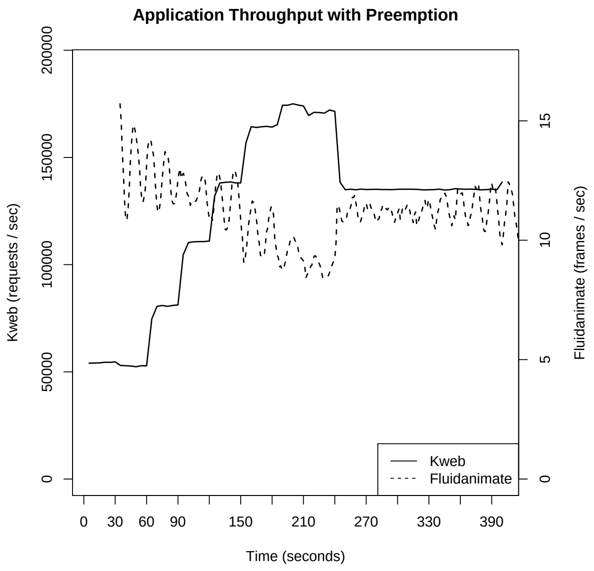

Figure 7.5: Kweb and Fluidanimate Throughput with Preemption

performance degrades. As I mentioned above, the webserver and network stack do not scale well. Regardless, the preemption and allocation mechanisms work as planned.

Future Work

Akaros's provisioning scheduler and preemption handlers do their job, but for real-world applications to take advantage of them, we need smarter user-level schedulers. One major part is to build logic that actually adapts to load. The depth of the connection queue is likely a good, basic metric for load. One minor issue with using the connection queue is that it requires fresh connections, which the "throughput plateau" test does not test. The Akaros network stack needs to be tuned and upgraded before scaling out to a large number of connections, at which point a smart 2LS that adjusts its resources based on the connection rate would be feasible. Likewise, the kernel's bottlenecks while handling numerous concurrent syscalls need to be profiled and improved. It is understandable that it is possible for the application to overload the kernel, but any degradation should be graceful.

On another note, there are many other opportunities presented by having a user-level scheduler. All of the queries used in the throughput test were for a static file that was loaded in the page cache. The only blocking calls were on socket operations. A more full-fledged webapp with serious processing and multiple syscalls that can block would lead to a wider variety of scheduling decisions. For instance, a 2LS could prioritize threads in an Earliest-Deadline-First (EDF) policy, possibly even preempting other newer threads to run older threads. Another option would be to prioritize threads based on the type of operation, say POST calls and other transactions over GETs and simple reads.

The kernel interface for requesting cores may need an improvement. Perhaps sibling cores need to be an abstraction, or at least the application can request particular physical cores. On a similar note, the uthread library's preemption handlers may need more hooks for 2LS writers. The preemption handlers work for basic user-level schedulers, but more intricate schedulers may have specialized concerns that I had not anticipated. As of now, no one is writing second level schedulers for Akaros applications, and once we have more developers, various interfaces will need to be tweaked. Changing parts of Akaros was always the plan of record; the kernel and user libraries are easily modifiable.

Chapter 8

Conclusion

In this work, I presented the "Many-Core Process" (MCP), a new process abstraction designed to support high-performance, parallel applications. The MCP is the primary component of Akaros, a research operating system designed for single-node, large-scale SMP and many-core architectures. The MCP embodies transparency, application control of physical resources, and performance isolation.

The big idea behind the MCP is the separation of the notion of cores from threads. The kernel grants spatially partitioned cores to processes, and userspace is responsible for the management of those cores. There are several benefits to this approach.

- The process retains control of its cores regardless of page faults or blocking system calls.
- With user-level scheduling, the application can prioritize whichever threads it desires and can pursue novel scheduling policies.
- Closely related to user-level scheduling, user-level *threading* provides a two-sided performance boost: faster context switches and lightweight thread structures.
- From the kernel's perspective, it can create lightweight threads or possibly skip threads completely to implement a user's system calls. Threads are not part of the contract between the user and the kernel.

In data centers and other multi-tenant systems, a single MCP will not exist in a vacuum. The Akaros scheduler allows userspace to *provision* a set of resources to an application, guaranteeing future, uninterrupted access to physical resources. When the MCP asks for the resource, the kernel will perform the *allocation*. Provisioning is directed by userspace; this is the coarse-grained resource allocation decision-making that userspace needs to be able to make. The fine-grained scheduling decisions are made by user-level schedulers *within* an MCP. Both coarse and fine-grained control over scheduling is important for applications.

I have described the MCP from the kernel and user's perspective, covering the details of virtual cores, second-level schedulers, and the process life cycle. The cores granted to MCPs are under the complete control of the application; the FTQ benchmark showed an order of magnitude less CPU interference for MCPs compared to Linux.

Granting cores to processes is relatively easy; however a more complete system must deal with blocking system calls, preempted vcores, and even the desire to yield vcores. To handle these scenarios, the kernel and user work together to guarantee event delivery and handle preempted vcores. Of special note, I have developed a class of locking algorithms that make use of the kernel's transparent interface to detect and recover from preemption: the PDR locks.

Smart applications can take advantage of the transparency and control afforded them by the kernel. Akaros's default pthread scheduler provides the basics of user-level threading, but applications can do more with customized user-level schedulers. Kweb's performance on Akaros varied substantially based on choices made by its user-level scheduler, and the best slightly outperformed Kweb on Linux, despite Akaros's immature networking stack.

Although user-level scheduling on Akaros has shown promise, much more can be done in the future. We need scheduler developers to push the limits of the MCP and help guide the evolution of its APIs, so that applications can maximize their potential and achieve predictable, high performance.

Bibliography

[1] Mike Accetta et al. “Mach: A New Kernel Foundation for UNIX Development”. In: 1986, pp. 93–112.

[2] T. E. Anderson et al. “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”. In: *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*. 1991.

[3] Thomas E. Anderson et al. “A Case for NOW (Networks of Workstations)”. In: *IEEE Micro* 15.1 (Feb. 1995), pp. 54–64. ISSN: 0272-1732. DOI: `10.1109/40.342018`. URL: `http://dx.doi.org/10.1109/40.342018`.

[4] Krste Asanović et al. “A View of the Parallel Computing Landscape”. In: *Communications of the ACM* 52.10 (Oct. 2009). DOI: `10.1145/1562764.1562783`.

[5] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006. URL: `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[6] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. “Resource containers: A new facility for resource management in server systems”. In: *Proc. of the ACM Symp. on Operating Systems Design and Implementation (OSDI)*. 1999.

[7] Paul Barham et al. “Xen and the art of virtualization”. In: *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*. 2003. ISBN: 1-58113-757-5. DOI: `http://doi.acm.org/10.1145/945445.945462`.

[8] Alan Bawden. *PCLSRing: Keeping Process State Modular*. `http://www.inwap.com/pdp10/pclsr.txt`. 1989.

[9] Rob von Behren et al. “Capriccio: Scalable threads for internet services”. In: *SOSP ’03*. Bolton Landing, NY, USA, 2003. ISBN: 1-58113-757-5. DOI: `http://doi.acm.org/10.1145/945445.945471`.

[10] Christian Bienia. “Benchmarking Modern Multiprocessors”. PhD thesis. Princeton University, Jan. 2011.

[11] David L. Black. “Scheduling Support for Concurrency and Parallelism in the Mach Operating System”. In: *Computer* 23.5 (May 1990), pp. 35–43. ISSN: 0018-9162. DOI: `10.1109/2.53353`. URL: `http://dx.doi.org/10.1109/2.53353`.

[12] Tim Brecht et al. "Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O". In: *SIGOPS Oper. Syst. Rev.* 40.4 (Apr. 2006), pp. 265–278. ISSN: 0163-5980. DOI: `10.1145/1218063.1217961`. URL: `http://doi.acm.org/10.1145/1218063.1217961`.

[13] Eric Brewer. "Parallelism in the Cloud". In: *Workshop on Hot Topics in Parallelism.* Keynote Talk. 2013.

[14] Neil Brown. *Linux Kernel Design Patterns, Part 1.* `http://lwn.net/Articles/336224/`. June 2009.

[15] Ben J. Catanzaro. *Multiprocessor System Architectures: A Technical Survey of Multiprocessor/Multithreaded Systems Using SPARC, Multilevel Bus Architectures and Solaris (SunOS).* Prentice Hall, 1994. ISBN: 0-13-089137-1.

[16] Adrian Chadd. *Upcoming RSS enhancements to the FreeBSD Network Stack.* `http://www.freebsdnews.net/2014/07/31/bafug-july-2014/`. July 2014.

[17] David Cheriton. "The V Distributed System". In: *Commun. ACM* 31.3 (Mar. 1988), pp. 314–333. ISSN: 0001-0782. DOI: `10.1145/42392.42400`. URL: `http://doi.acm.org/10.1145/42392.42400`.

[18] Melvin E. Conway. "Design of a Separable Transition-diagram Compiler". In: *Commun. ACM* 6.7 (July 1963), pp. 396–408. ISSN: 0001-0782. DOI: `10.1145/366663.366704`. URL: `http://doi.acm.org/10.1145/366663.366704`.

[19] E.C. Cooper and R. Draves. *C Threads.* Tech. rep. CMU-CS-88-154. Carnegie Mellon Computer Science Dept., June 1988.

[20] Jonathan Corbet. *LPC: Control Groups.* `http://lwn.net/Articles/459585/`. Sept. 2011.

[21] Jonathan Corbet. *(Nearly) full tickless operation in 3.10.* `http://lwn.net/Articles/549580/`. May 2013.

[22] Jonathan Corbet. *Ticket Spinlocks.* `http://lwn.net/Articles/267968/`. Feb. 2008.

[23] Systems Jan Edler et al. "Process Management for Highly Parallel UNIX Systems". In: *Proceedings of the USENIX Workshop on Unix and Supercomputers.* 1988, pp. 1–17.

[24] Thorsten von Eicken et al. *Active Messages: a Mechanism for Integrated Communication and.* Tech. rep. Berkeley, CA, USA, 1992.

[25] Jason Evans. *Kernel-Scheduled Entities for FreeBSD.* `http://www.cs.ucdavis.edu/~wu/ecs251/KernelScheduledEntity_FreeBSD_2000.pdf`. Nov. 2000.

[26] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler". In: *Proc. of the International Conference on Parallel Architecture and Compilation Techniques (PACT).* 2007. ISBN: 0-7695-2944-5. DOI: `http://dx.doi.org/10.1109/PACT.2007.40`.

[27] Dror G. Feitelson. *Job Scheduling in Multiprogrammed Parallel Systems.* 1997.

[28] Dror G. Feitelson and Larry Rudolph. “Gang Scheduling Performance Benefits for Fine-Grain Synchronization”. In: *Journal of Parallel and Distributed Computing* 16 (1992), pp. 306–318.

[29] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. “Characterizing application sensitivity to OS interference using kernel-level noise injection”. In: *SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing.* Austin, Texas: IEEE Press, 2008, pp. 1–12. ISBN: 978-1-4244-2835-9.

[30] *FreeBSD 7.0-RELEASE Release Notes.* `http://www.freebsd.org/releases/7.0R/relnotes.html`.

[31] Ruth E. Goldenberg. *Open VMS Alpha Internals and Data Structures: Scheduling and Process Control.* Elsevier, 1997. ISBN: 978-1555581565.

[32] Amit Gupta and Domenico Ferrari. “Resource partitioning for real-time communication”. In: *IEEE/ACM Trans. Netw.* 3.5 (1995), pp. 501–508. ISSN: 1063-6692. DOI: `http://dx.doi.org/10.1109/90.469956`.

[33] Sangjin Han et al. “MegaPipe: A New Programming Interface for Scalable Network I/O”. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12).* Hollywood, CA: USENIX, 2012, pp. 135–148. ISBN: 978-1-931971-96-6. URL: `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han`.

[34] Per Brinch Hansen. “Structured Multiprogramming”. In: *Commun. ACM* 15.7 (July 1972), pp. 574–578. ISSN: 0001-0782. DOI: `10.1145/361454.361473`. URL: `http://doi.acm.org/10.1145/361454.361473`.

[35] Per Brinch Hansen. “The Programming Language Concurrent Pascal.” In: *IEEE Trans. Software Eng.* 1.2 (1975), pp. 199–207.

[36] Bijun He, William N. Scherer, and Michael L. Scott. “Preemption Adaptivity in Time-published Queue-based Spin Locks”. In: *Proceedings of the 12th International Conference on High Performance Computing.* HiPC’05. Goa, India: Springer-Verlag, 2005, pp. 7–18. ISBN: 3-540-30936-5, 978-3-540-30936-9. DOI: `10.1007/11602569_6`. URL: `http://dx.doi.org/10.1007/11602569_6`.

[37] Maurice Herlihy. “Wait-free Synchronization”. In: *ACM Trans. Program. Lang. Syst.* 13.1 (Jan. 1991), pp. 124–149. ISSN: 0164-0925. DOI: `10.1145/114005.102808`. URL: `http://doi.acm.org/10.1145/114005.102808`.

[38] L.P. Hewlett-Packard Development Company. *HP OpenVMS System Services Reference Manual.* `http://h71000.www7.hp.com/doc/84final/4527/4527pro_contents.html`. June 2010.

[39] Benjamin Hindman et al. *Mesos: A platform for fine-grained resource sharing in the data center,” UCBerkeley.* Tech. rep. Online]. Available, 2010.

[40] C. A. R. Hoare. “Monitors: An Operating System Structuring Concept”. In: *Commun. ACM* 17.10 (Oct. 1974), pp. 549–557. ISSN: 0001-0782. DOI: `10.1145/355620.361161`. URL: `http://doi.acm.org/10.1145/355620.361161`.

[41] Tim Hockin. “Control Loops in Userspace”. In: *Linux Plumbers Conference*. `http://www.linuxplumbersconf.org/2011/ocw/system/presentations/705/original/Control_Loops_In_Userspace.pdf`. 2011.

[42] Steven Hofmeyr et al. “Juggle: Proactive Load Balancing on Multicore Computers”. In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing*. HPDC ’11. San Jose, California, USA: ACM, 2011, pp. 3–14. ISBN: 978-1-4503-0552-5. DOI: `10.1145/1996130.1996134`. URL: `http://doi.acm.org/10.1145/1996130.1996134`.

[43] IBM. *IBM System/370 Extended Architecture, Principles of Operation*. IBM Publication No. SA22-7085. 1983.

[44] Adaptive Computing Enterprises Inc. *Maui Scheduler Administrator’s Guide*. `http://docs.adaptivecomputing.com/maui/index.php`. 1999-2014.

[45] Apple Inc. *Grand Central Dispatch (GCD) Reference*. `https://developer.apple.com/library/mac/documentation/performance/reference/gcd_libdispatch_ref/Reference/reference.html`.

[46] Sun Microsystems Inc. *JDK 1.1 for Solaris Developers’s Guide*. `http://docs.oracle.com/cd/E19455-01/806-3461/book-info/index.html`. 2000.

[47] David B. Jackson, Quinn Snell, and Mark J. Clement. “Core Algorithms of the Maui Scheduler”. In: *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*. JSSPP ’01. London, UK, UK: Springer-Verlag, 2001, pp. 87–102. ISBN: 3-540-42817-8. URL: `http://dl.acm.org/citation.cfm?id=646382.689682`.

[48] Terry Jones et al. “Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the Operating System”. In: *SC ’03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10. ISBN: 1-58113-695-1.

[49] R. E. Kessler and Mark D. Hill. “Page Placement Algorithms for Large Real-indexed Caches”. In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 338–359. ISSN: 0734-2071. DOI: `10.1145/138873.138876`. URL: `http://doi.acm.org/10.1145/138873.138876`.

[50] Sanjay Kumar et al. *Re-architecting VMMs for Multicore Systems: The Sidecore Approach*. 2007.

[51] Leslie Lamport. “A New Solution of Dijkstra’s Concurrent Programming Problem”. In: *Commun. ACM* 17.8 (Aug. 1974), pp. 453–455. ISSN: 0001-0782. DOI: `10.1145/361082.361093`. URL: `http://doi.acm.org/10.1145/361082.361093`.

[52] Butler W. Lampson. *A description of the CEDAR language : a CEDAR reference manual*. Tech. rep. CSL-83-15. Xerox (Palo Alto, CA US), 1983. URL: `http://opac.inria.fr/record=b1017007`.

[53] Butler W. Lampson and David D. Redell. "Experience with Processes and Monitors in Mesa". In: *Commun. ACM* 23.2 (Feb. 1980), pp. 105–117. ISSN: 0001-0782. DOI: `10.1145/358818.358824`. URL: `http://doi.acm.org/10.1145/358818.358824`.

[54] J. Lange et al. "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing". In: *IPDPS '10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. Atlanta, Georgia, USA: IEEE Computer Society, Apr. 2010.

[55] Edward A. Lee. "The Problem with Threads". In: *Computer* 39.5 (May 2006), pp. 33–42. ISSN: 0018-9162. DOI: `10.1109/MC.2006.180`. URL: `http://dx.doi.org/10.1109/MC.2006.180`.

[56] Jonathan Levin. *GCD Internals: The Undocumented Side of the Grand Central Dispatcher*. `http://newosxbook.com/articles/GCD.html`. Feb. 2014.

[57] Jialin Li et al. "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 2014, 9:1–9:14. ISBN: 978-1-4503-3252-1. DOI: `10.1145/2670979.2670988`. URL: `http://doi.acm.org/10.1145/2670979.2670988`.

[58] *Linux Activations*. `http://moais.imag.fr/membres/vincent.danjean/linux-activations.html`.

[59] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. "Condor - A Hunter of Idle Workstations." In: *ICDCS*. IEEE Computer Society, 1988, pp. 104–111. URL: `http://dblp.uni-trier.de/db/conf/icdcs/icdcs88.html#LitzkowLM88`.

[60] *lxc Linux Containers*. `http://lxc.sourceforge.net/`.

[61] Brian D. Marsh et al. "First-Class User-Level Threads". In: *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*. 1991, pp. 110–121.

[62] Henry H. Mashburn. "The C.mmp/Hydra Project: An Architectural Overview". In: *Computer Structures: Principles and Examples*. Ed. by Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. New Yor: McGraw-Hill Book Company, 1982.

[63] *Maui Scheduler Open Cluster Software*. `http://mauischeduler.sourceforge.net/`.

[64] Marissa Mayer. "Imagination, Immediacy, and Innovation... and a little glimpse under the hood at Google". In: *Google I/O*. Keynote Talk. 2008.

[65] Cathy McCann, Raj Vaswani, and John Zahorjan. "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-memory Multiprocessors". In: *ACM Trans. Comput. Syst.* 11.2 (May 1993), pp. 146–178. ISSN: 0734-2071. DOI: `10.1145/151244.151246`. URL: `http://doi.acm.org/10.1145/151244.151246`.

[66] Declan McCullagh. *TurboTax e-filing woes draw customer ire*. Ed. by CNET.com. `http://news.cnet.com/TurboTax-e-filing-woes-draw-customer-ire/2100-1038_3-6177341.html`. Apr. 2007.

[67] Paul R. McJones and Garret F. Swart. *Evolving the Unix system interface to support multithreaded programs*. Tech. rep. Proceedings of the Winter 1989 USENIX Conference, 1987.

[68] Paul E. Mckenney et al. "Read-Copy Update". In: *Ottawa Linux Symposium*. 2001, pp. 338–367.

[69] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". In: *ACM Trans. Comput. Syst.* 9.1 (1991), pp. 21–65. ISSN: 0734-2071. DOI: `http://doi.acm.org/10.1145/103727.103729`.

[70] Clifford Mercer, Stefan Savage, and Hideyuki Tokuda. "Processor Capacity Reserves for Multimedia Operating Systems". In: *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*. 1994.

[71] Microsoft. *User-Mode Scheduling*. URL: `http://msdn.microsoft.com/en-us/library/windows/desktop/dd627187%28v=vs.85%29.aspx`.

[72] Sun Microsystems. *Multithreading in the Solaris Operating Environment*. `http://home.mit.bme.hu/~meszaros/edu/oprendszerek/segedlet/unix/2_folyamatok_es_utemezes/solaris_multithread.pdf`. 2002.

[73] Aloysius K. Mok, Xiang (Alex) Feng, and Deji Chen. "Resource Partition for Real-Time Systems". In: *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 75.

[74] Ingo Molnar. *1:1 Threading vs. Scheduler Activations*. `http://lkml.org/lkml/2002/9/24/33`.

[75] José Moreira et al. "Designing a highly-scalable operating system: the Blue Gene/L story". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: `http://doi.acm.org/10.1145/1188455.1188578`. URL: `http://doi.acm.org/10.1145/1188455.1188578`.

[76] David Mosberger and Tai Jin. "Httperf&Mdash;a Tool for Measuring Web Server Performance". In: *SIGMETRICS Perform. Eval. Rev.* 26.3 (Dec. 1998), pp. 31–37. ISSN: 0163-5999. DOI: `10.1145/306225.306235`. URL: `http://doi.acm.org/10.1145/306225.306235`.

[77] Steve Muir and Jonathan Smith. "AsyMOS - An Asymmetric Multiprocessor Operating System". In: *IEEE Conf on Open Architectures and Network Programming (OPENARCH)*. 1998, pp. 25–34.

[78] K.J. Nesbit et al. "Multicore Resource Management". In: *Micro, IEEE* 28.3 (May 2008), pp. 6–16. ISSN: 0272-1732. DOI: `10.1109/MM.2008.43`.

[79] Old-computers.com. *VAX 11/780, The First VAX System (October 1977)*. http://www.old-computers.com/history/detail.asp?n=20&t=3.

[80] *OpenMP*. http://openmp.org.

[81] J.K. Ousterhout. "Scheduling techniques for concurrent systems". In: *Proceedings of the 3rd International Conference on Distributed Computing Systems*. 1982, pp. 22–30.

[82] Heidi Pan. "Cooperative Hierarchical Resource Management for Efficient Composition of Parallel Software". PhD thesis. Massachusetts Institute of Technology, June 2010.

[83] Heidi Pan, Ben Hindman, and Krste Asanović. "Lithe: Enabling Efficient Composition of Parallel Libraries". In: *Proc. of HotPar*. 2009.

[84] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q". In: *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 55. ISBN: 1-58113-695-1.

[85] Rob Pike et al. "Plan 9 from Bell Labs". In: *Proceedings of the Summer 1990 UKUUG Conference*. 1990, pp. 1–9.

[86] Rob Pike et al. "Process sleep and wakeup on a shared-memory multiprocessor". In: *EurOpen Conference*. 1991, pp. 161–166.

[87] Daniel Price and Andrew Tucker. "Solaris Zones: Operating System Support for Consolidating Commercial Workloads". In: *Proceedings of the 18th USENIX Conference on System Administration*. LISA '04. Atlanta, GA: USENIX Association, 2004, pp. 241–254. URL: http://dl.acm.org/citation.cfm?id=1052676.1052707.

[88] David Probert. *Evolution of the Windows Kernel Architecture*. http://www.palermo.edu/ingenieria/Oct2009.ppt. Aug. 2009.

[89] Alexander Rasmussen et al. "TritonSort A Balanced Large-Scale Sorting System". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*. ACM, 2011.

[90] Greg Regnier et al. "TCP Onloading for Data Center Servers". In: *Computer* 37.11 (Nov. 2004), pp. 48–58. ISSN: 0018-9162. DOI: 10.1109/MC.2004.223. URL: http://dx.doi.org/10.1109/MC.2004.223.

[91] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007, pp. I–XXV, 1–303. ISBN: 978-0-596-51480-8.

[92] Injong Rhee and Chi-Yung Lee. "An Efficient Recovery-based Spin Lock Protocol for Preemptive Shared-memory Multiprocessors". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 77–86. ISBN: 0-89791-800-2. DOI: 10.1145/248052.248063. URL: http://doi.acm.org/10.1145/248052.248063.

[93] Rolf Riesen et al. "Designing and implementing lightweight kernels for capability computing". In: *Concurr. Comput. : Pract. Exper.* 21 (6 Apr. 2009), pp. 793–817. ISSN: 1532-0626. DOI: `10.1002/cpe.v21:6`. URL: `http://portal.acm.org/citation.cfm?id=1526619.1526623`.

[94] Jerome Howard Saltzer. "Traffic Control in a Multiplexed Computer System". PhD thesis. Massachusetts Institute of Technology, June 1966.

[95] Michael D. Schroeder and Jerome H. Saltzer. "A Hardware Architecture for Implementing Protection Rings". In: *Commun. ACM* 15.3 (Mar. 1972), pp. 157–170. ISSN: 0001-0782. DOI: `10.1145/361268.361275`. URL: `http://doi.acm.org/10.1145/361268.361275`.

[96] Michael L. Scott, Thomas J. Leblanc, and Brian D. Marsh. "Multi-model parallel programming in Psyche". In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 1990, pp. 70–78.

[97] Michael L. Scott and William N. Scherer. "Scalable Queue-based Spin Locks with Timeout". In: *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming.* PPoPP '01. Snowbird, Utah, USA: ACM, 2001, pp. 44–52. ISBN: 1-58113-346-4. DOI: `10.1145/379539.379566`. URL: `http://doi.acm.org/10.1145/379539.379566`.

[98] Christopher Small and Margo Seltzer. *Scheduler Activations on BSD: Sharing Thread Management Between Kernel and Application.* Tech. rep. 1995.

[99] Livio Soares and Michael Stumm. "FlexSC: flexible system call scheduling with exception-less system calls". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation.* OSDI'10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 1–8. URL: `http://portal.acm.org/citation.cfm?id=1924943.1924946`.

[100] Stephen Soltesz et al. "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors". In: *SIGOPS Oper. Syst. Rev.* 41.3 (2007), pp. 275–287. ISSN: 0163-5980. DOI: `http://doi.acm.org/10.1145/1272998.1273025`.

[101] Matthew J. Sottile and Ronald Minnich. "Analysis of microbenchmarks for performance tuning of clusters." In: *CLUSTER.* IEEE Computer Society, 2004, pp. 371–377. URL: `http://dblp.uni-trier.de/db/conf/cluster/cluster2004.html#SottileM04`.

[102] George G. Sutherland et al. "Livermore Time-sharing System for the CDC 7600". In: *SIGOPS Oper. Syst. Rev.* 5.2-3 (June 1971), pp. 6–26. ISSN: 0163-5980. DOI: `10.1145/1232916.1232917`. URL: `http://doi.acm.org/10.1145/1232916.1232917`.

[103] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. “Cm*: A Modular, Multi-microprocessor”. In: *Proceedings of the June 13-16, 1977, National Computer Conference*. AFIPS ’77. Dallas, Texas: ACM, 1977, pp. 637–644. DOI: `10.1145/1499402.1499515`. URL: `http://doi.acm.org/10.1145/1499402.1499515`.

[104] Daniel C. Swinehart et al. “A Structural View of the Cedar Programming Environment”. In: *ACM Trans. Program. Lang. Syst.* 8.4 (Aug. 1986), pp. 419–490. ISSN: 0164-0925. DOI: `10.1145/6465.6466`. URL: `http://doi.acm.org/10.1145/6465.6466`.

[105] Charles P. Thacker and Lawrence C. Stewart. “Firefly: A Multiprocessor Workstation”. In: *SIGOPS Oper. Syst. Rev.* 21.4 (Oct. 1987), pp. 164–172. ISSN: 0163-5980. DOI: `10.1145/36204.36199`. URL: `http://doi.acm.org/10.1145/36204.36199`.

[106] *The Go Programming Language*. `http://golang.org/`.

[107] James E. Thornton. “Parallel Operation in the Control Data 6600”. In: *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems*. AFIPS ’64 (Fall, part II). San Francisco, California: ACM, 1965, pp. 33–40. DOI: `10.1145/1464039.1464045`. URL: `http://doi.acm.org/10.1145/1464039.1464045`.

[108] Paul Turner. *User-level Threads... with threads*. `http://www.linuxplumbersconf.org/2013/ocw/proposals/1653`. 2013.

[109] *Ultrix*. `http://osdata.com/oses/ultrix.htm`.

[110] Ben Verghese, Anoop Gupta, and Mendel Rosenblum. “Performance isolation: sharing and isolation in shared-memory multiprocessors”. In: *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*. San Jose, California, United States: ACM, 1998, pp. 181–192. ISBN: 1-58113-107-0. DOI: `http://doi.acm.org/10.1145/291069.291044`.

[111] Frederic Weisbecker. *Nohz cpusets*. `https://lkml.org/lkml/2011/8/15/245`. Aug. 2011.

[112] Nathan J. Williams. “An Implementation of Scheduler Activations on the NetBSD Operating System”. In: *USENIX Annual Technical Conference*. 2002, pp. 10–15.

[113] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael L. Scott. “Scalable Spin Locks for Multiprogrammed Systems.” In: *IPPS*. Ed. by Howard Jay Siegel. IEEE Computer Society, 1994, pp. 583–589. ISBN: 0-8186-5602-6. URL: `http://dblp.uni-trier.de/db/conf/ipps/ipps1994.html#WisniewskiKS94`.

[114] Derek Wright. “Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor”. In: *Proceedings of the Linux Clusters: The HPC Revolution conference*. Champaign - Urbana, IL, June 2001.

[115] John Zahorjan, Ed Lazowska, and Derek Eager. “The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems”. In: *IEEE Trans. Parallel Distrib. Syst.* 2.2 (Apr. 1991), pp. 180–198. ISSN: 1045-9219. DOI: `10.1109/71.89064`. URL: `http://dx.doi.org/10.1109/71.89064`.

[116] John Zahorjan and Cathy McCann. “Processor Scheduling in Shared Memory Multiprocessors”. In: *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’90. New York, NY, USA: ACM, 1990, pp. 214–225. ISBN: 0-89791-359-0. DOI: `10.1145/98457.98760`. URL: `http://doi.acm.org/10.1145/98457.98760`.