The DASH Local Kernel Structure

David P. Anderson
Shin-Yuan Tzou

November 7, 1988

ABSTRACT

The DASH project has designed the network communication architecture for a large, high-performance distributed system, and is now building a portable operating system kernel to run on the nodes of this system. The DASH kernel supports the communication architecture by providing efficient local communication, support for user-level services, naming support, and transparent remote service access. It is designed to provide increased performance through parallelism on shared-memory multiprocessors.

This report describes some of the basic components of the DASH kernel: process scheduling, synchronization mechanisms, timers and message-passing. It also describes the ways in which these facilities are made available to user processes. The other components of the kernel, such as the virtual memory and network communication systems, are described in separate documents.

Sponsored by the California MICRO program, AT&T Bell Laboratories, Digital Equipment Corporation, IBM Corporation, Olivetti S.p.A, and the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871. Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

1. INTRODUCTION

The DASH project [2] is designing 1) an experimental distributed system architecture [14], and 2) a portable operating system kernel that implements the architecture. This report describes the design of the central part of the DASH kernel. The parts of the kernel dealing with virtual memory and network communication are described in separate documents ([14, 15]).

The report is organized as follows: Section 2 discusses the goals of the kernel design, and the technological projections on which it is based. Section 3 is an overview of the kernel design. Section 4 describes process mechanisms and scheduling. Section 5 discusses synchronization using spin-locks. Section 6 describes the timer mechanism, and Sections 7 through 11 describe the local message-passing system. Sections 12 and 13 discuss system calls and exceptions. Section 14 gives some examples of message-passing applications.

2. GOALS AND ASSUMPTIONS

The DASH kernel is designed for a class of shared memory multiprocessor (SMMP) machines with uniform shared memory access speed, large main memory, and paged virtual memory hardware. Due to the system bus bottleneck, machines of this class typically are limited to 10 to 20 processors [6]. We anticipate that this type of architecture will be common in both workstations and server machines in the future.

The DASH kernel provides processes, virtual memory, local and remote interprocess communication, and I/O device access. Its design has the following goals:

- **Kernel parallelism.** We want to generate "fine-grained" kernel parallelism [1], i.e., parallelism within single kernel operations as well as between concurrent operations. At the same time, we seek to avoid problems due to 1) increased software complexity, and 2) the performance overhead of process creation, scheduling, synchronization, and IPC.
- **Real-time capabilities.** The scheduler and message-passing system support real-time processing and communication requirements.
- **Portability.** The DASH kernel encapsulates its machine-dependent part in a "virtual machine" interface. This interface has been designed to encompass features that are unique to SMMP's, so that the upper levels of the kernel can exploit these features.
- **Support for user-level services.** We intend that services such as file services, window services, and transaction managers be run at the user level (i.e., in protected virtual address spaces). This has several advantages, including flexibility and reliability. The kernel design attempts to minimize the performance impact of user-level services by providing fast local IPC and support for user-level caching in the management of physical memory.

[1] In typical UNIX systems, about half of the total CPU time is spent in the kernel [3]. In future distributed systems this figure may be different: services such as file systems may be moved into user virtual spaces, while time spent in communication protocols will increase. However, it is likely that kernel parallelism will be as important as user-level parallelism in determining overall system performance.

3. THE DASH LOCAL STRUCTURE: OVERVIEW

This section gives a brief overview of the DASH kernel design. The structure of the kernel can be discussed in terms of both its *information structure* (the organization of data and instructions) and its *execution structure* (the organization of process and exception handler execution).

3.1. Information Structure

The DASH VM system [15] supports multiple virtual address spaces (VAS's). There is a single kernel VAS, and multiple user VAS's. Within the kernel VAS, code and data are organized according to the principles of object-oriented programming [2]. The kernel's data structures are organized as *objects* that can be accessed only through abstract procedural interfaces. With few exceptions, kernel code consists of *member functions* that define operations on objects.

The object-oriented structure of the kernel is reflected in the user-level facilities provided by the kernel. User processes can hold capability-like *user object references* to kernel objects, and can perform operations on these objects using "system calls".

3.2. Execution Structure

A DASH *process* is the abstraction of the sequential execution of a program. "Process" and "virtual address space" (VAS) are separate constructs; a VAS may contain any number of concurrent processes, and a process may move between VAS's during its execution. Processes are created and scheduled by the kernel.[3]

At any point, a process executes under the mapping of a particular VAS. While it executes in the kernel VAS the processor is placed in *kernel mode*, in which it can execute privileged instructions such as changing the processor interrupt priority. While a process executes in a user VAS the processor is placed in *user mode*, in which these instructions are illegal.

Kernel processes execute permanently in kernel mode. They perform ahead of time, or in parallel, tasks that in other systems are done in sequence with user processes or with interrupt handlers. Examples include the following.

- The VM system uses a static set of kernel processes for page replacement and zero-filling [15].
- Network drivers use processes to replenish the supply of receive buffers and to free completed transmit buffers.
- The network communication system [14] can uses parallel processes for concurrent execution of protocols. This can be done either *vertically* (a process is assigned to each incoming packet and "sheperds" it through protocol layers [5, 8]) or *horizontally* (a protocol layer is implemented as a set of processes).

[2] The DASH kernel is being implemented in C++ [11], an object-oriented programming language. C++ terminology (*object*, *class*, *base class*, *derived class*, *member function*, *virtual function*, *constructor*, *destructor*, *this*, and *private data*) is used throughout this report.

[3] It would be possible to also use a user-level multi-tasking system ([7, 9]) in which multi-tasking is handled by user-level functions. However, it is our intent that the potential concurrency of DASH processes will outweigh their higher scheduling overhead.

In addition to process-level execution, the kernel contains hardware and software interrupt routines. Threads of kernel-mode execution (processes and exception handlers) interact in two ways: 1) via synchronized access to shared objects; 2) via a local message-passing system (see below).

User processes normally execute in user mode. They communicate with other processes by performing message-passing operations. During these operations, the process traps into the kernel, and executes in kernel mode for a period. Some message-passing operations are requests for kernel services, or *system calls*. Multiple processes in a user VAS can execute concurrent message-passing operations, including system calls.

3.3. Real Time Capabilities

DASH is a ''real-time system''. The DASH kernel has traditional real-time features such as preemptive deadline-based processing of input events. In addition it supports *Real-Time Message Streams* (RMS), which are network communication channels with real-time performance parameters [14].

Part of the DASH philosophy is that real-time deadlines should replace other forms of prioritization in all parts of the system, both local and remote. This will allow activities with strict real-time limitations (e.g., digitized video and audio communication) to coexist with activities that have less stringent limits (e.g., conventional user interface traffic) or no limits.

The class `ABS_TIME` represents an absolute real time in a high-resolution (microsecond-level) coordinate system. Overloaded arithmetic operators allow `ABS_TIME`s to be conveniently manipulated [13]. The number of bits used to represent an absolute time must be large enough to accommodate the maximum anticipated time between crashes without rollover. In addition, a range of the `ABS_TIME` space is reserved for *transfinite* values that are greater than any attainable real time. In the current implementation, an `ABS_TIME` is a 64-bit quantity storing the number of microseconds since the system was booted.

The class `REL_TIME` represents a difference between absolute times. Its interface and implementation are analogous to those of `ABS_TIME`.

3.4. The Local Message-Passing System

The dynamic structure of the DASH kernel is centered around a *local message-passing (MP) system*. The MP system consists of two major parts:

- **Message representation.** A message (an object of class `MESSAGE`) is a logical array of bytes, implemented by a data structure consisting of a *header* and a set of non-contiguous *data areas*. The class provides a set of operations for creating, manipulating, and accessing messages.
- **MP operations.** Instead of providing a single IPC paradigm such as Remote Procedure Call (RPC), the DASH MP facility provides both stream and request-reply style IPC. Furthermore, the system uses the inheritance facility of C++ to provide an *extensible* MP system. Several different types of MP objects, with different semantics, can be accessed through a uniform MP interface.

The message-passing system is available to user processes; it is their sole means of communicating with other user processes and with the kernel. The DASH VM system provides a mechanism for efficient movement of bulk data between VAS's [15].

4. PROCESSES AND SCHEDULING

This section describes the DASH kernel mechanisms for process creation and scheduling. The goals of these mechanisms are:

- *Efficiency*: we are interested in kernel structuring based on multiple concurrent process. The time needed for process creation and scheduling should be sufficiently small that the process model is competitive with coroutine or procedural models. In particular, a context switch should take no longer than several procedure calls.
- *Real time support*: the scheduler must let processes express their real-time requirements, and must attempt to satisfy these requirements.

The DASH process scheduler accommodates both uniprocessors and shared-memory multiprocessors. All process scheduling is done on the basis of per-process real-time *deadlines*. The scheduling policy approximates *preemptive multiprocessor deadline scheduling*: on a machine with *n* CPU's, the processes with the *n* earliest deadlines should execute. [4]

Deadlines are represented as `ABS_TIME` objects (see Section 3.3). *Transfinite* deadlines are used for ''background'' processes. These deadlines are greater than all finite deadlines, ensuring that background processes will execute only when no ''foreground'' processes (i.e., those with finite deadlines) are runnable.

4.1. Process Facility Interface

The process facility is represented by two classes, `PROCESS` and `SCHEDULER`. Processes are represented by objects of the `PROCESS` class. At any given time, the *state* of a process is one of the following:

`RUNNING`: the process is currently executing.

`RUNNABLE`: the process is ready to execute, but is not currently executing.

`SLEEPING`: the process is neither ready to execute nor currently executing.

Kernel processes are created by

```
PROCESS::PROCESS(
        void* entry_point(),
        int   arg_count,
        ...
);
```

`Entry_point` is the address (in the kernel VAS) of the *initial procedure* of the process. `Arg_count` specifies the number of arguments to this procedure; they are additional arguments to the constructor. The initial state of the process is `SLEEPING`; it will not execute until awakened using `PROCESS::wakeup()`.

Each process has a *kernel stack* for its kernel-mode execution. This stack is allocated from the *resident subregion* of the kernel space [15] when the process is created, and is

[4] Cache footprint effects are ignored for kernel processes. We expect that, in the steady state, frequently used kernel code exists in the cache of all processors, and it makes little difference whether a kernel process runs on one processor or migrates randomly. This is probably not true for user processes; we

deallocated when the process is deleted.

User processes are created by

```
PROCESS::PROCESS(
        VAS*            vas,
        VIRT_ADDR       stack_top,
        void*           entry_point(),
        int             arg_count,
        ...
);
```

The new process will be associated with the user VAS specified by `vas`. The kernel stack is allocated as above. The user-mode stack grows downwards from (and not including) `stack_top`. Space for this stack must have already been allocated in the user VAS.

`Entry_point` is the address (in the user VAS) of the initial procedure of the process. The initial state is `SLEEPING`. `Arg_count` specifies the number of arguments to this procedure; they are additional arguments to the constructor. Because they are passed in registers to the new process (see below) the number of arguments is limited; on the Sun 3 implementation the limit is four.

When the new process is awakened, it continues executing the `PROCESS` constructor in kernel mode. The arguments to its initial procedure are moved into hardware registers. The constructor then arranges for the process to switch to user mode, and to execute a *user process startup stub*, a piece of code mapped into all user VAS's. The stub moves the arguments from the registers onto the process' user stack. The process then begins executing its initial procedure.

The other member functions of `PROCESS` are:

```
PROCESS::new_deadline(
        ABS_TIME    deadline
);

void
PROCESS::wakeup(
        ABS_TIME    deadline
);
```

`PROCESS::new_deadline()` should be called when a running or runnable process needs to be given a new deadline. `PROCESS::wakeup()` is used to make a sleeping process runnable.

Process scheduling is done by the `SCHEDULER` module. It has the following operations:

are considering other schemes for scheduling them.

```
SCHEDULER::sleep();

SCHEDULER::exit();

SCHEDULER::timed_sleep(
        REL_TIME    delay
);

SCHEDULER::set_own_deadline(
        ABS_TIME    deadline
);
```

`SCHEDULER::sleep()` changes the caller's state to `SLEEPING` and switches to another process. A process terminates by calling `SCHEDULER::exit()`. `SCHEDULER::timed_sleep()` puts the caller to sleep for the indicated amount of time. `SCHEDULER::set_own_deadline()` changes the caller's scheduling deadline; it is a special case of `PROCESS::new_deadline()`.

4.2. Process Scheduling Implementation

4.2.1. Context Switches on Software Interrupt

The scheduler invokes context switches by triggering *software interrupts*. A software interrupt mechanism (supported by most modern architectures) provides an interrupt type that 1) can be requested by software, 2) has lower priority than any hardware interrupt, and 3) is always handled on the requesting processor. If the software interrupt is requested more than once before it is handled, the handler is executed only once.

Since all context switches are done from the software interrupt level, preemption from within nested hardware interrupts does not cause ''orphaned'' exception frames. If multiple processes are enabled by nested hardware interrupt handlers, only one context switch is done.

4.2.2. Data Structures

The `SCHEDULER` object uses the following data structures:

- A single *runnable queue*, consisting of a list of `PROCESS` objects sorted in order of increasing deadline. All `RUNNABLE` processes are in this queue. [5]
- A single *running processes queue*, also represented as a list of `PROCESS` objects, sorted in order of increasing deadline. It contains all processes currently executing (i.e., all `RUNNING` processes).
- One *idle process* per processor. This process has a far deadline. The code executed by the idle process is an infinite loop.[6] The existence of idle processes ensures that there is always a runnable process for each processor.

[5] The decision to use a single runnable queue was made in the interests of simplicity. If this queue proves to be a bottleneck in the system, the decision will be reevaluated.

[6] Ideally this loop should generate no bus traffic, so that DMA operations (and other processors, on an SMMP) are not affected. If there is no per-processor cache large enough for a tight loop, a `WAIT` instruction can be used.

4.2.3. Scheduling Algorithms

This section describes the algorithm used by the scheduler. The following machine-dependent constants are used:

> `SWITCH_COST`: the approximate time (in `REL_TIME` units) needed for a context switch.
>
> `SOFTINT_COST`: the approximate time for a software interrupt and context switch.
>
> `IPI_COST`: the approximate time for an interprocessor interrupt and context switch.

The scheduler is divided into two parts. The "top half" consists of the member functions of the `PROCESS` and `SCHEDULER` classes. The "bottom half" consists of the software interrupt handler and the interprocessor interrupt handler; these are assumed to be mutually non-interrupting. Both halves are machine-independent.

The following variables are used:

> `earliest_runnable`: the deadline of the earliest runnable process.
>
> `current_deadline`: (per-processor) the deadline of the process currently executing on this CPU.
>
> `latest_running`: the deadline of the latest running process.
>
> `latest_CPU`: the CPU on which the latest running process is running.

The top half logic is as follows:

- Suppose a process is made runnable (`PROCESS::wakeup()`) or the deadline of a runnable process is decreased (`PROCESS::new_deadline()`). Let X denote the new deadline. Suppose that

  $X < earliest\_runnable$

  If, in addition,

  $X < current\_deadline - SOFTINT\_COST$

  then a software interrupt is requested. Otherwise, if

  $X < latest\_running - IPI\_COST$

  an interprocessor interrupt is sent to `latest_CPU`.
- When a process sleeps or exits, a software interrupt is requested.
- When a process increases its deadline (using `SCHEDULER::new_deadline()`) so that

  $current\_deadline > earliest\_runnable + SOFTINT\_COST$

  a software interrupt is requested.

The bottom half logic is as follows:

- On an interprocessor interrupt, if

  $current\_deadline > earliest\_runnable + SWITCH\_COST$

  a context switch is done to the earliest runnable process. If in addition (after removing that process from the runnable queue) *earliest_runnable < latest_running - IPI_COST*

an interprocessor interrupt is sent to `latest_CPU`.

- On a software interrupt, a context switch is done to the earliest runnable process. If in addition (after removing that process from the runnable queue)

  latest_running > earliest_runnable + IPI_COST

  then an interprocessor interrupt is sent to `latest_CPU`.

5. SPIN LOCKS

The DASH kernel uses both *sleep locks* and *spin locks* to serialize concurrent accesses to shared objects. When a sleep lock is busy, the requesting process blocks and another process is executed on that CPU. When a spin lock is busy, the requesting process (and CPU) loops until the lock is available. [7] The choice of lock type is determined as follows:

- If the object is accessed from an interrupt routine, a spin lock must be used.
- The object is part of the process scheduling mechanism (e.g., a process queue), a spin lock must be used.
- If the expected CPU time of an object access (averaged over all the operations on the object, not just the one being performed) is longer than four context switch times, a sleep lock should be used.

This section discusses spin locks. Different types of sleep locks can be implemented either using the message-passing system (Section 13) or directly on top of the `SCHEDULER` class.

5.1. Execution States

On a shared-memory multiprocessor, spin locks must protect against concurrent access both from the caller's CPU and from other CPU's. While protection from other CPU's can be achieved using a test-and-set bit in shared memory, protection from the caller's CPU requires a mechanism such as interrupt masking. To minimize the duration and severity of interrupt masking, we categorize spin locks according to the sources of their lock requests. [8]

To formalize this situation, we divide kernel-mode execution (i.e., periods during which spin-lock requests may occur) into the following *execution states* (see Figure 5.1).

- **Process states.** The normal execution of kernel processes, user processes during MP operations (including system calls), and user processes during exception traps and page faults, takes place in this state. Each process defines a separate execution state.

[7] It is possible that other lock types (such as a spin/sleep lock that spins for a fixed period, then sleeps) will be used in later versions of DASH.

[8] Interrupt-handling latency, and missed interrupts in particular, is of great concern on uniprocessors, and suggests minimizing the duration of interrupt handlers and of process-level interrupt masking. On SMMP architectures that "load-balance" interrupt handling (e.g., the Sequent Symmetry [10]), the problem is less severe. If there are more CPU's than interrupt sources, interrupt handlers may safely take about as long as the average time between successive interrupts. However, it may still be necessary to minimize interrupt masking from the process level.

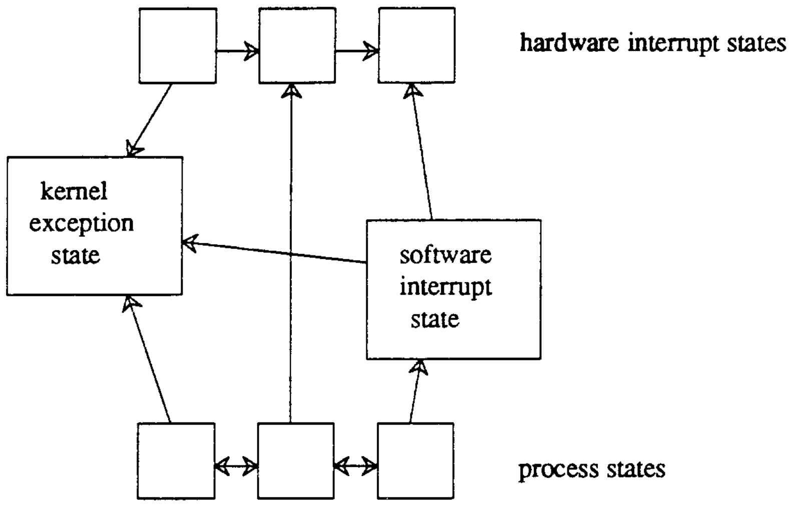

Figure 5.1: Execution states (arrows denote possible preemption).

- **Hardware interrupt states.** Each interrupt source defines a separate execution state.
- **Software interrupt state.** The software interrupt mechanism is used for scheduling, and optionally used for timer event and network packet handling.
- **Kernel exception state.** This state is entered by page faults in kernel mode and by kernel error traps.

A state may *interrupt* another state (this includes preemption of one process by another). Each processor has a logical *state transition mask* that limits interruptions between execution states. The implementation of the state transition mask might be the interrupt priority level of the processor, or an interrupt-masking bit array. It might also include a software flag that disables context switching.

A given shared object may be accessed from several execution states (for example, a message-passing port may be sent to from a hardware interrupt state and received from in a process state). Therefore requests to acquire the spin lock protecting that object are made only from this set of execution states. To prevent deadlock, the spin-lock implementation must use the state transition mask mechanism to enforce the following condition:

> **Suppose a particular spin lock is requested at execution states X and Y, and state Y can interrupt state X. Then, whenever the lock is held in state X, the transition mask must be such that interrupt transitions from X to Y are disabled.**

Handlers for kernel-generated exceptions (e.g., page faults on pageable kernel data structures) present a potential problem, since these handlers cannot be masked. If the handler were to request a lock held by the faulting process, that process would be deadlocked. This situation can be avoided by imposing the following restrictions: 1) the page fault handler cannot access pageable data, and 2) pageable data structures cannot be accessed while holding locks that may be requested by the page fault handler.

5.2. Specification of State Sets

To allow the spin lock implementation to enforce the above rule while reducing the limitations on transitions between states, the programmer must specify the set of execution states that can request each spin lock. A set of execution states is specified using the machine-dependent macro:

```
ADD_SET(set, state)     // add a state to a set
```

The possible values of `state` are machine-dependent; they include at least the following:

```
NO_STATE          // represents the empty set of states
PROCESS_STATE
SOFT_INT_STATE
CLOCK_STATE
ALL_STATES        // represents the set of all possible states
```

Other (machine-dependent) symbols correspond to different hardware interrupt sources. Depending on the architecture and implementation, `ADD_SET()` might take the maximum of its arguments (for priority-based masking) or the logical or of its arguments (for selective masking).

5.3. Spin Lock Interface

The `SPIN_LOCK` class has the following interface:

```
SPIN_LOCK::SPIN_LOCK(                 // constructor for a spin lock
      STATE_SET   state_set           // accessed from these states
);

SPIN_LOCK::acquire();                 // acquire the lock

SPIN_LOCK::release();                 // release the lock
```

Spin lock usage is subject to the following rules:

- Spin locks must be released in the reverse order from that in which they were acquired. Failure to do so may leave the processor in a state in which interrupts and/or preemption are impossible.
- Spin locks must be requested in increasing order of execution state set. In other words, if lock *X* is requested while lock *Y* is held, the execution state set of *X* must be a subset of the state set of *Y*.
- There is a static partial order < on the spin locks with a particular execution state set. Processes that acquire multiple locks must do so in strictly increasing order with respect to <.
- A process may not sleep while it holds a spin lock (this can result in poor performance or deadlock).

The last rule prohibits sleeping while a lock is held. However, in some cases releasing a lock before sleeping can lead to race conditions (i.e., if the lock is for the sleep queue). This potentially creates a problem: how to release one or more locks and sleep, in a way that is ''atomic'' with respect to the locks.

This problem has a simple solution in DASH, stemming from the use of software interrupts to invoke the scheduler. Software interrupts (and hence process scheduling) are disabled on a processor while a spin lock is held. Therefore if a process holding spin locks deschedules itself (by calling `SCHEDULER::sleep()` or `SCHEDULER::set_own_deadline()`), it will immediately return from the call, since these functions merely schedule a software interrupt. It will not actually be descheduled until it releases all of its spin locks, which it can do at its leisure. For example:

```
lock1->acquire();          // acquire some locks
lock2->acquire();

// operate under protection of locks

SCHEDULER::sleep();        // schedules software interrupt and returns
lock2->release();
lock2->release();          // context switch takes place here
```

5.4. Spin Lock Implementation

The implementation of spin locks is machine-dependent. Typically a spin lock object contains three pieces of data: a *flag*, a *new mask*, and the *previous mask*. The flag can be one of the following: (1) null for uniprocessors; (2) a test-and-set bit; (3) *N* words such that the lock and the data it protects are not in a same cache line; (4) a pointer to a test-and-set bit (the purpose of this indirection is to separate the lock and the data it protects); (5) a pointer to a word in a memory region with hardware locking support [4].

The *new mask* describes the transition mask to use when the lock is acquired. Usually it is an interrupt bitmask or an interrupt priority level. The *previous mask* stores the the transition mask to be used when the lock is released.

The optimal use of the execution state set by the spin lock implementation may depend on the machine architecture. One extreme is to make a process holding a spin lock non-interruptible (i.e., mask all interrupts while a lock is held). On a machine with a single CPU this policy could cause missed interrupts. On a machine with many processors and symmetric interrupt handling, this policy might reduce lock contention without causing missed interrupts.

6. THE TIMER MECHANISM

This section describes the DASH kernel's timer mechanism. This mechanism allows processes to read the current time, and to schedule *tasks* (short, non-blocking operations) to be performed at specified times. The goal of the mechanism is to provide a high-resolution (1-10 microseconds) readable clock, and low-resolution (1-10 milliseconds) task scheduling. The reasons for this distinction are as follows:

- The readable clock is used for scheduling resources (processor and network interfaces) for units of time that may be very small. Typically, a process or network packet is given a *deadline* that is computed as the current real time plus a delay. For this to be meaningful, an accurate and high-resolution current real time must be obtainable.

- The task-scheduling facility is used for network protocol retransmission and time-slicing, neither of which has stringent timing requirements. For simplicity, the facility uses a periodic clock interrupt, and to minimize interrupt-handling overhead the clock period must be fairly long.

Tasks are performed asynchronously in the context of whatever process is running at that time. Task execution has priority over process execution, so tasks should be short. If a piece of work is time-consuming, it should be done by a process that is awakened by a timer task, not by the timer task itself.

Each task is represented by a *timer request* object. Timer request objects are of classes (such as `PROCESS` and `MESSAGE`) derived from the base class `TIMER_REQUEST`, which has the following public data:

```
ABS_TIME deadline;
void handler_function();
void* handler_this
BOOLEAN pending;
```

`Deadline` is the real time at which the task is to be performed. `Handler_function` is a pointer to the function to be called at the due time, with a pointer to the `TIMER_REQUEST` object as an argument. This procedure is responsible for freeing the `TIMER_REQUEST` object, if necessary. `Handler_this` is a pointer to the object on which the handler function is to be invoked. `Pending` is true iff the request is currently pending (requested but not yet performed).

`TIMER_REQUEST` objects provide the following interface:

```
TIMER_REQUEST::request();

BOOLEAN
TIMER_REQUEST::cancel();
```

`TIMER_REQUEST::request()` issues a timer request; the due time is contained within the `TIMER_REQUEST` object. The object may already be pending; i.e., the operation can be used to change the due time of an existing request. `TIMER_REQUEST::cancel()` cancels a previous request. It returns the pending flag prior to the cancel (i.e., whether the task was done or not).

6.1. Timer Mechanism Implementation

The timer mechanism is implemented using a timer request queue (`TRQ`) object, which stores the set of pending timer requests. The `TRQ` object is implemented as an array of lists of `TIMER_REQUEST` objects. The array is indexed by a group of bits in the due time. This structure is similar to the ''extended timing wheel'' structure described in [12].

`TIMER_REQUEST::request()` is implemented as follows:

```
1)  Lock the TRQ object, then the TIMER_REQUEST object.
2)  If the TIMER_REQUEST object is pending, unlink it from the TRQ.
3)  Insert the TIMER_REQUEST object in the TRQ.
4)  Unlock the TIMER_REQUEST and TRQ objects.
```

`TIMER_REQUEST::cancel()` is implemented as follows:

```
1)  Lock the TRQ object, then the TIMER_REQUEST object.
2)  If the TIMER_REQUEST is pending, remove it from the TRQ.
3)  Unlock the TIMER_REQUEST and TRQ objects.
```

The clock interrupt handler does the following:

```
1)  Lock the TRQ.
2)  Increment the ``current time'' index.
3)  If the table entry for the current time is empty, return.
4)  Scan the list at the current time index,
    performing those requests scheduled for the current time.
5)  Perform each due request as follows:
    a) Lock the TIMER_REQUEST object.
    b) If it is still pending, call the handler function
       with the ``this'' pointer and TIMER_REQUEST object address as arguments
    c) Clear the pending flag and unlock the TIMER_REQUEST object.
6) unlock the TRQ.
```

6.2. Uses of the Timer Mechanism

We now describe some classes derived from `TIMER_REQUEST`, and their uses in the DASH kernel.

`DELAYED_CALL`

These objects provide *delayed procedure calls*. Their `handler_function` points to the procedure to be executed, and additional arguments are stored in the object. Delayed procedure calls are used by the scheduler to implement time-slicing. Each processor has a delayed procedure call request object. When a time-sliced process is switched to, a request is made. When a context switch (preemptive or not) is done from a time-sliced process, the request is canceled. The handler function calls `SCHEDULER::set_own_deadline()` to preempt the process.

`MESSAGE`

A *delayed message-passing operation* is done by using the `MESSAGE` object as a `TIMER_REQUEST` object. The `handler_this` field of the message points to a message-passing object (see Section 7) and the `handler_function` field points to a member function of that object.

`PROCESS`

This is used for `SCHEDULER::timed_sleep()`, which is implemented as follows: the state of caller is changed to `TIMED_SLEEPING`, and `TIMER_REQUEST::request()` is performed on the caller's `PROCESS` object. `Handler_this` is the `SCHEDULER` object, and `handler_function` is a private member function that awakens the process.

7. MESSAGE-PASSING OBJECTS AND OPERATIONS

The DASH kernel's message-passing (MP) system plays several roles:

(1) It provides interprocess communication (IPC) between different processes, and between interrupt routines and processes, on a single host.

(2) It provides an interface for control and data transfer for a single process as it moves between address spaces. For example, system calls, during which a process switches from user to kernel mode and back, use MP (see Section 11).

(3) It provides allocation and queueing of data buffers. This is used by network interface drivers and the virtual memory system [15] to manage buffer pools.

(4) It is used to implement other synchronization mechanisms such as multiple wait, semaphores, sleep locks, and read-write locks (see Section 13).

The MP system is not used directly for network communication. However, it provides the interface between the various entities (network interface drivers, subtransport layer, transport protocol processes, and user processes) that together support network communication on a host.

MP operations are implemented as member functions of *MP objects* (MPO's). These functions can be called directly only by kernel-mode processes. The same general interface is available for user process MP, but the mechanisms for referring to MPO's and performing operations on them is different; see Section 10.

7.1. Degrees of Freedom in Message-Passing Operations

There are several MPO types, and together they provide a variety of MP semantics. MP operations have the following four binary degrees of freedom:

- **Is the operation in "stream" or "request/reply" mode?**

  In *stream mode*, message flow is unidirectional. The sender may block because of flow control, but does not wait for a receiver to arrive or to deliver a reply. In *request/reply mode*, message exchanges occur in synchronized pairs. The requesting process delivers a request message, and proceeds only when a reply message has been obtained.

- **Is the "receiver" a continuation of the sending process, or is it a different process?**

  The processing of a stream-mode message, or of the request message in a request/reply operation, may be done by either the sending process or by a second process. MP operations are called *uniprocess* and *dual-process* accordingly.

- **Is the sending process executing in user or kernel mode?**

  Kernel processes invoke MP operations by making procedure calls. MP operations can also be invoked from user-level processes. The semantics are essentially the same as for kernel processes, but MP operations are initiated via a trap instruction; the kernel trap handler completes the operation (see Section 10).

- **Is the receiving process executing in user or kernel mode?**

  For both uniprocess and dual-process MP operations, the receive processing may be done in a different mode than that of the sending process. For example, system calls are implemented as uniprocess MP operations that are initiated in user mode and processed in kernel mode.

These four degrees of freedom yield 16 logical combinations. All 16 combinations are possible and potentially useful, but only a subset are realized by currently implemented MPO's (see Table 7.1).

7.2. Message-Passing Classes and Objects

An MPO supports either stream mode or request/reply mode operations. Each mode is represented by a base class (`STREAM_MPO` and `REQ_REPLY_MPO` respectively) whose virtual functions define the operations on MPO's of that mode. Derived classes realize these virtual functions. The base classes have no private data or constructors.

MPO's exist in the kernel address space, and may be dynamically allocated. All MPO's may be accessed from within the kernel (by processes or interrupt routines) by calling their member functions.

MPO's may be *pseudo-permanent* ([15]). Such an MPO may be deleted (and its memory freed) while pointers to it still exist; the invalidity of these pointers is detected when they are next used.

There is no fixed correspondence between processes and MPO's. For example, multiple instances of a server may receive request messages from a single MPO, or a server

mode	*sender*	*receiver*	*# processes*	*examples*
stream	kernel	kernel	1	outgoing RMS endpoint (Section 7.3.6)
stream	kernel	kernel	2	stream port (Section 7.3.5)
stream	kernel	user	1	
stream	kernel	user	2	stream port
stream	user	kernel	1	outgoing RMS endpoint
stream	user	kernel	2	stream port
stream	user	user	1	
stream	user	user	2	stream port
req/reply	kernel	kernel	1	kernel service port (Section 7.4.2)
req/reply	kernel	kernel	2	request/reply port (Section 7.4.2)
req/reply	kernel	user	1	
req/reply	kernel	user	2	user-level service port (Section 7.4.2)
req/reply	user	kernel	1	system call object (Section 7.4.3)
req/reply	user	kernel	2	kernel service port
req/reply	user	user	1	
req/reply	user	user	2	user-level service port

Table 7.1: Varieties of Message-Passing Objects.

process may receive messages from multiple MPO's to get requests from different sources.

The representation of a message, to be described in detail in Section 8, includes space in the header for optional arguments to MP operations. MESSAGE objects contain the following fields used for this purpose (not all options are used for all operations):

```
int          flags;              // selects options (see below)
ABS_TIME     deadline;           // scheduling deadline for receiver
UID          uid;                // UID if MPO is pseudo-permanent
STREAM_MPO*  signal_object;      // if ASYNCHRONOUS, where to send signal message
MESSAGE*     signal_message;     // if ASYNCHRONOUS, the signal message to send
int          client_id;          // identifies the client in request/reply messag
```

The meaning of these fields is explained in subsequent sections. The bitfields in `flags` include the following (these are explained in more detail below).

```
URGENT                  // queue this message LIFO
CHECK_ID                // MPO is pseudo-permanent
DEADLINE                // this message carries a deadline for the receiver
ASYNCHRONOUS            // if operation would block, do it asynchronously
CONDITIONAL             // if operation would block, don't do it
IGNORE_FLOW_CONTROL
ALWAYS_SIGNAL           // with ASYNCHRONOUS, send signal message
                        // even if the operation is done immediately

INTER_SPACE             // this message is passed between virtual addr spaces
SENDER_TRUSTED          // whether the receiver trusts the sender
IMMEDIATE_USE           // whether the receiver plans to access the data soon
REPLACE_PAGES           // allocate new IPC pages? (user MP only)
```

MP operation return codes include the following:

```
SUCCESS                 // operation completed
PENDING                 // ASYNCHRONOUS operation is queued and pending
WOULD_BLOCK             // a CONDITIONAL operation would block
MPO_DELETED             // pseudo-permanent MPO has been deleted
```

The following structure is use to pass arguments to some MP operations (this is done to reduce parameter-passing overhead):

```
struct MP_BUFFER {
      int          flags;
      MESSAGE*     outgoing_msg;
      MESSAGE*     incoming_msg;
      MESSAGE*     user_header;
};
```

7.3. Stream Mode Message-Passing

The STREAM_MPO base class includes the following virtual functions:

```
int
STREAM_MPO::send(
        MESSAGE*    msg             // the message to be sent
);

int
STREAM_MPO::receive(
        MP_BUFFER*  args
);

int
STREAM_MPO::control(
        int   opcode,               // the operation to be performed
        void* argptr                // additional data (may be input or output)
);
```

All operations return a status code. `STREAM_MPO::receive()` is provided only by dual-process stream MPO's.

Dual-process MPO's may have to enqueue waiting receiver processes or messages waiting to be delivered. The MPO determines the ordering; for example, `STREAM_PORT` objects use FIFO order by default. The `URGENT` flag, used in a `STREAM_MPO::send()` operation, indicates that the message is to be given priority; `STREAM_PORT` objects queue such messages LIFO.

7.3.1. Flow Control

Dual-process stream MPO's (such as `STREAM_PORT` objects) may act as buffers between producer and consumer processes. Operations on such objects can be subjected to *flow control.* The flow control mechanism is as follows. Flow-controlled MPO's include some or all of the following data in their internal state:

Flow control mode: a flag indicating whether flow control is based on number of queued messages (*message mode*) or on the number of data bytes in the queue (*byte mode*).

Current queue length: the ''logical'' length of the message queue. The units of length are either messages or bytes, depending on the mode. When a message is queued on the MPO, the field is incremented by one or by the size of the message.

Maximum queue length: an upper bound on the queue length. A send operation blocks if it would cause this limit to be exceeded.

Sender restart queue length: all blocked senders are unblocked when the queue length falls below this value.

Receiver restart queue length: a read operation blocks if the queue is empty, and is unblocked when its length rises above this number.

Queue size adjustment flag: if true, the queue length is decremented by a `STREAM_MPO::receive()` operation. Otherwise it must be decremented explicitly by a `STREAM_MPO::control()` operation with the `REDUCE_QUEUE_LENGTH` opcode.

This mechanism provides hysteresis for both the sender and the receiver. Hysteresis can reduce the number of context switches, since a process can handle a batch of messages in one context switch. If this hysteresis is undesirable, the restart lengths can be set to the maximum length and to zero, respectively. When no more messages are to be written to an MPO, reader hysteresis must be turned off using the `STREAM_MPO::control()` operation.

This flow control mechanism is intended to allow transport protocols to control the flow of data from client user processes. Since interrupt handlers can never block, a stream MPO written to by an interrupt handler should not use flow control.

The mechanism for initializing the above parameters depends on the MP class; it may use the class constructor or a class-specific control operation. See, for example, the `STREAM_PORT` class (Section 7.3.5). Flow-controlled stream MPO's may supply a `STREAM_MPO::control()` operation with opcode `REDUCE_QUEUE_LENGTH`, which reduces the queue length by the value of `argptr` (which is an integer in this case). Flow control is bypassed if the `IGNORE_FLOW_CONTROL` flag is set in `STREAM_MPO::send()` and `STREAM_MPO::receive()` operations.

7.3.2. Synchronization

Dual-process stream mode MP is asynchronous in the sense that a `STREAM_MPO::send()` operation doesn't wait for a reply to arrive, or for a matching `STREAM_MPO::receive()` operation to be performed. However, stream mode MP operations may cause the process to block, either because the queue is full (`STREAM_MPO::send()`) or because no message is available (`STREAM_MPO::receive()`). Stream mode MP operations that can block have several synchronization alternatives. The following flags are used:

> `CONDITIONAL`: if the request cannot be performed without blocking, the operation returns immediately with the `WOULD_BLOCK` code.
>
> `ASYNCHRONOUS`: this allows operations to be performed asynchronously. The requester supplies a *signal object* (a stream MPO) and a message to be sent to the signal object when the original request finishes. An error code is included in the header of the signal message. If the signal message cannot be sent immediately (e.g., because the signal object was deleted, or because of flow control on the signal object) at the point when the main request succeeds or fails, then the main operation will be canceled.
>
> `ALWAYS_SIGNAL` (`ASYNCHRONOUS` mode only): send the signal message even if the main operation can be done immediately (i.e., without blocking). If the flag is absent and the main operation can be done immediately, the call returns `SUCCESS` and does not send the signal message.

The default synchronization is ''blocking'' mode: if the request cannot be performed immediately, the process blocks, and is awakened when the request succeeds or fails. This mode is indicated by the absence of the `CONDITIONAL` and `ASYNCHRONOUS` flags.

7.3.3. MPO Deletion

The following `STREAM_MPO::control()` operations may be done on dual-process stream MPO's:

> `DELETE`: atomically frees the MPO and returns (via `argptr`) a list of messages queued on it. If any operations are pending on the MPO, an error code is returned

to the blocked processes.

DELAYED_DELETE: if at least one message is enqueued at the MPO, it is deleted as above. If no messages are queued, the MPO is put into a *delete pending* state. Exactly one more message can be sent to the MPO, and the MPO is deleted atomically when that message is received. This operation makes it possible to atomically do a blocking read from a stream MPO and delete it. This can be used to implement an ''atomic select'' operation; see Section 13.

7.3.4. Assignment of Deadlines

A stream MP operation may associate a deadline with the message being sent. This is used as a scheduling deadline in the processing of the message. Call the message deadline *X*. In uniprocess operations, the caller's deadline is changed to the minimum of its current deadline and *X*; it is restored on completion of the operation. In dual-process operations, the receiver's deadline is changed to *X*.

The deadline *X* may be assigned in the following ways. If, in the STREAM_MPO::send() operation, the DEADLINE flag is set in the flags field of the message header, then the deadline field of the message header is the explicit deadline. Alternatively, a stream MPO can be assigned a *deadline offset* by a STREAM_MPO::control() operation. When a message is sent to such an MPO, the deadline is computed as the current real time plus this offset.

7.3.5. Stream Ports

Stream ports (objects of class STREAM_PORT) are dual-process stream MPO's. In terms of implementation, they provide queues for messages, for waiting receivers, and perhaps for waiting senders. The constructor is

```
STREAM_PORT::STREAM_PORT(
        STATE_SET   state_set
);
```

where state_set specifies the set of execution states (see Section 5) that may access the port. A stream port initially has no flow control. Flow control is enabled, and its parameters are set, by STREAM_PORT::control() operations.

7.3.6. Outgoing RMS Endpoints

Classes OUT_ST_RMS or OUT_NET_RMS are derived from STREAM_MPO. An object of one of these classes is called an *outgoing RMS endpoint object.* Such an object represents an endpoint of a Real-Time Message Stream (RMS) [14]. When STREAM_MPO::send() is performed on an RMS endpoint object, the message is eventually sent via a network to a remote host. In the current implementation, RMS endpoint objects are uniprocess; whatever work is necessary to send the message (fragmentation, checksumming, queueing and so forth) is done in the caller's process.

Clients of an RMS must obey a bound on outstanding data on the RMS. This suggests variants of RMS objects that supply this flow control automatically. One way to do this is with reverse acknowledgements; another is with rate-based flow control. Both approaches can be implemented as MPO's (rather than as full-fledged protocols) since they only have to perform actions on send requests.

7.4. Request/Reply Mode Message-Passing

The REQ_REPLY_MPO base class has the following virtual functions:

```
int
REQ_REPLY_MPO::request_reply(
        MP_BUFFER*  args
);

int
REQ_REPLY_MPO::get_request(
        MP_BUFFER*  args
);

int
REQ_REPLY_MPO::send_reply(
        MESSAGE     *msg
);

int
REQ_REPLY_MPO::control(
        int         opcode,
        void*       argptr
);
```

A client calls REQ_REPLY_MPO::request_reply(), supplying a request message and receiving a reply message in return. If the MPO is uniprocess, the operation is done by the client process itself. If the MPO is dual-process, a *server process* obtains the request message using REQ_REPLY_MPO::get_request(), performs the operation, and delivers a reply message using REQ_REPLY_MPO::send_reply().

Because multiple concurrent operations on a request/reply MPO can potentially take place, it is necessary for REQ_REPLY_MPO::send_reply() to identify the client to whom the reply is being sent. This is done as follows: REQ_REPLY_MPO::request_reply() stores a value in the client_id field of the request message. The server must copy this field to the reply message. REQ_REPLY_MPO::send_reply() uses it to determine which client to wake up.

If the server is a kernel process, the client ID is a pointer to the PROCESS object of the client. If the server is a user process, the ID is a temporary "user object reference" (see Section 9) to the PROCESS object of the client.

Because of the synchrony of request/reply operations, there is no need for flow control on request/reply MPO's. Nor is there a need for additional synchronization options such as CONDITIONAL or ASYNCHRONOUS.

7.4.1. Assignment of Deadlines

A request/reply MP operation may provide a *deadline*, i.e., the time by which the operation should be completed. This is used as a process scheduling deadline. Call the message deadline X. In uniprocess operations, the client's deadline is changed to the minimum of its current deadline and X; it is restored on completion of the operation. In dual-process operations, the server's deadline is changed to X.

The deadline X may be assigned in several ways. If, in the REQ_REPLY_MPO::request_reply() operation, the DEADLINE flag is set in the `flags` field of the message header, then the `deadline` field of the message header is the explicit deadline. Alternatively, some request/reply MPO's can be assigned a *deadline offset* by a `REQ_REPLY_MPO::control()` operation. When a message is sent to such an MPO, the deadline is computed as the current real time plus this offset.

7.4.2. Request/Reply Port

The class `REQ_REPLY_PORT` defines the basic dual-process request/reply MPO. The object queues clients when there are no available servers, and queues servers when there are no requests. Clients and servers may be either user or kernel processes. This type of MPO might be used for a kernel-resident server.

7.4.3. System Call Object

The `SYSTEM_CALL` object is a uniprocess request/reply MPO used to make system calls from user processes (see Section 11). There is one such object per kernel; it has no internal state and therefore requires no synchronization.

8. MESSAGE REPRESENTATION, ALLOCATION AND MANIPULATION

This section describes the message representation part of the MP system. A *message* is an object of class `MESSAGE`. This class provides the abstraction of an untyped byte array. The design of the message representation is targeted at two cases: (1) moving large messages between VAS's; (2) protocol handling, such as header insertion/deletion, copying for retransmission, and fragmentation.

Message data cannot, in general, be randomly accessed by array indexing. It is accessible only through the set of operations described below. The implementation of message objects does not necessarily store the data contiguously in memory; indeed, the data may not even be mapped in the VAS of the message owner.

This encapsulation of message structure gives the MP system considerable freedom in representing and transporting messages. The MP system uses this freedom to avoid unnecessary memory copying when moving data between VAS's; this is done by using virtual memory remapping [15].

A `MESSAGE` object is implemented as a variable-size *header block* and a set of IPC pages. The header block contains an array of *descriptors* pointing to the IPC pages. Each descriptor contains the following information: 1) a pointer to the IPC page; 2) a read/write flag; 3) the offset and size of data in the page.

The header block also contains other bookkeeping information (such as the total message size) and space for parameters to MP operations (see Section 7.2). Figure 8.1 shows the structure of a message object.

8.1. Operations on Message Objects

There are three constructors for `MESSAGE` objects:

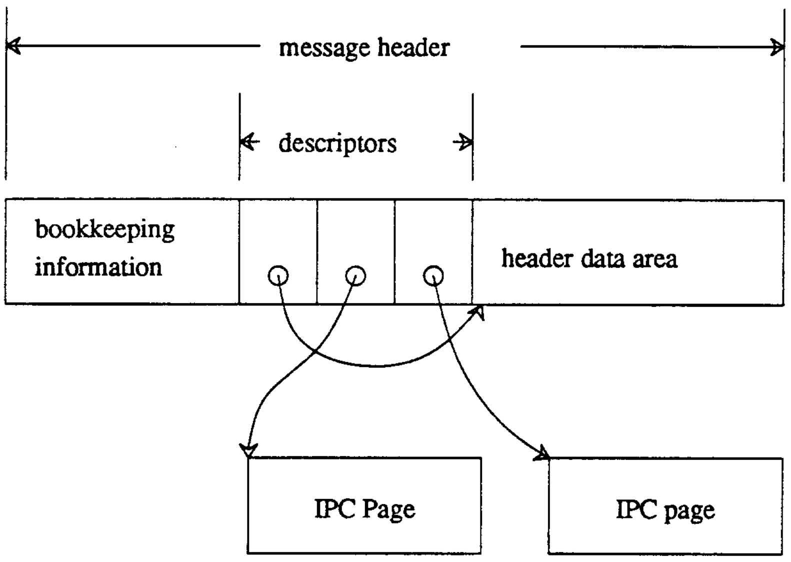

Figure 8.1: Message Representation.

```
MESSAGE::MESSAGE(
      int         head_size,
      int         body_size,
      int         tail_size,
      VIRT_ADDR*  data
);

MESSAGE::MESSAGE(
      int    head_size,
      int    tail_size
);

MESSAGE::MESSAGE();
```

In the first form, `body_size` is the initial size (number of data bytes) of the message. `Head_size` and `tail_size` are the maximum size of protocol headers and trailers to be added to the message. These values must be obtained from the protocol layers through which the message will pass. A pointer to the initial data part of the message is returned in `data`.

In the second form, there is no initial data in the message. This is used when the message is to be formed from IPC pages.

In the third form, the message will never be used to contain data. This is used for signal messages, which may use the data fields in the message header.

```
VIRT_ADDR
MESSAGE::access(
      BOOLEAN     rw,
      int         offset,
      int         *contiguous_size
);

VIRT_ADDR
MESSAGE::access(
      BOOLEAN     rw,
      int         offset,
      int         length
);
```

Both forms return a pointer to the message data starting at the given `offset`. If `rw` is true, the data can be read or written; otherwise it can only be read. In the first form, the amount of contiguous data is returned in `contiguous_size`. In the second form, the message will be reorganized, if necessary, to make the requested amount of data contiguous.

The following functions add data to a message:

```
MESSAGE::append_message(
      MESSAGE*    msg
);

MESSAGE::append_ipc_page(
      VIRT_ADDR   data,
      int         size
);

MESSAGE::prepend_ipc_page(
      VIRT_ADDR   data,
      int         size
);

MESSAGE::append_space(
      int         length,
      VIRT_ADDR   data
);

MESSAGE::prepend_space(
      int         length,
      VIRT_ADDR   data
);
```

`MESSAGE::append_message()` appends another message to this one. `MESSAGE::prepend_ipc_page()` and `MESSAGE::append_ipc_page()` add an IPC page to the start or end of this message, respectively. `Data` points to the start of the data (which is not necessarily at the start of the IPC page). `MESSAGE::prepend_space()` and `MESSAGE::append_space()` add uninitialized data to the start or end of this message. This space must have been previously reserved by the `head` and `tail` arguments to the message constructor.

The following functions remove data from a message:

```
MESSAGE::remove_start(
        int    length
);

MESSAGE::remove_end(
        int    length
);
```

These remove the given amount of data from the start or end of this message. There is no provision for inserting or deleting data in the middle of a message.

```
MESSAGE*
MESSAGE::duplicate();
```

This produces a separate "logical copy" of this message; (e.g., for protocol retransmission). This is done by 1) copying the header and 2) incrementing the reference count of the message's IPC pages, and making the references read-only (see Section 8.3). Each copy can be independently deallocated or sent to other VAS's. Writing to one copy (after using `MESSAGE::access()`) will not affect the other copy.

```
MESSAGE*
MESSAGE::split(
        int    offset,
);
```

This separates the message into two parts around the given `offset` (e.g., to fragment a large message into network packets). As with `MESSAGE::duplicate()`, copying of IPC pages is avoided.

The `MESSAGE` class also provides member functions `MESSAGE::start_transfer()` and `MESSAGE::finish_transfer()` that transfer messages between VAS's (see Section 10.5).

8.2. Synchronization of Message Operations

Message access operations are not synchronized (e.g., there is no per-message spin lock). It is the responsibility of the clients to avoid concurrent operations. Typically, a single process is responsible for a message at any point. This responsibility is transferred by the MP operations: the sender loses responsibility, and the receiver acquires it.

8.3. Data Sharing Between Messages

Messages can share IPC pages. However, there is always a separate header for each message. There is no lock to synchronize access to the header; this is the responsibility of clients, as indicated above. On the other hand, an IPC page may be shared by two messages and pointed to by multiple descriptors. The VM system ([15]) maintains reference counts for the IPC pages. Operations on these reference counts are serialized by the VM system.

A descriptor contains a flag indicating the access mode (read/write or read-only) of the IPC page. A IPC page is shared only in read-only mode. Since there is at most one writer, it is not necessary to synchronize the access to the data part of a message.

An IPC page may be shared by multiple messages as a result of `MESSAGE::duplicate()` or `MESSAGE::split()`. In the second case, the two resulting message headers have descriptors pointing to the same IPC page, but with different offsets. The read/write flag of a shared IPC page is always `read_only`.

Sharing reduces the overhead of `MESSAGE::duplicate()` and `MESSAGE::split()`. Only a new header block need be allocated; the existing IPC pages can be used without copying.

A message containing shared IPC pages may be modified without affecting other messages sharing the IPC pages. This is possible because modifying a message is allowed only after calling `MESSAGE::access()`, which makes a private copy of the IPC page if the flag is `read_only`. This is essentially a *copy-on-write* mechanism [1] with a fine granularity and without page faults. `MESSAGE::duplicate()` and `MESSAGE::split()` are used mostly by protocols for purposes of retransmission, and subsequent message modification is therefore infrequent.

8.4. An Example

The following example describes the work involved in sending a large message from a user VAS, through the kernel, and out onto a network. Observe that IPC page data is never copied.

(1) The user process allocates IPC pages and creates a message consisting of a header and several IPC pages. It reserves space for protocol headers and trailers in the message header (the amount of space may depend on the protocols being used; if necessary, the protocols can be queried).

(2) The user sends a message to the kernel. The message header is copied into the kernel VAS and ownership of the IPC pages is transferred to the kernel VAS.

(3) Protocol headers and/or trailers are added to the message. Because of (1), they fit within the message header.

(4) The message is duplicated for retransmission. A new message header is allocated, and the reference counts of the IPC pages are adjusted.

(5) The message is split into several small message of network packet size. New message headers are allocated, but IPC pages are not copied.

(6) The packet-size messages are mapped into the I/O space if necessary, If the network interface supports chaining-mode DMA, the packets can be sent without copying the IPC page data.

9. USER OBJECT REFERENCES

The *user object reference* (UOR) mechanism allows user processes to obtain "capabilities" to kernel-level objects. UOR's are issued (by the kernel) to a particular user virtual address space (VAS), and can be used by any process in that VAS. They exist only for the life of the VAS.

For each user VAS, the kernel maintains a table describing the UOR's allocated to that VAS. A UOR is an *(index, UID)* pair. The *index* is an index into the table for that VAS. The *UID* is used to identify a *pseudo-permanent* object ([15]), and is used only for such objects.

Each entry in a UOR table includes:

- The address (in the kernel VAS) of the object.
- The type of the object, encoded as an integer.
- A bitmask representing a set of access rights to the object (i.e., operations that can be performed on it). The meaning of these bits is class-specific. Some classes may not use them at all.
- A UID (pseudo-permanent objects only).

User processes pass UOR's to the kernel during MP operations, both to refer to the MPO, and (for system calls) to refer to other objects. Whenever a UOR is resolved to an object address, the following validity check is done:

> If the UID in the table is nonzero, it must match that in the UOR. This check is for security purposes; it does not detect dangling references.

In addition, the kernel routines that directly handle user requests (MP processing and system call interface routines) may perform the following checks:

- The type field in the UOR entry is checked for correctness.
- Individual object operations may check the access-rights bitmask.
- The object UID is compared with the UOR's UID. This detects dangling references to deleted objects.

Some UOR's are allocated on VAS creation, while others are allocated via system calls. The initial (and permanent) entries in the table include references to

- the system-call object (see Section 7.4.3);
- the exception port for this VAS (see Section 12).;
- the `VAS` object for this VAS (see [15])
- the `NAMED_ENTITY` object representing the root of the global name space (see [14]).

The implementation of the UOR's uses the class `UOR_TABLE`, which implements a UOR table. It provides the following interface:

```
struct UOR_ENTRY {
      OBJECT_TYPE type;
      U32         access_rights;
      UID         uid;
      void*       object;
};

U32
UOR_TABLE::add_entry (
      OBJECT_TYPE type,
      U32         access_rights,
      void*       object,          // object address
      UID         uid
);
```

```
UOR_ENTRY*
UOR_TABLE::get_entry (
        U32             index,
        UID             uid
);

UOR_TABLE::delete_entry (
        U32             index
);
```

`UOR_TABLE::add_entry()` creates a UOR for the given object, returning the table index. `UOR_TABLE::get_entry()` returns a pointer to the UOR table entry referred to by the given UOR. It returns `NULL` if the index is unassigned or if the UID's do not match. `UOR_TABLE::delete_entry()` frees a UOR table entry.

10. MESSAGE PASSING BETWEEN VIRTUAL ADDRESS SPACES

The MP operations described in Section 7 are available to user-level processes, using the mechanism described in this section. This mechanism is used for (1) user-to-kernel, (2) user-to-user in different VAS's, and (3) user-to-user in a single VAS. The operations are invoked by trapping into the kernel, even when the sender and receiver are in the same VAS [9]. Input and output parameters of the operations are passed in hardware registers.

10.1. Message Representation and Allocation

The message representation described in Section 8 applies to user-level message-passing as well. A message consists of a variable-size header, which must be contained in a single IPC page, and possibly some IPC pages containing data. These constraints guarantee that 1) the message header is accessible (for copying) in the kernel VAS, and 2) the data pages can be remapped efficiently.

Operations on message objects (e.g., `MESSAGE::access()`) are implemented as user-level library routines. In some cases these routines must make system calls to allocate or free IPC pages. The potential circularity is avoided by having an IPC page that is permanently owned by the VAS, for the purpose of making system calls.

To receive a message, a user process must supply space for a message header in an IPC page, to be filled in by the MP operation. The message data is stored either in IPC pages or in the available space in the message header.

10.2. Message Ownership

The semantics of message-passing are based on the notion of IPC page ownership described in [15]. They can be summarized as follows:

- ''Ownership'' of a message implies ownership of the message's IPC data pages, and of the IPC page containing the header.
- When a message is sent, ownership of the IPC data pages is transferred from the VAS of the sender to that of the receiver. The sender VAS retains ownership of the

[9] It would also possible to implement MP between processes in the same VAS as user-level library routines, plus system calls for process control operations. This approach would save a trap into the kernel in simple cases, but would require more traps when an MP operation involves several process operations (e.g., when multiple processes are awakened because of flow control).

IPC page containing the header.

- A user process can access an IPC page only if its VAS owns the page. It can write to the page only if the ownership is exclusive.

10.3. Binding of MP Objects to VAS's

A message-passing object to be used for message-passing between VAS's is always *bound* to a particular VAS. This allows the MP system to transfer a message on the `send` operation, even when no process is waiting to receive it. A VAS is bound to an MPO using `STREAM_MPO:control()` or `REQ_REPLY_MPO::control()` with `BIND_SPACE` as the opcode. Only processes in the bound VAS can perform `STREAM_MPO::receive()` or `REQ_REPLY_MPO::get_request()` operations on the MPO.

10.4. Arguments to the MP Operations

Each MP operation requires that certain information be passed in hardware registers:

- All operations require a code identifying the operation, a UOR to the MP object, and the UID of the MP object.
- Operations that send a message (`STREAM_MPO::send()`, `REQ_REPLY_MPO::request_reply()`, and `REQ_REPLY_MPO::send_reply()`) require a pointer to the message to be sent. The message header contains flags, including a `REPLACE_PAGES` flag indicating that new IPC pages are to be allocated to replace those sent out in the message. Pointers to the new pages are returned in the message header descriptors.
- Operations that receive a message (`STREAM_MPO::receive()`, `REQ_REPLY_MPO::request_reply()`, and `REQ_REPLY_MPO::get_request()`) require a pointer to a message header, and the flags for the receive operation.
- `STREAM_MPO::control()` and `REQ_REPLY::control()` require the opcode of the particular control operation, and possibly a word of additional information.

10.5. Message Transfer Operations

Moving a message from one VAS to another is done by member functions of the `MESSAGE` class. A message contains a header and possibly IPC pages. The header is moved by software copying, whereas IPC pages are moved by VM remapping [15]. The VM system remaps an IPC page from one VAS to another in two steps: one initiates the operation, and one ensures that the operation is completed. Moving a message is also divided into two functions to reflect the remapping facility provided by the VM system. This section describes these functions; the next section describes the scenario of using these functions. The first function is called by the sender.

```
MESSAGE::start_transfer (
      VAS*          sender,            // source VAS
      VAS*          receiver,          // destination VAS
      MESSAGE**     new_header,
      BOOLEAN       sender_trusted
);
```

This transfers a message from one VAS to another. New_header points to a message pointer. If on input the latter pointer is NULL, then there is no waiting receiver. In this case, a temporary buffer is allocated in the kernel VAS, the message header is copied there, and the pointer is returned via new_header. If the pointer is not NULL, it points to the header supplied by a waiting receiver, and the header is directly copied there. In addition, this function checks the validity of the message (i.e. that all the IPC pages are owned by the VAS). For every IPC page in the message, it calls IPC_REGION_MGR::start_transfer() of the VM system to transfer the ownership of the page from the sender VAS to the receiver VAS, passing the sender_trusted flag directly.

The second function is called by the receiving process to ensure that the transfer is complete.

```
MESSAGE::finish_transfer (
      VAS*          receiver,          // destination VAS
      MESSAGE*      receiver_header,   // receiver's header
      BOOLEAN       sender_trusted,    // whether to purify
      BOOLEAN       immediate_map      // whether to hardware-map
);
```

It does the following:

- If the message header is in a temporary kernel buffer (this occurs, for example, when STREAM_MPO::send() is called before STREAM_MPO::receive()), it copies the message header into the receiver's header, and deallocates the buffer.
- For every IPC page in the message, it calls IPC_REGION_MGR::finish_transfer of the VM system to ensure that the IPC page has been transferred properly. The two boolean flags are passed directly.

10.6. Stream Mode MP Scenarios

This section describes the steps taken in sending a message between VAS's using a pair of STREAM_MPO::send() and STREAM_MPO::receive() operations. The details differ somewhat according whether:

(1) the STREAM_MPO::receive() precedes the STREAM_MPO::send(), and the receiving process sleeps;

(2) the STREAM_MPO::send() precedes the STREAM_MPO::receive(), and the message is enqueued.

The ordering of the two operations (which may be concurrent) is determined by the order in which they acquire the spin lock on the MPO.

10.6.1. Receiver First

The steps executed by the receiver process are:

(1) It stores the MP operation arguments in machine registers and executes a TRAP instruction.

(2) The machine-language trap handler routine pushes the values in the registers onto the stack and calls the C++ trap handler.

(3) The C++ trap handler switches to the kernel VAS. It checks the UOR to the MP object, including the rights to perform the MP operation.

(4) The trap handler then calls `STREAM_MPO::receive` on the MPO. This acquires the spin lock that is used to serialize operations on the MPO. Since there is no message queued, the receiver sleeps and releases the spin lock. Before doing this it stores the receive flags, and a pointer to the receiver header, in its context block.

(5) After the process is awakened by the sender (see below) it returns from `STREAM_MPO::receive()` to the trap handler. `STREAM_MPO::receive()` returns a pointer to the message received by the MPO. The trap handler calls `MESSAGE::finish_transfer()` to ensure the message has been transferred to the receiver VAS properly.

(6) The trap handler switches back to the user VAS and does a ''return from trap'' instruction.

The sender executes steps (1) to (3) as above. Then the trap hander calls `STREAM_MPO::send()` on the MPO. Since the corresponding `STREAM_MPO::receive()` has been called, the message-passing system knows at this point the VAS of the receiver, the address of the receive header, and the receive flags (i.e., `SENDER_TRUSTED`, and `IMMEDIATE_USE`). The `STREAM_MPO::send()` operation does the following:

(1) It calls `MESSAGE::start_transfer()` to move the message to the receive VAS.

(2) It wakes up the receiver, which resumes at step (5) above.

(3) It returns to the trap handler, which in turn returns to the user VAS.

10.6.2. Sender First

In this case, when `STREAM_MPO::send()` is called, the message-passing system does not know the receiver header and the receive flags. However, it does know the receive VAS because the VAS has been bound to the MPO. It uses the receive VAS, a `NULL` receive header and the flags of the sender to call `MESSAGE::start_transfer()`. This copies the message header to a temporary kernel buffer, and remaps IPC pages in the message to the receiver VAS. Then the message stored in the kernel buffer is enqueued, and `STREAM_MPO::send()` returns to the trap handler.

At some later point a receiver invokes `STREAM_MPO::receive()` and finds a message in the MPO queue. It calls `MESSAGE::finish_transfer()` to copy the header from the temporary kernel buffer, and complete the remapping of the IPC pages.

10.7. Request-Reply Mode MP Operations

A request/reply MP operation is basically a pair of stream-mode MP operations; the request message is sent from the client to the server, and the reply message is sent from the server to the client. The implementation parallels that described above for stream-mode operations. The following simplifications, however, are made: Since the client is always blocked waiting for the reply message, the transfer of the reply message always follows the ''receiver first'' scenario.

11. SYSTEM CALLS

This section describes the DASH system call mechanism, which is based on user-to-kernel request/reply message passing. System calls are done by executing a `request_reply()` operation on a uniprocess request/reply MPO called the *system call object*. Every address space has a well-known UOR to the system call object.

A system call request message consists of an operation code followed by data. The system call object includes a table (indexed by operation code) of pointers to *interface routines*. Each interface routine is logically part of a kernel class; it is responsible for converting a request/reply operation into a call to a member function of the class. The system call object decodes the request message and forwards it (as a conventional object operation) to the appropriate interface routine.

11.1. System Call Scenario

The steps in executing a system call are as follows:

(1) A user process performs a `REQ_REPLY_MPO::request_reply()` operation on the system call object, resulting in a trap to the kernel.

(2) The system call object checks the validity of the operation code, and branches to the relevant interface routine.

(3) The interface routine checks the validity of the arguments contained in the request message. If the arguments include UOR's, the interface routine converts them to kernel object pointers and verifies type correctness and access rights. The routine then calls a standard member function of the kernel object.

(4) When the member function returns, the interface routine packages the return values into a message. If needed, pointers to kernel objects are translated to UOR's.

(5) The interface routine returns to the system call object, which returns to the user VAS via the trap handler.

11.2. Defining a System Call

Each system call defines a structure for the request and reply messages. The message specification language DML (see [14]) is used for defining these structures. A typical definition is:

```
msg_typedef struct {
      U32   request_code;
      BYTES owner_name;        // owner name whose public key is requested
} GET_PUBLIC_KEY_REQ;

msg_typedef struct {
      U32   return_code;       // return code
      BYTES key;               // the key
} GET_PUBLIC_KEY_REP;
```

The following facilities could be provided to user-level programmers to facilitate system calls:

- An include file defining the operation codes and predefined UOR's (for the system call object, the current address space, and others) used in making system calls.
- A library of routines for creating and accessing system call messages.
- A library of routines that give a procedural interface to system calls.

For kernel implementors, defining a system call consists of the following steps:

(1) Register a system call operation code.

(2) Define the structures of the request and the reply messages.

(3) Write an interface routine that performs the tasks described above.

(4) Add an entry to the interface routine table of the system call object.

12. EXCEPTIONS

Each user VAS has an associated *exception MPO* (a `STREAM_MPO` object) which has a well-known UOR. Exception sources include illegal instructions, floating-point errors (e.g., divide by zero), writes to read-only pages, references to unallocated pages, references to unassociated pages, user-requested VM exceptions, etc. When a user process incurs an exception, a message with the following format is written to the exception MPO:

```
typedef struct {
        USER_OBJ_REF        process;     // reference to offending process
        VIRT_ADDR           instr;       // address of offending instruction
        EXC_TYPE            type;        // type of exception
        U32                 data[8];     // data specific to a type of exception,
                                         // e.g., the reference addr for VM errors
} EXCEPTION_MSG;
```

When a process generates an exception, it enters the `SLEEPING` state, and does not execute until explicitly resumed. Each VAS is responsible for processing its exception messages. This is normally done by having a *exception handler process* that executes an infinite loop reading and processing messages from the exception MPO. If a message remains unread in the exception MPO for more than some fixed period, the VAS and its processes are assumed to have crashed, and are deleted.

The exception handler may resume the faulting process after eliminating the reason for the exception, kill the faulting process gracefully, or kill the whole space gracefully. The following two functions are available to the exception handler through system calls.

```
CONTEXT
PROCESS::read_context();

void
PROCESS:write_context(CONTEXT*);
```

The structure of `CONTEXT` is machine-specific. It stores the state, e.g., registers, of a process. By manipulating the context block as well as the VAS of a process, the exception handler can repeat an instruction, skip an instruction, simulate an instruction, unwind the stack of a process, perform a long jump across stack frames, etc.

13. MESSAGE-PASSING: EXAMPLES

Many high level synchronization mechanisms can be implemented using the MP facility. Some examples of high level synchronization are presented in this section; they are not intended to be complete nor rigorous, but to point out possibilities and ideas.

13.1. Multiple Wait and Timeout

This example allows a process to wait on multiple events and return when the first one is done. It guarantees that exactly one event is done. [10]

(1) Create a `STREAM_PORT` object (call it `sigport`) to receive a signal message that indicates an event has happened.

(2) `sigport->control(DELAYED_DELETE)` (This port will accept only one message).

(3) Issue as many `STREAM_MPO::send()` and `STREAM_MPO::receive()` operations as necessary. Each operation includes the `ASYNCHRONOUS` and `ALWAYS_SIGNAL` flags, and specifies `sigport` as the port to which the signal message is to be sent.

(4) Receive a signal message from `sigport` in `BLOCKING` mode. When done, examine the signal message to determine the source of the event.

Step 2 ensures that only one operation issued in Step 3 will successfully deliver the signal message; therefore, it ensures that only one operation will succeed. The pseudo-permanence of the signal port takes care of the case when the signal port has already been deleted.

Timeout on multiple wait can be done by arranging a message to be delivered to the signal port after some delay time. The process, if still blocked in Step 4, will be awakened when the timeout message arrives.

In many situations it is not necessary to guarantee that exactly one event occurs, e.g., a server waiting on multiple sources of requests. The `STREAM_MPO::control()` operation in Step 2 can be removed:

(1) Create a `STREAM_PORT` object, call it `sigport`.

(2) Issue initial `STREAM_MPO::send()` and `STREAM_MPO::receive()` operations. Each operation includes the `ASYNCHRONOUS` and `ALWAYS_SIGNAL` flags, and specifies `sigport`.

(3) Receive a signal message from `sigport` in `BLOCKING` mode. Examine the signal message to determine the source of event.

(4) Process the event, and issue a new request on the port where the previous operation succeeded. Loop back to Step 3.

---

[10] Multiple wait is achieved by the `select()` system call in UNIX. However, there is no guarantee that an I/O system call on a "ready" descriptor immediately after the return from `select()` will succeed without blocking, since other processes may read from or write to the descriptor between the `select()` and the I/O system call.

13.2. Sleep Locks (Binary Semaphores)

At most one process can hold the lock at any time. If the lock is not available the requesting process is put to sleep. Such a lock can be implemented as a port and a "token message". The token message is written to the port when the lock is created. The lock is acquired by receiving from the port, and released by writing to the port. Whoever owns the token message holds the lock. The various synchronization modes of the `STREAM_MPO::receive()` operation (conditional, blocking and asynchronous) are automatically available when acquiring the lock. Other forms of sleep locks, such as single-writer multiple-reader locks, can be implemented similarly.

13.3. Producer/Consumer Synchronization

Producer/consumer synchronization can be implemented using a flow-controlled stream port:

(1) Create a `STREAM_PORT` with flow control enabled, and with a data limit of *n* bytes.

(2) The producer simply sends to the port at the rate it can generate data. There is no explicit synchronization; the producer blocks due to the flow control on the port.

(3) The consumer simply receives at the rate it can consume data from the port. It blocks when the port is empty, or by the flow control mechanism of `STREAM_MPO::receive()`.

Variations of producer/consumer synchronization can be implemented using the hysteresis property of the flow control mechanism.

14. ACKNOWLEDGEMENTS

Raj Vaswani implemented a major part of the DASH kernel, was involved in its design, and contributed to this report. Robert Wahbe, and Giuseppe Facchetti also were involved in the design and implementation. G. Scott Graham provided feedback on the report.

REFERENCES

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, June 9-13, 1986, 81-92.
2. D. P. Anderson and D. Ferrari, "The DASH Project: An Overview", Technical Report No. UCB/Computer Science Dpt. 88/405, Computer Science Division, EECS, UCB, Berkeley, CA, Feb. 1988.
3. M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Systems", *AT&T Bell Labs Technical Journal 63*, 8 (Oct. 1984), 1733-1750.
4. B. Beck and B. Kasten, "VLSI Assist in Building a Multiprocessor UNIX system", *Proceedings of the 1985 Summer USENIX Conference*, Portland, Oregon, June 11-14, 1985, 255-275.
5. D. D. Clark, "The Structuring of Systems Using Upcalls", *Proc. of the 10th ACM Symp. on Operating System Prin.*, Orcas Island, Eastsound, Washington, Dec. 1-4, 1985, 171-180.
6. D. D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems", *IEEE Computer*, June, 1985, 9-28.
7. N. H. Gehani and W. D. Roome, "Concurrent C", *Software—Practice & Experience 16(9)* (Sep. 1986), 821-844.
8. N. Hutchinson and L. Peterson, "Design of the x kernel", *ACM SIGCOMM 88*, June 1988.
9. J. Kepecs, "Lightweight Processes for UNIX Implementation and Applications", *Proceedings of the 1985 Summer USENIX Conference*, Portland, Oregon, June 11-14, 1985, 299-308.
10. *Symmetry Technical Summary*, Sequent Computer Systems, Inc., 1987.
11. B. Stroustrup, "The C++ Programming Language", *Addison-Wesley*, 1986.
12. G. Varghese and T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility", *Proc. of the 11th ACM Symp. on Operating System Prin.*, Austin, Texas, Nov. 8-11, 1987, 25-38.
13. "The DASH Programmer's Manual", Internal document, August 1988.
14. "The DASH Network Communication Architecture", UCB/Computer Science Dpt. Technical Report, in preparation, August 1988.
15. "The DASH Virtual Memory System", UCB/Computer Science Dpt. Technical Report, in preparation, August 1988.