Implementation of Multiprocessing SPUR Lisp

Kinson Ho
Paul Hilfinger

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley California

November 4, 1988

[1]This research was funded in part by DARPA contract number N00039-85-C-0269 as part of the SPUR research project.

Abstract

This paper describes an implementation of the multiprocessing features of SPUR Lisp. The implementation consists of a set of C runtime routines that is being integrated into the Lisp system. It was developed on Sun workstations under the Sprite operating system, and will be ported to the SPUR multiprocessor workstations when the hardware becomes available. In this report, we describe the internal structure of the system, design tradeoffs, and present preliminary performance figures. We hope this paper will be useful to other implementors of these multiprocessing features.

Contents

List of Figures

List of Tables

1 Introduction

This paper describes an implementation of the multiprocessing features of SPUR Lisp [6]. Each multiprocessing extension is implemented as a set of C runtime routines that can be called from the SPUR Lisp system [5] .

The implementation consists of a large, machine-independent part written in C and a small, machine-dependent part written in assembly language. It runs on Sun-2 and Sun-3 workstations under the Sprite operating system [4]. Eventually the system will be ported to multiprocessor SPUR workstations [2].

This paper describes the implementation in great detail. It highlights the problems encountered in the design, the solutions selected, and various design tradeoffs. Preliminary performance numbers and potential improvements are also presented. We hope this paper will be useful to other implementors of these multiprocessing features.

This paper begins with an overview of the different modules of the implementation. It then presents detailed descriptions of the various modules, including the Lightweight Process scheduler, the Synchronization module, the Mailbox module, the Future module, and the Asynchronous module. The operating system support required by the implementation is also discussed briefly. The discussion concludes by explaining the design philosophy behind various high-level design decisions.

The appendices contain the C interface of the implementation, a list of all the files, the size of the implementation, the parameters of the implementation, and some preliminary performance numbers. A list of potential performance improvements is also discussed.

This paper does not address the issue of interfacing the C and Lisp subsystems, since the garbage collection problems in this mixed-language system have not been solved completely. As the SPUR port is still in progress, machine-dependent code is shown only for the Sun (MC68000) implementation.

2 System Overview

The system [1] is implemented as 9 modules:

Lightweight Process Scheduler Implements lightweight processes.

Synchronization Synchronization primitives include Sprite locks, Mesa-style monitors [3], and semaphores [1]. These primitives are used by the Lightweight Process Scheduler, the Mailbox module, and the Asynchronous module.

Mailbox Implements the mailbox operations, including Send, Receive, and Send-after-delay. The mailbox routines are also used by the Future module and the Asynchronous module.

Future Implements futures and delays using mailbox routines.

Asynchronous Interactions Implements asynchronous interprocess communication primitives using mailbox routines and Sprite signals.

Clock Timer routines used by Send-after-delay.

Machine Dependencies Machine-dependent routines used to create lightweight processes and perform context switching. These are written in assembly language.

Main Top-level initialization routines and debugging support. A fast (but simple-minded) version of the C memory allocation routine *Mem_Alloc* is also included.

[1] In this paper the terms *this implementation* and *this system* are used interchangeably.

3 Lightweight Processes (Lwp)

Multiprocessing SPUR Lisp processes are designed to be lightweight to encourage fine-grain parallelism. Each SPUR Lisp process is implemented as a lightweight process (lwp). A SPUR Lisp process is not implemented as an operating system process (Sprite process) because a lwp is cheaper to create than a Sprite process. In addition, the overhead of performing synchronization operations (including long-term waits for a resource or condition) between lwps is lower. This factor is critical because synchronization operations may occur very frequently in a system of multiprocessing SPUR Lisp processes.

For the Sun (MC68000) implementation, the saved context or hardware state of a lwp consists of the runtime stack, the saved registers d0-d7, a0-a7, and a saved Sprite signal mask. The program counter and condition code register are saved on the stack.

3.1 Implementation of Lwp

Lwps are implemented with a lwp scheduler and a pool of Sprite server processes.

Each lwp may be in one of the following states: [2]

Ready-State (:blocked) A new lwp enters the system in this state. It remains in this state until it is scheduled for execution by one of the server processes. A lwp also goes into this state when it wakes up after being blocked by a semaphore. A lwp in this state is enqueued in the runnable queue (ReadyL-wpQ).

Run-State (:executing) A lwp in this state is executing in the context of a server process. The lwp runs until it terminates completely, is blocked by a mailbox operation, is blocked when it touches an undetermined future, or is suspended or killed by another lwp. A server process is not time-sliced between executing lwps. A lwp in this state is enqueued in the run queue (RunLwpQ).

Blocked-State (:blocked) A lwp enters this state to perform a long-term wait on a resource or condition. This happens when a lwp attempts to aquire the mutex semaphore of a locked mailbox, tries to receive mail from an empty mailbox, touches an undetermined future, or waits for the Kill or Suspend operation it invoked to complete. When the lwp wakes up it goes to Ready-State. These long-term waits are implemented as the semaphore operation P. A lwp in this state is enqueued in the lwp queue associated with the semaphore on which it is blocked.

[2]The value returned by the SPUR Lisp function Process-state for a lwp in that state is given in parentheses.

Suspended-State (:suspended) A lwp enters this state when it is suspended by another process (or itself). When it is resumed it goes to Ready-State. A lwp in this state is enqueued in both SuspendLwpQ and the lwp queue associated with one of its special semaphores (SelfSuspend_SemaPtr). (See Section 3.6.)

Dead-State (:terminated) A lwp enters this state when it has terminated. This may happen because the code the lwp is executing returns completely, or because it is killed by another lwp. A lwp in this state is enqueued in both DeadLwpQ and the lwp queue associated with one of its special semaphores (SelfKill_SemaPtr). (See Section 3.6.)

The state transitions of lwps is shown in Figure 1.

The queues maintained by the lwp scheduler (ReadyLwpQ, RunLwpQ, SuspendLwpQ, DeadLwpQ and a lwp queue associated with each semaphore) are implemented as FIFOs because the issue of priority between lwps is still being investigated. Also see Section 7.3 for a discussion of scheduling lwps for futures.

The server processes are created at system initialization time. Currently, the number of server processes does not change dynamically with the number of lwps in Ready-State, or the number of processors available on the workstation. It would be simple to vary the size of the server pool dynamically, however. There is no fixed assignment of lwps to server processes.

The server processes share the same kernel, code and heap segments, and have their own stack segments. All server-specific data is kept in the stack segment, despite the lack of support from the Sun C compiler. The way server-specific data is stored and accessed is described in detail in Section 3.2.

As all lwps are presumably cooperating in performing a computation, server processes are not time-sliced between executing lwps.

A server process only blocks when the lwp it is executing invokes an operating system call that blocks, such as an I/O or a (low-level) locking operation.

3.2 Sprite Segments of the Address Space

All the server processes execute in the same address space, sharing the kernel, code and heap segments. The stack segment of a server process is private. All the data of a lwp (including its stacks) must be kept in the shared heap segment.

There are two types of global data in the system:

Shared Data shared (read/write) by all the server processes. These include system data structures such as Sprite monitor locks, lwp queues of the lwp scheduler, data structures of the allocators for stacks and signal stacks (Sections 3.5 and 8.10), and the FIFO of undelivered messages maintained by the Send-after-delay module.

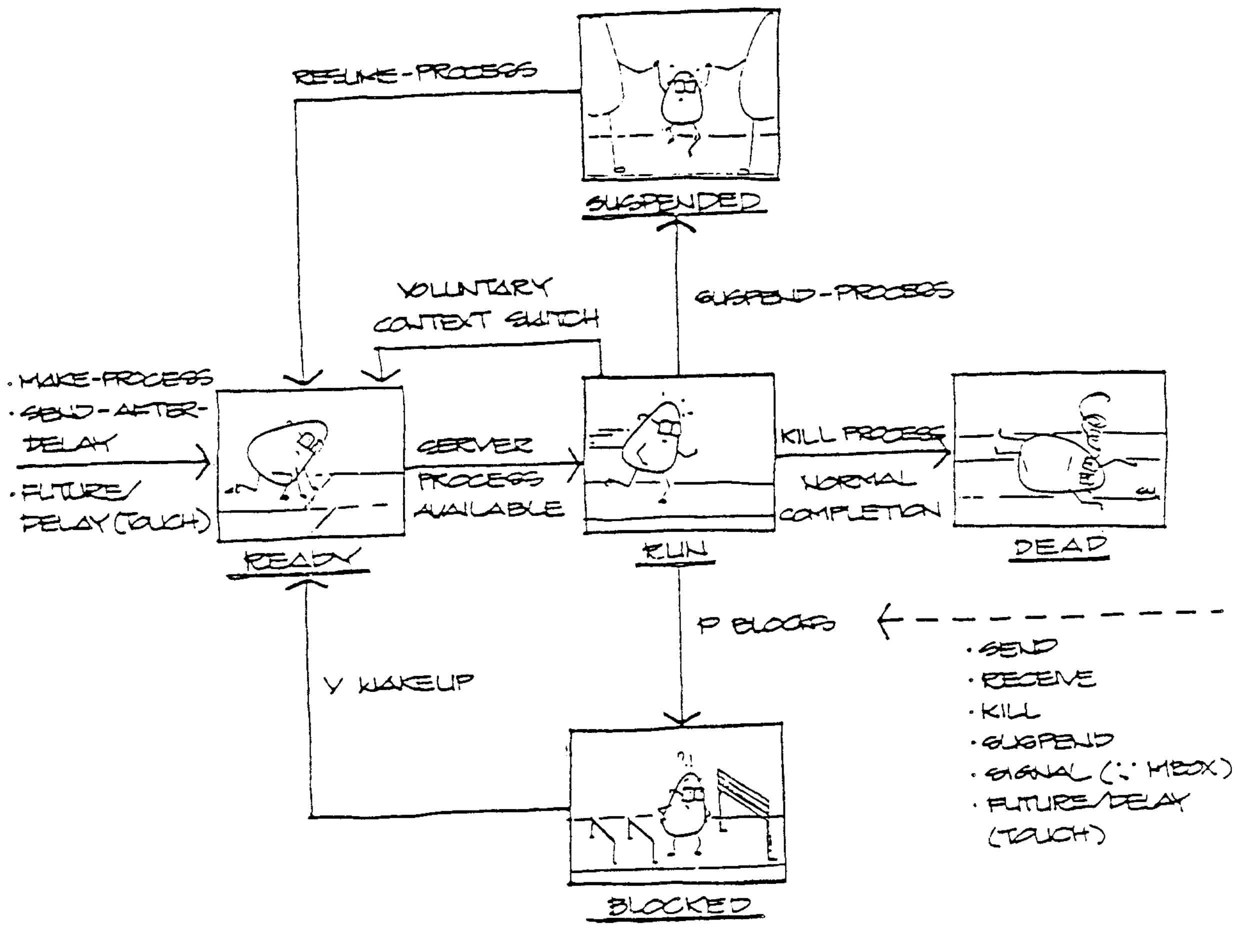


Figure 1: State Transitions Of Lwps

Private Data specific to a given server process. These include the Sprite process-id of a server process and a pointer to the lwp it is currently executing.

Global data are treated differently by the Sun C compiler and the SPUR C compiler. This section describes the differences and ways to write code for allocating and accessing global data (shared and private) that is identical for the Sun port and SPUR port.

For the Sun C compiler and linker, all the global or external data of a process are allocated in shared storage (heap segment). Server-specific data are kept on the private stack of the process.

For the SPUR C compiler and linker, all the global or external data of a process are allocated in private storage (stack segment). Shared data must be allocated dynamically in the shared heap using *Mem_Alloc*.

```
    PerServerType  *PerServerPtr; /* Private */
    SharedObjType  *SharedObjPtr; /* Shared */

void
main()
{
    PerServerType  PerServerInfo;

    ...

    PerServerPtr = &PerServerInfo;

    ...

    SharedObjPtr = (SharedObjType *)
                     Mem_Alloc(sizeof(SharedObjType));

    ...

    /* fork Sprite processes here */

    ...

}
```

Figure 2: Allocating and Accessing Global Data

Figure 2 illustrates how shared and private global data are allocated and accessed. This method may be used for both the Sun port and the SPUR port.

Figures 3 and 4 illustrate how shared and private global data are allocated at the implementation level for the Sun port and the SPUR port.

3.2.1 Shared Global Data

A shared global object is allocated dynamically in the shared heap *before* the server processes are forked. It is accessed through a global variable such as `SharedObjPtr`.

For the Sun C compiler, the shared global variable `SharedObjPtr` of all the server processes points to the shared global object. For the SPUR C compiler, the private global variable `SharedObjPtr` of each server process points to the shared global object.

3.2.2 Private Global Data

All private global data are kept on the private stack of a server process. The global variable `PerServerPtr` points to the same stack *address* of each server process. This address corresponds to the structure `PerServerInfo` in the *main* function of the server process dereferencing `PerServerPtr`. All private data of a server process are stored in its *copy* of `PerServerInfo`.

For the Sun C compiler, the shared global variable `PerServerPtr` of all the server processes points to the private copy of `PerServerInfo` of the server process dereferencing `PerServerPtr`. For the SPUR C compiler, the private global variable `PerServerPtr` points to the private copy of `PerServerInfo` of the server process dereferencing its copy of `PerServerPtr`.

3.3 Lwp Context Switching

The saved context or hardware state of a lightweight process consists of the runtime stack, the saved registers, and the Sprite signal mask. The program counter and condition code register are saved on the stack of the lwp. This is shown in Figure 5.

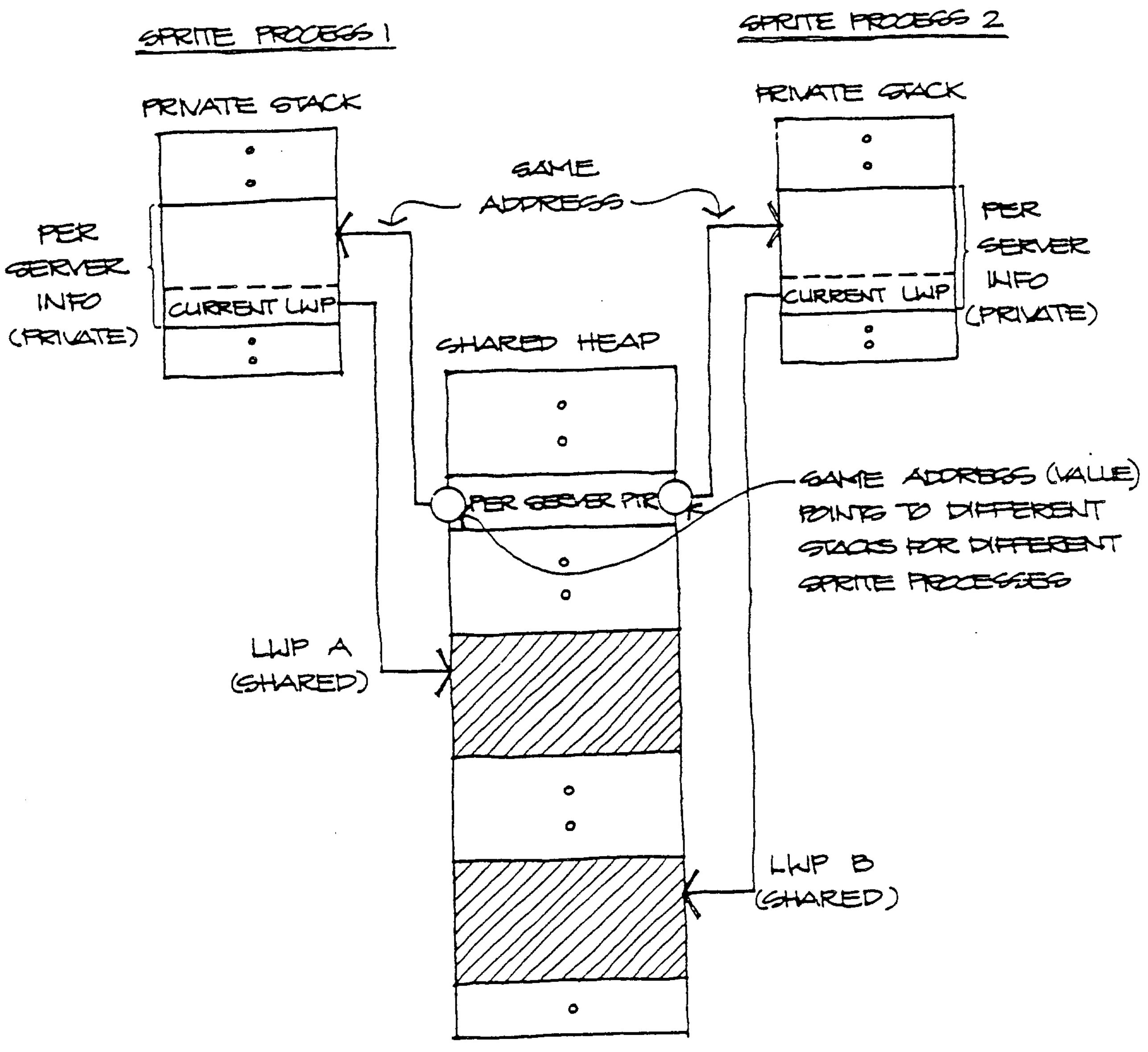
During a context switch,

1. The condition code register is pushed onto the stack of the lwp.

2. A magic number is pushed onto the stack of the lwp.

3. All the registers (d0-d7, a0-a7) of the lwp to be switched *out* are saved in its process control block (Pcb) in the shared heap.

4. All the registers (d0-d7, a0-a7) of the lwp to be switched *in* are restored from its process control block. This implictly changes the stack pointer (a7).



*[CANNOT HAVE PTR FROM LWP TO A PARTICULAR SPRITE PROCESS] (PER SERVER INFO)

Figure 3: Global Data for Sun Port

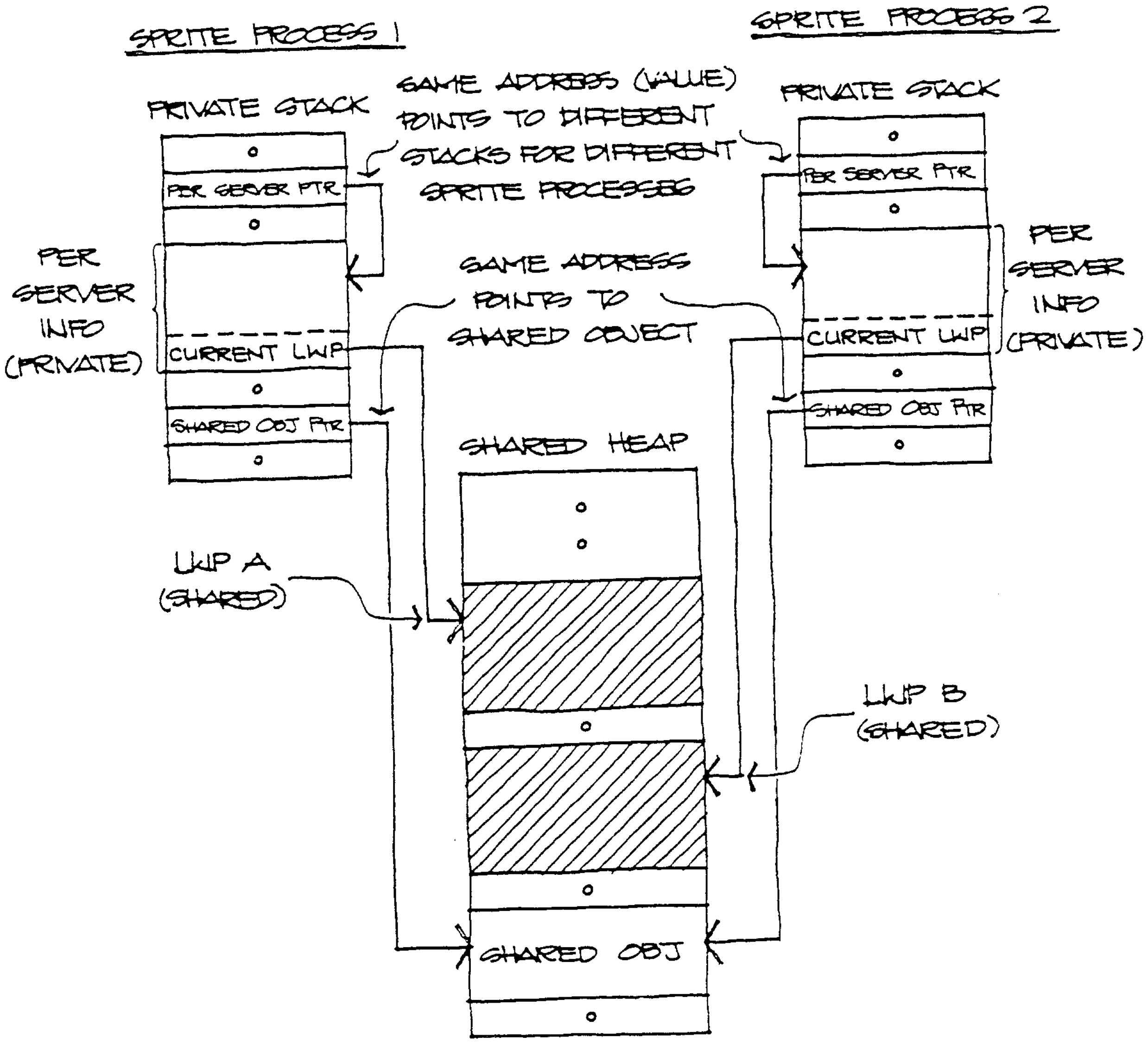


Figure 4: Global Data for SPUR Port

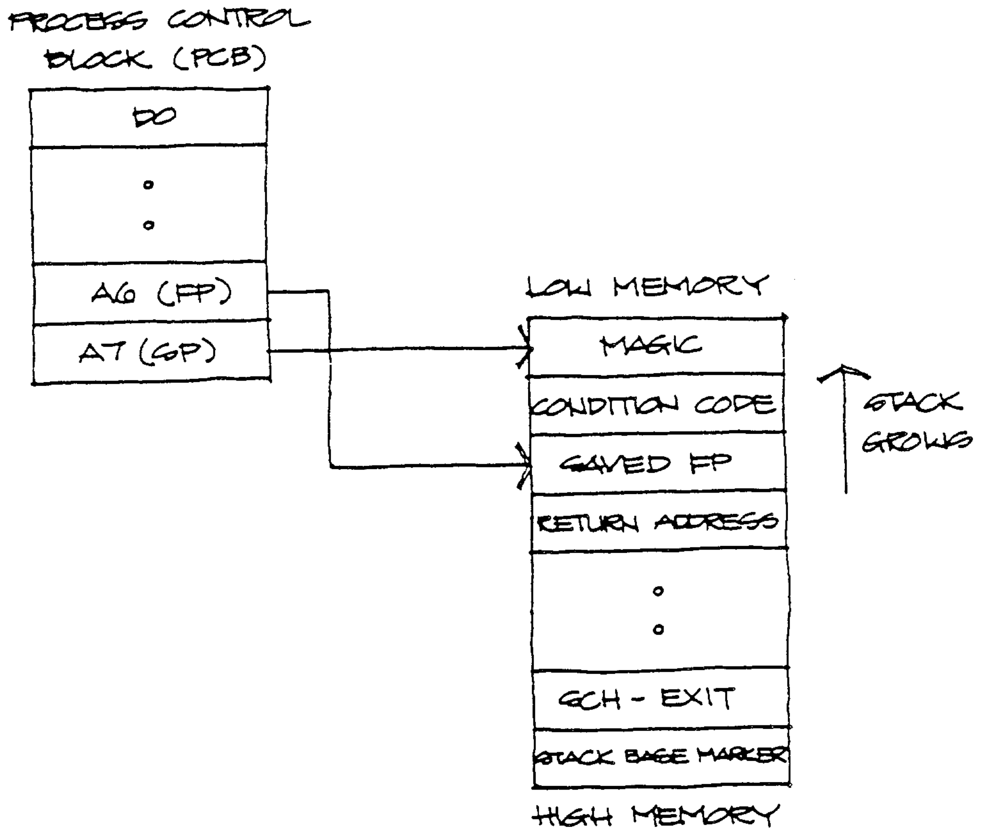


Figure 5: Stack of a Saved Lwp

5. The magic number is popped from the lwp stack and checked to ensure the integrity of contents of the stack.

6. The condition code register is popped from the lwp stack.

7. The new lwp returns to user code.

Synchronization during a context switch is described in Section 4.3.

When a lwp is switched out (because it is blocked or has terminated), the thread of control associated with the server process is resumed. This thread executes on the stack provided by the operating system when the server process was forked. This thread contains an infinite loop that executes any lwp in ReadyLwpQ, and sleeps if there is none. The lwp scheduler always switches from a lwp to the server thread, and never switches from one lwp to another directly. This organization simplifies the lwp scheduler considerably, at the cost of extra context switches.

3.4 Lwp Creation

When a lwp is created, certain things are placed on its stack so that it looks like a descheduled lwp, as shown in Figure 6. In this way, executing a lwp for the first time is very similar to resuming a lwp that has been switched out. When a lwp is *restored* for the first time, it calls Sch_Start_Lwp (for initialization), then the user code, and finally the exit routine of the lwp scheduler (Sch_Exit). See Section 4.4 for an explanation of Sch_Start_Lwp.

3.5 Allocation of Lwp Stacks

Each lwp executes on a runtime stack that is a section of a dynamically allocated C object. This object is allocated from the shared heap segment, and is readable and writable by all server processes. The lwp stack is not allocated from a Sprite stack segment, which is non-sharable and is owned by the server process that created the lwp.

A lwp stack contains an integral number of operating system pages, and is aligned on page boundaries. To detect stack overflows and underflows, the Sprite system call *Vm_DestroyVA* is used to put an invalid page between every two lwp stacks. An address fault (Sprite signal) will be generated if a lwp stack overflows or underflows. This is illustrated in Figure 7.

Stacks of dead lwps are reused by the lwp stack allocator. This is described in Section 3.8.

3.6 Special Semaphores

Each lwp has the following special semaphores, which are used by the lwp scheduler to context switch, suspend or kill a lwp. P and V operations on these semaphores

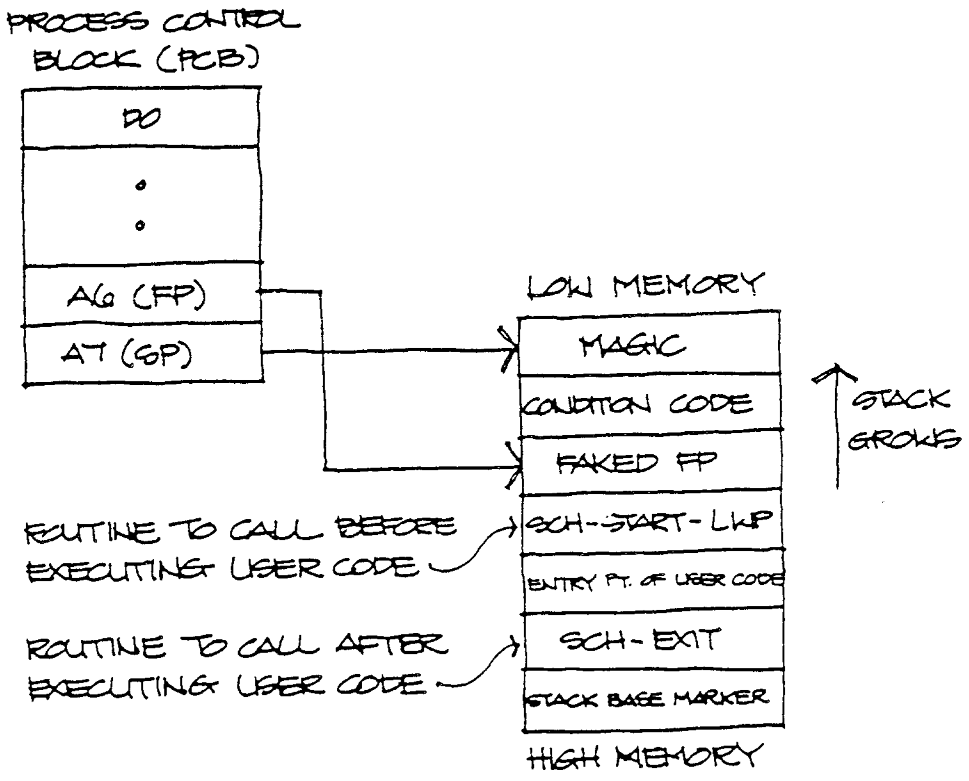


Figure 6: Stack of a New Lwp

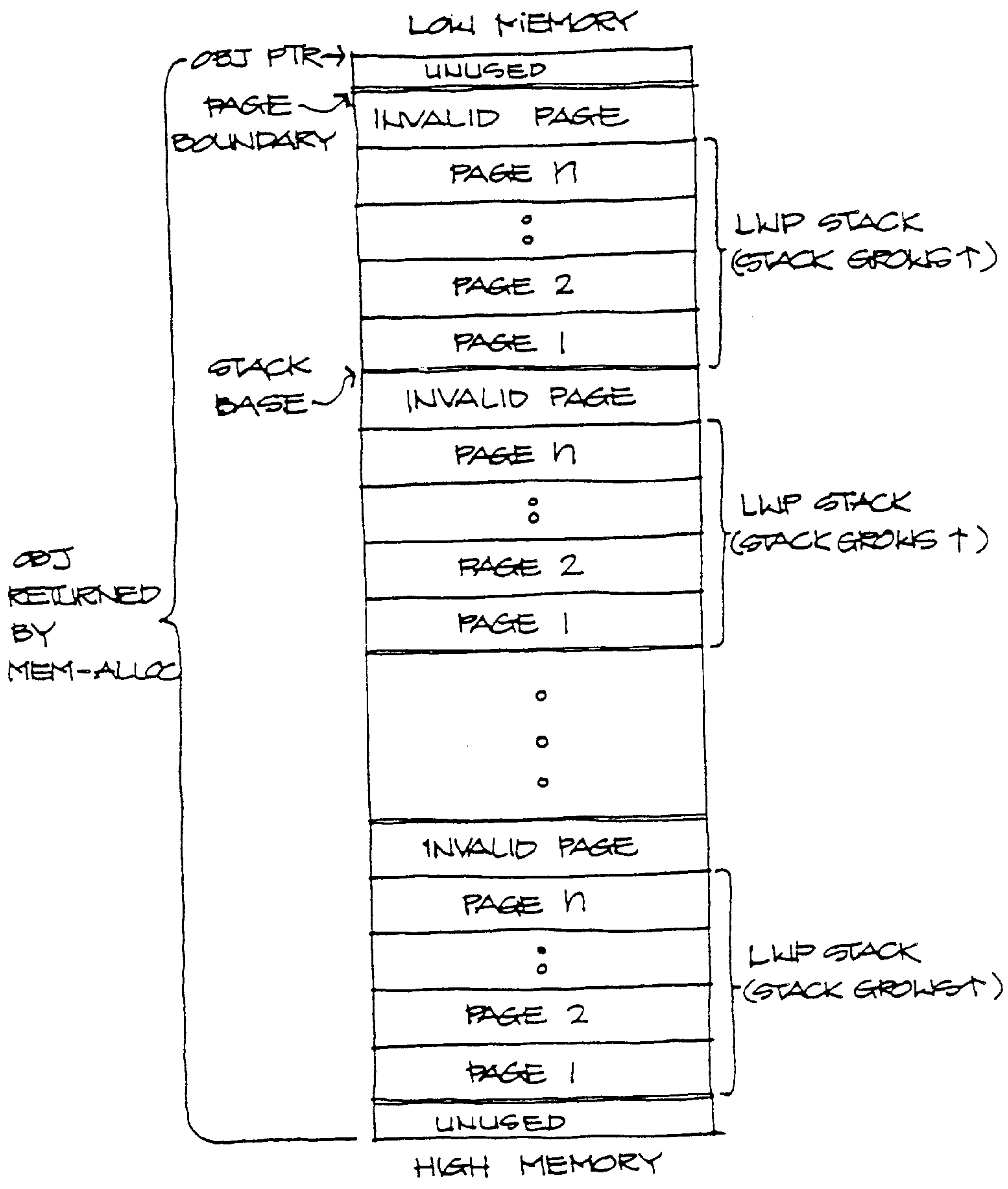


Figure 7: Allocation of Lwp Stacks

are handled differently by the semaphore module.

SelfSwitch_SemaPtr Initialized to 0. Used by a lwp to do a voluntary context switch. Caller lwp is blocked on ReadyLwpQ (Ready-State). No need to call V to make lwp ready again.

SelfSuspend_SemaPtr Initialized to 0. Used by a lwp to suspend itself. Caller lwp is blocked (suspended) on SuspendLwpQ (Suspended-State). Calling V on this semaphore will resume the suspended lwp.

SelfKill_SemaPtr Initialized to 0. Used by a lwp to kill itself. Caller lwp is blocked (killed) on DeadLwpQ (Dead-State) forever. No V call on this semaphore is allowed.

Using these semaphores for context switching, killing or suspending lwps merges all the code for blocking a lwp, and reduces the complexity of the lwp scheduler significantly.

3.7 Lwp Object

This section summarizes all the fields of a lwp object. See the sections shown in parentheses for details.

PrcsName String name of lwp.

PcbPtr Saved hardware state of lwp. (Section 3.)

ServerPid Sprite process-id of the server process. (Section 3.1.)

PrcsState Process state of lwp. (Section 3.1.)

SigMask The set of disabled Sprite signals for the server process. A saved lwp has the saved signal mask of the server process at the time the lwp was switched out. When the lwp is scheduled to run again, the new server process will begin executing the lwp with this signal mask. (Sections 4.5 and 8.3.)

LinkElt Link field used by PrcsFifo implementation.

Blocked_SemaPtr Semaphore the lwp is blocked on. (Section 4.6.)

SelfSwitch_SemaPtr Semaphore the lwp uses to context switch voluntarily. (Section 3.6.)

SelfSuspend_SemaPtr Semaphore the lwp uses to suspend itself. (Section 3.6.)

SelfKill_SemaPtr Semaphore the lwp uses to kill itself. (Section 3.6.)

NoInterrupt TRUE iff lwp is in a critical section. (Section 8.15.)

SchmonFlag TRUE iff lwp is holding the Schmon lock (outside a critical section). (Section 4.3.)

JmpEnv Each lwp has a C structure that contains all the information saved during a call to the C library routine *_setjmp*. This information is used by *_longjmp* to re-execute a blocked Receive that has been canceled by an asynchronous operation (Kill, Suspend or Signal).

To ensure that this implementation is recursive (a canceled receive inside a canceled receive inside ...), this structure is copied onto the lwp stack before a Receive, and restored when the Receive returns. (Section 8.17.)

MboxSeqPtr Sequence of mailboxes the receiver is receiving mail from. (Section 5.5.)

Mail_drop Message delivered by sender lwp. (Section 5.5.)

From_MboxPtr Mailbox from which the message in Mail_drop came. (Section 5.5.)

Mail_SemaPtr Semaphore receiver lwp blocks on to wait for mail. (Section 5.4.)

Sender_SemaPtr Semaphore to synchronize a receiver lwp and potential sender lwps. (Section 5.4.)

KillFlag TRUE iff lwp is being killed. (Section 8.1.)

SuspendFlag TRUE iff lwp is being suspended. (Section 8.1.)

SignalFlag TRUE iff lwp has pending signals. (Section 8.1.)

Async_MboxPtr Mailbox for sending special messages indicating Kill, Suspend or Signal to a lwp. (Section 8.1.)

Killed_SemaPtr A lwp calling Kill is blocked on this semaphore of its target lwp until the target lwp is killed. (Section 8.7.)

Suspend_SemaPtr A lwp calling Suspend is blocked on this semaphore of its target lwp until the target lwp is suspended. (Section 8.7.)

SigStackPtr Signal stack contains the signal handlers and enabled signals of each dynamic environment of the lwp. (Section 8.10.)

SigObj_MboxPtr Mailbox for signal objects sent to a lwp. (Section 8.1.)

3.8 Deallocating Storage for Dead Lwp

All the dead lwps are enqueued in DeadLwpQ. When a server process becomes idle (i.e. ReadyLwpQ becomes empty), it checks DeadLwpQ for lwps with undeallocated storage. It frees most of the objects associated with the lwp, including the saved hardware state, the lwp stack, the signal stack, and most of the semaphores. The unreclaimed fields are maintained by the lwp scheduler to prevent races.

When the objects associated with a lwp are deallocated, they are returned to the memory allocator using *Mem_Free*. More measurements are necessary to determine if the system should maintain its own storage explicitly to reduce the overhead of allocating and deallocating objects.

4 Synchronization

Multiprocessing SPUR Lisp is designed for implementation on a shared memory multiprocessor. At the operating system level, the implementation consists of a number of user-level Sprite processes executing in the same address space, sharing (read/write) the same heap segment. These processes include the server processes used to execute lwps (Section 3), as well as the processes used to implement Send-after-delay (see Section 6.) These processes are all time-sliced by the operating system.

At the next level, the system consists of lwps. These lwps access shared objects such as mailboxes, futures and processes, as well as system data structures including the lwp queues of the lwp scheduler, the stack array of the lwp stack allocator, the signal stack allocator, and the FIFO of undelivered messages maintained by the Send-after-delay module.

The Sprite processes are synchronized using Sprite monitors and condition variables. The system data structures are protected by Sprite monitors. Lwps are synchronized using semaphores.

4.1 Uses of Synchronization Primitives

Test-And-Set This operation is used by the Sprite lock routines to avoid the overhead of a system call when a lock is acquired or released, if possible. It is implemented using the atomic test-and-set hardware instruction.

Sprite locks These are used to implement semaphores (Section 4.6) and Sprite monitors. They are provided by the Sprite operating system.

Sprite monitors These are used by the lwp scheduler, the lwp stack allocator, the signal stack allocator, the Send-after-delay module, and the Debug module. They are provided by the Sprite operating system.

Condition variables These are used for the long-term blocking of Sprite processes. This includes the blocking of idle server processes and the daemon process for the Send-after-delay module. They are provided by the Sprite operating system.

Semaphores These are used by lwps for synchronization and mutual exclusion in the Mailbox module, the Future module, and parts of the Asynchronous module. They are implemented by lwp scheduler using Sprite locks and monitors.

4.2 Sprite Locks and Monitors

This section provides a simple description of the synchronization primitives provided by the Sprite operating system.

Sprite locks A Sprite lock is conceptually a binary semaphore [1] for Sprite processes. In the best case a lock is acquired or released by setting or clearing flags. In the worst case a blocking system call has to be invoked to acquire the lock, or to awaken all the blocked processes waiting for the lock.

Sprite monitors A Sprite monitor is a Mesa-style monitor with broadcast semantics [3].

4.3 Monitor for the Lwp Scheduler (Schmon)

The global data structures of the lwp scheduler (ReadyLwpQ, RunLwpQ, SuspendedLwpQ, DeadLwpQ) are protected by the Sprite monitor Schmon. As a side effect, locking Schmon also prevents any lwp from changing its process state. To avoid contention at Schmon, Schmon is only locked for very short durations.

Schmon is *not* locked and unlocked using the Mesa-style entry procedures. Instead, it is locked and unlocked explicitly. This locking style allows the lwp scheduler to be coded conveniently and efficiently, but it also introduces the potential of errors and deadlocks.

4.4 Schmon and Context Switching

The following scenario demonstrates a race condition that may occur if Schmon is unlocked *before* calling the assembly language routine that performs the actual context switch, `Mach_ContextSwitch`:

1. Lwp1 running under server process Sa is about to be switched out.
2. Lwp1 locks Schmon, and enqueues itself into a global queue of the lwp scheduler (say ReadyLwpQ).
3. Lwp1 unlocks Schmon.
4. Lwp1 calls `Mach_ContextSwitch`.
5. Before Mach_ContextSwitch completes, another (idle) server process Sb locks Schmon, dequeues Lwp1 from ReadyLwpQ, and attempts to call Mach_ContextSwitch to switch in Lwp1.
6. When this happens, two threads of control (servers Sa and Sb) are attempting to execute Lwp1 simultaneously.

To avoid race conditions, the lwp scheduler is coded so that Schmon is unlocked *after* a context switch, i.e. the lwp (or server) that *returns* from a call to Mach_ContextSwitch is holding Schmon, and has to unlock it before returning to user code. For a newly created lwp, the initial stack frame is set up so that Schmon is unlocked (by procedure Sch_Start_Lwp) before any user code is executed (Figure 6).

4.5 Schmon and Sprite Signals

Sprite signals are used to implement the asynchronous interprocess actions including Kill, Suspend and Signal. As Sprite system calls to disable and enable Sprite signals are relatively inexpensive, all Sprite signals are disabled around any section of code that locks Schmon. This leads to a much cleaner implementation, at a small cost to efficiency. See Section 8.5 for details.

4.6 Semaphores

Semaphores are used by lwps for long-term waits for a resource or condition. When a lwp blocks on a semaphore, the server process becomes idle, and it checks ReadyLwpQ for a ready lwp to execute.

The following operations are provided by the semaphore module:

Sema_P Counting semaphore operation P.

Sema_V Counting semaphore operation V.

Sema_P_In_Schmon, Sema_V_In_Schmon In certain parts of the Asynchronous module, Schmon is acquired explicitly. If the lwp holding Schmon performs a semaphore operation that requires locking Schmon, the system will deadlock. This problem is solved by providing special variants of the semaphore operations P and V that do not acquire Schmon again.

Sema_P_Non_Blocking A non-blocking form of the semaphore operation P is provided. This form returns FALSE in the case when the operation P would have blocked. It is used by the Mailbox module to avoid a potential deadlock. (See Section 5.)

4.7 Semaphore Object

MutexPtr The Sprite lock that serializes accesses to the semaphore (Section 4.1).

Count The integer value of the counting semaphore.

PrcsQPtr The FIFO of lwps blocked on the semaphore.

4.8 Implementation of Semaphores

The Sprite lock of a semaphore is acquired for any semaphore operation (P or V). In the best case only a test-and-set operation is required. If there is contention for this lock a blocking Sprite system call is invoked to acquire it.

If no lwp is blocked or unblocked by a semaphore operation (P or V), only the integer value of the semaphore is incremented (for V) or decremented (for P); Schmon is not locked, and no Sprite system call is required. If a lwp is blocked or unblocked by the operation, Schmon, which protects the global queues of the lwp scheduler, has to be acquired. The semaphore operations P and V are implemented as follows:

Semaphore operation P

```
SelfLwpPtr is a pointer to the current lwp

P(SemaPtr)

if (Test-and-Set(SemaPtr->MutexPtr))
{
    Sprite wait system call on SemaPtr->MutexPtr
}

SemaPtr->Count--
if (SemaPtr->Count < 0)
{
    Lock Schmon Sprite monitor
    Move SelfLwpPtr from RunLwpQ to SemaPtr->PrcsQPtr
    Unlock(SemaPtr->MutexPtr)

    Context switch out SelfLwpPtr


    ...


    /* wakeup: V(SemaPtr) called by another lwp */
    Unlock Schmon Sprite monitor
    return
}
else
{
    Unlock(SemaPtr->MutexPtr)
    return
}
```

Semaphore operation V

```
SelfLwpPtr is a pointer to the current lwp

V(SemaPtr)

BlockedLwpPtr is a local variable

if (Test-and-Set(SemaPtr->MutexPtr))
{
    Sprite wait system call on SemaPtr->MutexPtr
}

SemaPtr->Count++
if (PrcsFifo_Delete(SemaPtr->PrcsQPtr, &BlockedLwpPtr)
{
    Unlock(SemaPtr->MutexPtr)
    Lock Schmon Sprite monitor
    Enqueue BlockedLwpPtr into ReadyLwpQ
    Sprite broadcast system call to wake up idle server processes
    Unlock Schmon Sprite monitor
    return
}
else
{
    Unlock(SemaPtr->MutexPtr)
    return
}
```

5 Mailboxes

5.1 Message Object

Conceptually, a mail item, or message, is any Lisp object reference. It is implemented as an object with the following fields:

Data The actual message, a Lisp pointer.

MesgClass The kind of message (Normal, Kill, Suspend, Signal, Signal object) (See Section 8).

Sender Sender lwp (for debugging).

Receiver Receiver lwp (for debugging).

Mbox Mailbox the message is sent to (for debugging).

SerialNumber Serial number of message (for debugging).

5.2 Mailbox Object

A mailbox is an unbounded FIFO of message objects. It is implemented as an object with the following fields:

Mbox_SemaPtr Semaphore to serialize concurrent accesses to the mailbox.

MesgQPtr The FIFO of message objects that have not been received from the mailbox.

PrcsQPtr The queue of receiver lwps that are waiting for mail from this (empty) mailbox. Note that lwps may be dequeued from this out of order when a receiver receives mail from another mailbox, or when a receive operation is canceled because of an asynchronous operation.

MboxName The string name of the mailbox.

5.3 Algorithm for Send and Receive

The algorithm for Send and Receive (multiple receive) are described at a high level without reference to synchronization here. See Section 8.17 for a detailed description of the implementation, including the interactions with asynchronous operations.

Send

```
if there is no waiting receiver lwp in the mailbox
{
        enqueue message
        return
}
else if there is a waiting receiver lwp
{
        deliver message to receiver
        dequeue receiver from all other mailboxes it is enqueued in
        wakeup receiver
        return
}
```

Receive

```
for each mailbox in the sequence
{
        if there is no message in the mailbox
        {
                enqueue itself in the mailbox
        }
        else
        {
                dequeue message from the mailbox
                dequeue itself from all mailboxes it is enqueued in
                return
        }
}
block self
retrieve mail delivered by sender
return
```

5.4 Data Structures

In this section the data structures used by the mailbox routines are described.

Mbox_SemaPtr Mutex semaphore initialized to 1. One per mailbox. Used to serialize concurrent accesses to a mailbox.

Mail_SemaPtr Semaphore initialized to 0. One per lwp. Used by a receiver lwp to wait for receipt of a message. Receiver has been dequeued from all other mailboxes when it proceeds from this semaphore.

Sender_SemaPtr Semaphore initialized to 1. One per lwp. Used to synchronize sender and receiver lwps. A receiver checking a sequence of mailboxes uses this semaphore to prevent senders from delivering mail to it, while the first (successful) sender to a blocked receiver uses this semaphore to prevent other senders from sending mail to that blocked receiver.

5.5 Mail Delivery

The following fields in a receiver lwp are used by a sender lwp to deliver mail to a *blocked* receiver:

Mail_drop Message delivered by a sender for pending receive.

MboxSeqPtr The sequence of mailboxes the receiver is receiving mail from. This is used by the cleanup routine of the sender to dequeue the receiver from all other mailboxes after delivering mail to the receiver.

From_mboxPtr The mailbox from which the message in Mail_drop came.

If a sender discovers a blocked receiver in the mailbox it is sending to, it puts the message into the Mail_drop slot of the receiver, sets Received_mail and From_mboxPtr, and dequeues the receiver from all other mailboxes of MboxSeqPtr before waking up the receiver. When a receiver wakes up, it retrieves mail from its fields and returns.

5.6 Race Conditions

Here are some of the race conditions that have been detected and corrected:

- Mail arrives at a mailbox a receiver has enqueued itself in *before* the receiver has finished polling all the mailboxes in the sequence.
  Solution: Sender is blocked (on Sender_SemaPtr of receiver).

- Receiver is blocked after checking a sequence of mailboxes. Two sender lwps concurrently send to two mailboxes the receiver is enqueued in.
  Solution: The first sender to lock Sender_SemaPtr of the receiver wins.

- Receiver is blocked after checking a sequence of mailboxes. Sender S1 finds receiver in mailbox m1 and delivers mail to it. Before S1 can dequeue the receiver from mailbox m2, another sender S2 discovers the receiver in m2.
  Solution: The first (successful) sender, S1, clears the MboxSeqPtr field of

the receiver before unlocking the Sender_SemaPtr of the receiver. When S2 discovers that MboxSeqPtr of the receiver is empty, it will unlock m2, de-schedule itself (onto ReadyLwpQ), and retry the send operation to m2 when it is scheduled to run again. It will continue doing so until S1 has dequeued the receiver from m2.

- Receiver R is blocked after checking a sequence of mailboxes. A sender S1 unblocks Sender_SemaPtr of R after it has delivered mail to R, but before it dequeues R from all the mailboxes. If R is awakened before this cleanup is completed, R may begin to execute yet another Receive operation. As the MboxSeqPtr field of R will now be set again (by the second receive operation), mail may be delivered to R at one of the mailboxes from the *previous* receive operation.
  Solution: Wake up R after the cleanup operation has completed.

5.7 Deadlock Prevention

- To prevent deadlocks, each lwp (sender or receiver) will only lock one mailbox at any time.

- A sender locks a mailbox before locking Sender_SemaPtr of a receiver it finds in a mailbox, while a receiver locks its Sender_SemaPtr before locking a mailbox. This locking protocol is prone to deadlocks.
  Solution: The sender uses the non-blocking operation Sema_P_Non_Blocking to acquire Sender_SemaPtr of the receiver. If the locking fails, the sender unlocks the mailbox and retries the locking sequence.

6 Send-after-delay

Send-after-delay sends a message to a specified mailbox after a specified delay. The caller lwp is *not* blocked until the message is sent.

6.1 DSendInfo Object

The information associated with a call to Send-after-delay is packaged into an object with the following fields:

Mbox Mailbox the message is being sent to.

Mesg Message object to be sent.

Interval Delay interval before sending the message.

Sender Sender lwp (for debugging).

6.2 Implementation of Send-after-delay

The lwp calling Send-after-delay cooperates with two Sprite processes (not server processes) and a new lwp to implement Send-after-delay. When Send-after-delay is called, the caller lwp enqueues the DSendInfo object into a global FIFO (`DSendQ`). It then wakes up DSend_Daemon, a Sprite process. The caller lwp returns immediately. When DSend_Daemon wakes up, it checks `DSendQ` for DSendInfo objects. For each such object, it forks a Sprite process (SleepPrcs), and passes the object to SleepPrcs. If `DSendQ` is empty, DSend_Daemon sleeps. SleepPrcs sleeps for the duration specified in the DSendInfo object passed to it, creates a lwp (SendLwp) that will execute the send operation when scheduled, and exits. When SendLwp is scheduled, it sends the message to the specified mailbox and exits.

Caller of Send-after-delay (a lwp)

```
DSend-Delay-Send(MboxPtr, MesgPtr, Interval)

DSendInfoPtr is a local variable

    DSendInfoPtr = Create DSendInfo object
    Lock DSend_MonLock monitor
    Enqueue DSendInfoPtr into DSendQ
    Sprite broadcast to wakeup DSend_Demon Sprite process
    return
```

DSend_Daemon Sprite Process

```
Lock DSend_MonLock monitor

while (TRUE)
{
    while (DSendQ is not empty)
    {
        Dequeue DSendInfo object from DSendQ
        Fork Sprite process SleepPrcs,
            passing it DSendInfo object
    }

    Unlock DSend_MonLock and wait
        for Sprite broadcast (atomically)
    /* When this returns, DSend_MonLock is held again */

}
```

SleepPrcs Sprite Process

```
SleepPrcs(DSendInfoPtr)

    Sleep for interval specified in *DSendInfoPtr
    Create a new lwp SendLwp that will send the message
        in *DSendInfoPtr when scheduled
    SleepPrcs exits
```

An Idle Sprite Server Process

```
    Dequeue ready lwp SendLwp from ReadyLwpQ and start execution
```

SendLwp

```
    Sends message to specified mailbox
    SendLwp exits
```

The initial stack of SendLwp is shown in Figure 8.

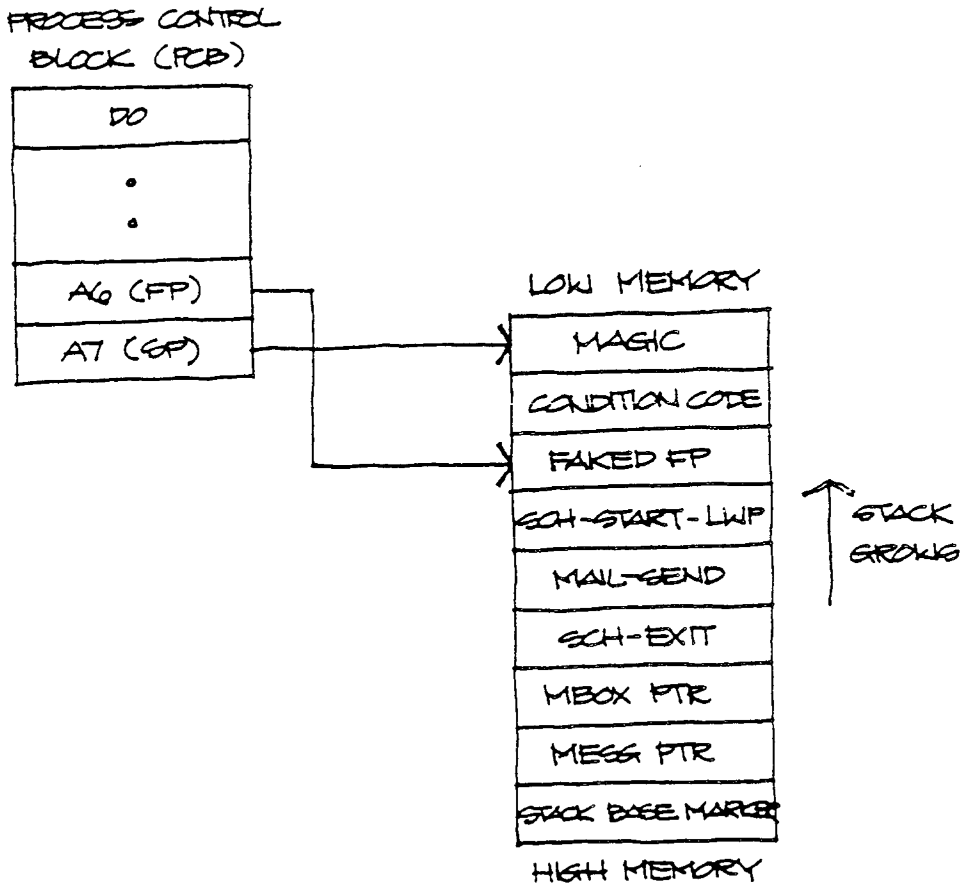


Figure 8: Stack of lwp created by Send-after-delay

6.3 Design Decision

The implementation described is simple but very expensive, as a Sprite process is forked for each call to Send-after-delay. If Send-after-delay is found to be used heavily, a more efficient (but more complex) scheme should be used instead.

7 Future and Delay

Futures and delays are implemented using the mailbox facilities.

7.1 Future Object

A future or delay object has the following fields:

Future_SemaPtr Mutex semaphore initialized to 1. Used to serialize concurrent accesses to a future object.

FormToEvalPtr Entry point of code the future is created to evaluate.

ParentPrcsPtr Lwp that created the future (for debugging).

FuturePrcsPtr Lwp created to compute FormToEvalPtr,
NULL for a delay that has not been touched.

Determinedp TRUE iff the future has been determined.

ValuePtr Value of the future, if determined.

NumBlockedPrcs Number of lwps blocked after touching the undetermined future.

Future_MboxPtr Mailbox the blocked (touching) lwps are waiting for mail from.

7.2 Implementation of Future and Delay

When Future is invoked, a future object is created, and a lwp is created to compute the value of the future. When Delay is invoked, a delay object is created, but no lwp is formed. When a determined future (or delay) is touched, its value is retrieved from the future (or delay) object. When an undetermined future is touched, the touching lwp increments a counter in the future object, and blocks by receiving from an empty mailbox associated with the future object. When a delay is first touched, a lwp is created to compute its value, and the touching lwp is blocked in the same way as a lwp that touches an undetermined future. When the lwp associated with a future (or delay) has computed the value of the future (or delay), it stores the value into the future (or delay) object. It then sends one message containing the value of the future (or delay) to the mailbox associated with the future (or delay) for each blocked touching lwp. The future lwp then exits.

Creating a Future

```
Fut-Create-Future(FnPtr)

FuturePtr is a local variable

    Create a future object *FuturePtr
    FuturePtr->FormToEvalPtr = FnPtr (entry point of code)
    FuturePtr->FuturePrcsPtr = new lwp created to compute
        Fut-Compute-Future(FnPtr, FuturePtr)
    return (FuturePtr)
```

Creating a Delay

```
Fut-Create-Delay(FnPtr)

FuturePtr is a local variable

    Create a future object *FuturePtr
    FuturePtr->FormToEvalPtr = FnPtr (entry point of code)
    return (FuturePtr)
```

Touching a Future or Delay

```
Fut-Touch-Future(FuturePtr)

MboxSeqPtr, MesgPtr and MboxPtr are local variables

    P(FuturePtr->Future_SemaPtr)
    if (Determinedp)
    {
        if (FuturePtr->ValuePtr is an undetermined future)
        {
            Fut-Touch-Future(FuturePtr->ValuePtr)
        }
        V(FuturePtr->Future_SemaPtr)
        return value in FuturePtr->ValuePtr
    }
    else
    {
        FuturePtr->NumBlockedPrcs++
        /* Delay */
        if (FuturePtr->FuturePrcsPtr == NULLP(PrcsType))
        {
            Create a new lwp to compute FormToEvalPtr,
                and store it in FuturePtr->FuturePrcsPtr
        }

        V(FuturePtr->Future_SemaPtr)

        MboxSeqPtr = single mailbox FuturePtr->Future_MboxPtr
        Mail-Receive-Atomic(MboxSeqPtr, &MesgPtr, &MboxPtr)
        return value in MesgPtr
    }
```

Future or Delay becoming Determined

```
Fut-Compute-Future(FnPtr, FuturePtr)

ValuePtr, PrcsCount and MesgPtr are local variables

    ValuePtr = (*FnPtr)()
    P(FuturePtr->Future_SemaPtr)
    FuturePtr->ValuePtr = ValuePtr
    FuturePtr->Determinedp = TRUE

    for (PrcsCount = 0;
         PrcsCount < FuturePtr->NumBlockedPrcs;
         PrcsCount++)
    {
        MesgPtr = message object with ValuePtr as data
        Mail-Send-Atomic(FuturePtr->Future_MboxPtr, MesgPtr)
    }

    V(FuturePtr->Future_SemaPtr)
```

The initial stack of a lwp created to compute a future is shown in Figure 9.

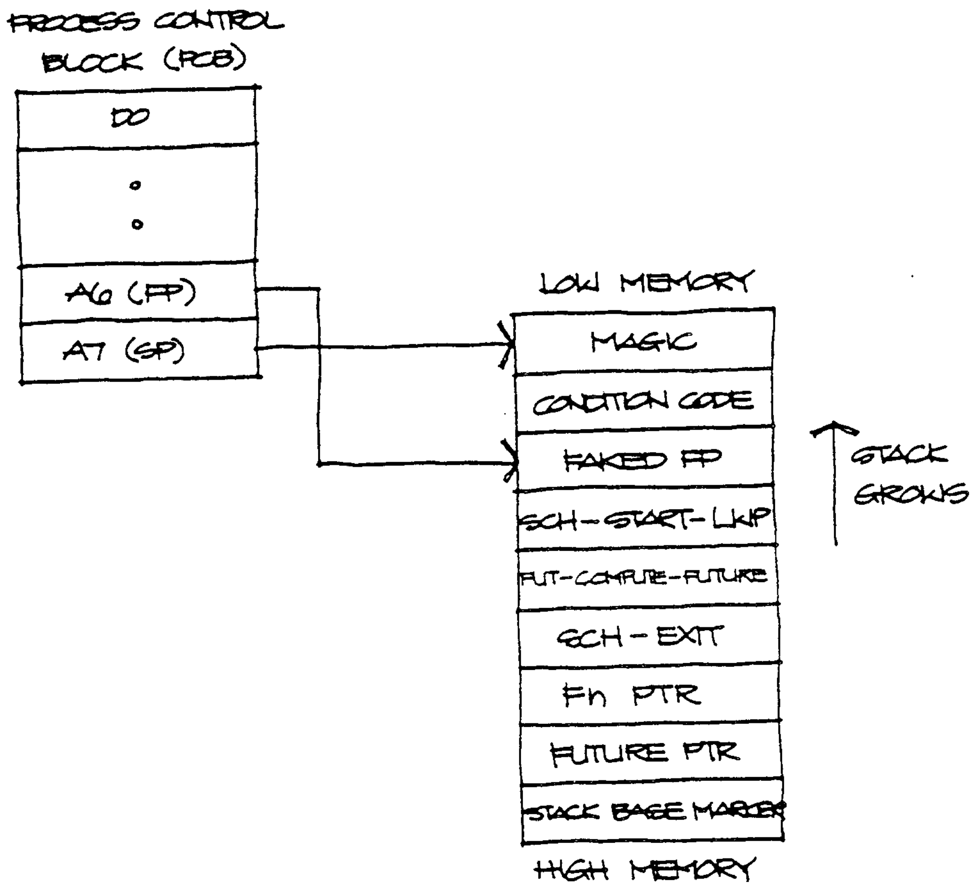


Figure 9: Stack of lwp created to compute a future

7.3 Future and Lwp Scheduler

Currently the lwp scheduler does not distinguish between a lwp created by the user and a lwp created to compute the value of a future. A lwp computing the value of a future is not favored by the lwp scheduler over other lwps, even if there are lwps blocked waiting for the value of the future. Furthermore, a lwp invoking Future is not descheduled in order to execute the lwp created to compute the future. More measurements are necessary to determine if this policy produces reasonable performance.

7.4 Design Decisions

The Future module is implemented on top of the Mailbox module. This implementation is simple and highly portable, at the price of sacrificed performance. If the speed of futures is of critical importance, a faster implementation using lower level primitives (such as semaphores) should be chosen.

8 Asynchronous Interactions

The asynchronous operations between lwps include Kill-Process, Suspend-Process, Signal-Process and Resume-Process.[3] They are implemented using mailbox routines and Sprite signals. The caller of Kill, Suspend or Signal does *not* change the state of the target lwp directly. Instead, it sets flags in the target lwp, sends messages to special mailboxes of the target lwp, and perhaps sends a Sprite signal to the target lwp. A lwp checks for the arrival of asynchronous events when it is safe to do so, and kills, suspends or signals itself as appropriate.

8.1 Data Structures

Each lwp contains the following data structures used to handle asynchronous interactions:

KillFlag TRUE iff lwp is being killed. Set by the lwp calling Kill.

SuspendFlag TRUE iff lwp is being suspended. Set by the lwp calling Suspend.

SignalFlag TRUE iff lwp has pending signals. Set by the lwp calling Signal.

Async_MboxPtr Each lwp adds its Async_MboxPtr to the end of the mailbox sequence of every Receive operation it executes. The lwp calling Kill, Suspend or Signal sends a message to this mailbox to wakeup a lwp blocked by a Receive.

SigObj_MboxPtr Mailbox for signal objects sent to a lwp.

8.2 Checking for Asynchronous Events

To preserve the atomicity of Send and Receive, a lwp checks its KillFlag, SuspendFlag and SignalFlag for asynchronous events when it is safe to do so. This is done at the following points:

- Before executing user code the first time a lwp is scheduled (procedure Sch_Start_Lwp).
- Before returning to user code when a lwp that has been blocked at a semaphore is made runnable (procedure Sema_P).
- Upon exiting from a critical (procedure CS_Exit).
- Upon exiting from a code section that locks Schmon (procedure Schmon_UnlockMonStub).

[3]These will be referred to as Kill, Suspend, Signal and Resume respectively.

- In Sprite signal handlers for the Sprite signals used to implement Kill, Suspend and Signal (procedures Server_Kill_Sig_Handler, Server_Suspend_Sig_Handler and Server_User_Sig_Handler).
- When a message is received from the Async_MboxPtr (procedure Mail_Receive).

8.3 Sprite Signals and Asynchronous Interactions

A lwp only checks for the arrival of asynchronous events at certain given points (Section 8.2). When a lwp is executing non-multiprocessing code, it does not poll for asynchronous events. To force the server process of an executing target lwp to check for asynchronous events, Sprite signals are used. Three distinct user-defined Sprite signals are used in the implementation of Kill, Suspend and Signal. To keep the implementation simple, Sprite signals are disabled in critical sections, as well as in sections of code locking Schmon. When a Sprite signal is delivered to a server process, the corresponding Sprite signal handler (for Kill, Suspend or Signal) checks if it is executing a lwp. If so, it checks the KillFlag, SuspendFlag or SignalFlag of the lwp, and handles the Kill, Suspend or Signal as appropriate.

8.4 Handling Asynchronous Events

If a lwp discovers that its KillFlag is set, it kills itself by invoking the semaphore operation P on its SelfKill_SemaPtr. This indirectly invokes the semaphore operation V on the Killed_SemaPtr of the lwp being killed, releasing the blocked killing lwp.

If a lwp discovers that its SuspendFlag is set, it suspends itself by invoking the semaphore operation P on its SelfSuspend_SemaPtr. This indirectly invokes the semaphore operation V on the Suspended_SemaPtr of the lwp being suspended, releasing the blocked suspending lwp.

If a lwp discovers that its SignalFlag is set, it checks its SigObj_MboxPtr for signal objects of enabled signals. If these are found, the lwp handles them in FIFO order. For each enabled signal object in the mailbox, the lwp receives it from the mailbox, looks up the signal handler, and invokes the signal handler with the signal and the optional arguments in the signal object.

8.5 Schmon and Sprite Signals

This section describes how asynchronous events are checked (and possibly handled) after a section of code locking Schmon (but not inside a critical section).

```
SelfLwpPtr is a pointer to the lwp locking Schmon

    Disable Sprite signals
```

```
SelfLwpPtr->SchmonFlag = TRUE

Lock Schmon

[Actual code inside Schmon]

Unlock Schmon

SelfLwpPtr->SchmonFlag = FALSE

Enable Sprite signals

if (SelfLwpPtr->KillFlag)
    Kill self

if (SelfLwpPtr->SuspendFlag)
    Suspend self

if (SelfLwpPtr->SignalFlag)
    Process enabled signal objects in SigObj_MboxPtr

return
```

8.6 Overview of Kill and Suspend Implementation

To kill or suspend a lwp, the following events occur:

1. Set KillFlag of target lwp for Kill. (Set SuspendFlag of target lwp for Suspend)
2. Send special message to Async_MboxPtr of target lwp. (Kill_Mesg or Suspend_Mesg)
3. Lock Schmon.
4. If target lwp is in Run-State, send Sprite signal to its server process. Different Sprite signals are used for Kill and Suspend.
5. Unlock Schmon.
6. Block untill target lwp kills or suspends itself.

8.7 Semaphores used for Kill and Suspend

The following semaphores are used in the implementation of Kill and Suspend:

- A lwp kills itself by invoking the semaphore operation P on its SelfKill_SemaPtr. See Section 3.6.
- A lwp suspends itself by invoking the semaphore operation P on its Self-Suspend_SemaPtr. It is resumed when another lwp invokes the semaphore operation V on this semaphore. See Section 3.6.
- A lwp calling Kill is blocked on Killed_SemaPtr of its target lwp until the target lwp is killed.
- A lwp calling Suspend is blocked on Suspended_SemaPtr of its target lwp until the target lwp is suspended.

8.8 Signal Implementation

The implementation of Signal is discussed in the following sections. This includes the data structures used to maintain the set of enabled signals, the binding of signals and signal handlers, and the way signals are delivered and handled.

8.9 Overview of Signal Implementation

To signal a lwp, the following events occur:

1. Send signal object to SigObj_MboxPtr of target lwp.
2. Send special message to Async_MboxPtr of target lwp (Signal_Mesg).
3. Set SignalFlag of target lwp.
4. Lock Schmon.
5. If target lwp is in Run-State, send Sprite signal to its server process.
6. Unlock Schmon.

8.10 Data Structures for Signal

The following data structures are used to handle signals:

Signal Object A signal object contains the signaling lwp (for debugging), the signaled lwp (for debugging), the signal, and the optional arguments to the signal handler. It is packaged inside a message object and then sent to the SigObj_MboxPtr of the signaled lwp. The current implementation supports a maximum of 4 optional arguments to a signal handler.

Signal Stack The signal stack of a lwp (SigStack) maintains the set of enabled signals and the (signal, signal handler) pairs of each dynamic extent of the lwp. A stack structure is required because the set of enabled signals before invoking a signal handler is identical to that upon returning from the signal handler. In the current implementation the size of a SigStack may not be expanded dynamically. Signal stacks are allocated in exactly the same way as lwp stacks. See Section 3.5 for details.

Enabled Signals The enabled signals of a dynamic extent of a lwp is maintained as a singly-linked list. The list contains a special signal if all signals are enabled. If not all signals are enabled, the enabled signals are maintained explicitly.

8.11 Set of Enabled Signals

The following operations are used to update and query the set of enabled signals of the current dynamic extent of a lwp:

EnbSig_Enable_Sig Enable a signal in the current dynamic extent.

EnbSig_Enable_All_p Return TRUE iff all signals are currently enabled.

EnbSig_SigObj_Enable_p Return TRUE iff the signal in the signal object argument is currently enabled.

EnbSig_Save Push an empty set of enabled signals onto the SigStack, disabling all signals. The previous set of enabled signals is implicitly saved.

EnbSig_Restore Pop the current set of enabled signals from the SigStack. This restores the set of enabled signals in effect before the last call to EnbSig_Save. The (signal, signal handler) pairs established after the last call to EnbSig_Save are also popped.

8.12 Binding of Signals and Signal Handlers

The following operations are used to maintain the (signal, signal handler) bindings of the current dynamic extent:

Hdl_Push_Handler Push a (signal, signal handler) pair onto the SigStack. Used to implement With-signal-handler.

Hdl_Pop_Handler Pop a (signal, signal handler) pair from the SigStack. Used to implement With-signal-handler.

Hdl_Lookup_Handler Return the current signal handler of the given signal.

8.13 Signal Handling

A signal object is packaged in a message object and sent to the SigObj_MboxPtr of the target lwp. This mailbox enqueues all the unprocessed signal objects of the lwp. The signal objects in this mailbox are handled as follows:

```
while there are signal objects for enabled signals
    in SigObj_MboxPtr do
{
    Receive first enabled signal object
        from SigObj_MboxPtr

    Save all currently enabled signals, and
        disable all signals

    Lookup signal handler for signal

    Invoke signal handler with the signal and
        optional arguments in signal object

    Restore all previously enabled signals
}

clear SignalFlag

return
```

8.14 Mail and Asynchronous Interactions

To preserve the atomicity of Send and Receive, a lwp only checks for asynchronous events when it is safe to do so. Send and Receive are coded as critical sections, within which Sprite signals are disabled. A lwp calling Receive always adds its Async_MboxPtr to the *end* of the sequence of mailboxes it is receiving from. A lwp executing Kill, Suspend or Signal sends a special message to the Async_MboxPtr of the target lwp. This message *cancels* the blocked Receive, and forces the target lwp to check for asynchronous events. When a blocked receiver wakes up because of mail arriving at its Async_MboxPtr, it checks its KillFlag, SuspendFlag and SignalFlag. If the operation is Kill, the target lwp kills itself. If the operation is Suspend or Signal, the target lwp will re-execute the Receive when the lwp is Resumed, or when it has handled all the enabled signal objects.

In the following sections the interactions between asynchronous interactions and mailbox operations are descibed in detail. This includes a discussion on critical sections, including their interactions with Sprite signals. The interaction of asynchronous events and blocked Receive, the most complex part of the mailbox implementation, is then presented. This is followed by a complete description of the algorithm for Send and Receive.

8.15 Critical Sections

Send and Receive are coded as critical sections, within which Sprite signals are disabled. This section describes how asynchronous events are checked (and possibly handled) before and after a critical section.

```
SelfLwpPtr is a pointer to the lwp entering a critical section

    Disable Sprite signals

    SelfLwpPtr->NoInterrupt = TRUE

    [Code for Send or Receive]

    SelfLwpPtr->NoInterrupt = FALSE

    Enable Sprite signals

    if (SelfLwpPtr->KillFlag)
        Kill self

    if (SelfLwpPtr->SuspendFlag)
        Suspend self

    if (SelfLwpPtr->SignalFlag)
        Process enabled signal objects in SigObj_MboxPtr

    return
```

8.16 Asynchronous Interactions and Blocked Receive

Receive is atomic unless the lwp blocks because all the mailboxes are empty. This is implemented as follows:

- Each lwp has a special mailbox Async_MboxPtr. This mailbox is added to the *end* of each mailbox sequence for receive.

- If the receive does not block (mail is found), the receive is atomic, and adding Async_MboxPtr to the mailbox sequence does not make any difference.
- If the receive does block because all the mailboxes are empty, the blocked receive may be broken by an asynchronous action. To do so, a lwp trying to Kill, Suspend or Signal the blocked lwp does the following:

  1. Set KillFlag or SuspendFlag of target lwp for Kill or Suspend.
  2. Send a special message to the SigObj_MboxPtr of the target lwp for Signal. Note that the target lwp is *not* receiving from this mailbox.
  3. Send a special message to the Async_MboxPtr of blocked receiver lwp. This wakes up the blocked receiver and informs it about the asynchronous operation.
  4. Set the SignalFlag of the target lwp for Signal.
  5. For Kill or Suspend, the calling lwp blocks until the operation has completed, i.e. the target lwp enters Suspended-State or Dead-State.
  6. Upon wakeup (after receiving mail), the target (receiver) lwp checks if the message came from its Async_MboxPtr. If so, it checks the message to see which asynchronous operation it should handle.
  7. Target lwp checks KillFlag or SuspendFlag to verify. (See race conditions in Section 8.19.)
  8. For Suspend, the canceled Receive is re-executed when the lwp is resumed.
  9. For Signal, the canceled Receive is re-executed when the lwp has handled all the signal objects of enabled signals.

A Receive that has been canceled by an asynchronous operation uses the C library routines *_setjmp* and *_longjmp* to re-execute the Receive. The code is as follows:

```
SelfLwpPtr is a pointer to the receiver lwp

Mail_Receive_Atomic(MboxSeqPtr, MesgPtrPtr, FromMboxPtrPtr)

Saved_JmpEnv and Value are local variables

    /* Save setjmp environment into local variable */
    Saved_JmpEnv = SelfLwpPtr->JmpEnv

Retry:
    if ((Value = _setjmp(SelfLwpPtr->JmpEnv) == 0)
{
    Enter critical section

    Mail_Receive(MboxSeqPtr, MesgPtrPtr, FromMboxPtrPtr)

    Exit critical section
}
else
{
    goto Retry
}

/* Restore setjmp environment from local variable */
SelfLwpPtr->JmpEnv = Saved_JmpEnv

return
```

8.17 Complete Algorithm for Send and Receive

This section presents the algorithms for Send and Receive in pseudo-code form. The code for critical sections, which surrounds the code for Send or Receive, is presented separately in Section 8.15.

Initial values of Semaphores

```
MboxPtr->Mbox_SemaPtr         1

WaitPrcsPtr->Sender_SemaPtr   1
WaitPrcsPtr->Mail_SemaPtr     0
```

Send

```
Mail_Send(MboxPtr, MesgPtr)

WaitPrcsPtr, SavedMboxSeqPtr and XMboxPtr are local variables

while (TRUE)
{
    P(MboxPtr->SemaPtr)
    if there is no waiting (receiver) lwp in MboxPtr->PrcsQPtr
    {
        Enqueue MesgPtr in MboxPtr->MesgQPtr
        V(MboxPtr->Mbox_SemaPtr)
        return
    }
    else
    {
        /* This does NOT dequeue WaitPrcsPtr */
        WaitPrcsPtr = First(MboxPtr->PrcsQPtr)

        /* Non_Blocking to prevent deadlock */
        if (! P_Non_Blocking(WaitPrcsPtr->Sender_SemaPtr) )
        {
            V(WaitPrcsPtr->Sender_SemaPtr)

            /* This is necessary if server processes are not
               time-sliced
            */
            Voluntary context switch
            continue
        }

        /* Race with another sender: WaitPrcsPtr is blocked on
           2 or more mboxes
        */
        if (WaitPrcsPtr->MboxSeqPtr == NULLP(MboxSeqType))
        {
            V(WaitPrcsPtr->Sender_SemaPtr)

            /* Unlock MboxPtr so that the other sender can
               dequeue WaitPrcsPtr from MboxPtr
            */
            V(MboxPtr->Mbox_SemaPtr)
        /* This is necessary if server processes are not
           time-sliced
        */
        Voluntary context switch
        continue
    }
    else
    {
        Dequeue WaitPrcsPtr from MboxPtr->PrcsQPtr
        WaitPrcsPtr->From_MboxPtr = MboxPtr
        WaitPrcsPtr->Mail_Drop    = MesgPtr
        SavedMboxSeqPtr = WaitPrcsPtr->MboxSeqPtr

        WaitPrcsPtr->MboxSeqPtr = NULLP(MboxSeqPtr)
        /* Only after setting WaitPrcsPtr->MboxSeqPtr
           to NULL
        */
        V(WaitPrcsPtr->Sender_SemaPtr)

        /* Unlock MboxPtr so that only 1 mbox is locked
           at a time to prevent deadlocks
        */
        V(MboxPtr->Mbox_SemaPtr)

        /* Sender dequeues receiver from all other mailboxes */
        for each XMboxPtr in SavedMboxSeqPtr do
        {
            if (XMboxPtr != MboxPtr)
            {
                P(XMboxPtr->Mbox_SemaPtr)
                Dequeue WaitPrcsPtr from XMboxPtr->PrcsQPtr
                V(XMboxPtr->Mbox_SemaPtr)
            }
        }

        /* This must come after WaitPrcsPtr is dequeued from
           all mailboxes in SavedMboxSeqPtr, or else WaitPrcsPtr
           may start another receive, setting its MboxSeqPtr,
           so that another sender will dequeue it from the
           mailbox before the above code (loop) does so.
        */
```

```
                V(WaitPrcsPtr->Mail_SemaPtr)

                return

            }
        }
    }
```

Receive

RecLwpPtr is a pointer to the receiver lwp

Mail_Receive(MboxSeqPtr, MesgPtrPtr, FromMboxPtrPtr)

EmptyMboxPtr, EmptyMboxSeqPtr and MboxPtr are local variables

```
    P(RecLwpPtr->Sender_SemaPtr)

    /* See if this is a retry after a canceled receive */
    if (LastElt(MboxSeqPtr) != RecLwpPtr->Async_MboxPtr)
        add RecLwpPtr->Async_MboxPtr to tail of MboxSeqPtr

    RecLwpPtr->MboxSeqPtr = MboxSeqPtr
    EmptyMboxSeqPtr = empty mailbox sequence

    for each MboxPtr in MboxSeqPtr do
    {
        P(MboxPtr->Mbox_SemaPtr)
        if there is message in MboxPtr->MesgQPtr
        {
            *MesgPtrPtr = first message dequeued from
                          MboxPtr->MesgQPtr
            *FromMboxPtrPtr = MboxPtr

            /* Before cleanup loop to prevent deadlocks */
            V(MboxPtr->Mbox_SemaPtr)

            for each EmptyMboxPtr in EmptyMboxSeqPtr
            {
                P(EmptyMboxPtr->Mbox_SemaPtr)
                Dequeue RecLwpPtr from EmptyMboxPtr->PrcsQPtr
                V(EmptyMboxPtr->Mbox_SemaPtr)
            }

            RecLwpPtr->MboxSeqPtr = NULLP(MboxSeqPtr)
            V(RecLwpPtr->Sender_SemaPtr)

            if ( (*FromMboxPtrPtr) == RecLwpPtr->Async_MboxPtr )
            {
                switch ((*MesgPtrPtr)->MesgClass)
                {
```

```
            case Kill_Mesg:
                if (RecLwpPtr->KillFlag)
                {
                    Kill self
                }
                break;

            case Suspend_Mesg:
                Exit from critical section
                if (RecLwpPtr->SuspendFlag)
                {
                    Suspend self
                }

                /* RecLwpPtr being Resumed */
                /* Redo receive by jumping back to
                   Mail_Receive_Atomic*/
                _longjmp(RecLwpPtr->JmpEnv)
                break;

            case Signal_Mesg:
                Exit from critical section
                Handle signal objects in
                    RecLwpPtr->SigObj_MboxPtr

                /* Redo receive by jumping back to
                   Mail_Receive_Atomic*/
                _longjmp(RecLwpPtr->JmpEnv)
                break;
        }
    }
    else
    {
        return
    }

}
else
{
    Enqueue RecLwpPtr in MboxPtr->PrcsQPtr
    Add MboxPtr to EmptyMboxSeqPtr
    V(MboxPtr->Mbox_SemaPtr)
```

```
        }

    }

    /* By now all the mailboxes in MboxSeqPtr are found
       to be empty
    */
    V(RecLwpPtr->Sender_SemaPtr)

    /* A sender lwp may get in at this point, but this does not
       create a problem since receiver will bolck on Mail_SemaPtr
    */
    P(RecLwpPtr->Mail_SemaPtr)

    /* Mail has been delivered to RecLwpPtr when it wakes up */

    *MesgPtrPtr     = RecLwpPtr->Mail_Drop
    *FromMboxPtrPtr = RecLwpPtr->From_MboxPtr

    RecLwpPtr->Mail_Drop     = NULLP(MesgType)
    RecLwpPtr->From_MboxPtr = NULLP(MboxType)

    if ( (*FromMboxPtrPtr) == RecLwpPtr->Async_MboxPtr )
    {
        switch ((*MesgPtrPtr)->MesgClass)
        {
            case Kill_Mesg:
                if (RecLwpPtr->KillFlag)
                {
                    Kill self
                }
                break;

            case Suspend_Mesg:
                Exit from critical section
                if (RecLwpPtr->SuspendFlag)
                {
                    Suspend self
                }

                /* RecLwpPtr being Resumed */
                /* Redo receive by jumping back to
                Mail_Receive_Atomic*/
             _longjmp(RecLwpPtr->JmpEnv)
             break;

         case Signal_Mesg:
             Exit from critical section
             Handle signal objects in
                 RecLwpPtr->SigObj_MboxPtr

             /* Redo receive by jumping back to
                Mail_Receive_Atomic*/
             _longjmp(RecLwpPtr->JmpEnv)
             break;
     }
}

return
```

8.18 Design Decisions

- The Asynchronous module is implemented on top of the Mailbox module. The implementation is highly portable; it does not contain any machine-dependent code. A machine-dependent implementation of the Asynchronous module may be more efficient, but much more complex. This can only be justified if the asynchronous operations are found to be used heavily.

- Send and Receive are coded as critical sections to simplify the implementation. In the *typical* case a Receive involves only a small number of mailboxes. Each mailbox in the sequence is only locked for a short, finite duration. Since the time spent by a lwp in a critical section is typically short, the *effect* of an asynchronous action may be delayed until the target lwp is no longer in the critical section. This includes the case when the target lwp blocks after discovering that all mailboxes are empty. In the worst case, however, this implementation may take an arbitrary amount of time for an asynchronous operation to take effect. (This happens when the target lwp is trying to gain access to a mailbox being locked by another lwp.)

- Sprite signals are disabled within critical sections and sections of code locking Schmon. This is done because Sprite system calls, which are used to enable and disable Sprite signals, are relatively inexpensive. This simplifies the implementation considerably, at a small cost to efficiency.

- When a blocked Receive is awaken by an asynchronous operation, the C library routines *_setjmp* and *_longjmp* are used to execute a non-local exit. To make this scheme recursive, the (previous) *_setjmp* environmemt must be saved in a local variable (on the stack) before Receive is called, and restored if the Receive completes normally. On a SPUR workstation, the large number of hardware registers on a processor that must be saved by *_setjmp* and *_longjmp* make this implementation rather expensive. An alternative way of handling canceled Receive operations has to be developed.

8.19 Race Conditions

Here are some of the race conditions that have been detected and corrected:

- A Sprite signal may be delivered to a server process that is no longer running the target lwp for which the Sprite signal is intended. When a Sprite signal arrives at a server process, the Sprite signal handler checks if it is executing a lwp. If so, the KillFlag, SuspendFlag or SignalFlag of the lwp are examined as appropriate. If the proper flag is not set, the Sprite signal handler simply returns.

- A special message (Suspend_Mesg or Signal_Mesg) is sent to the Async_MboxPtr of a target lwp as part of a Suspend or Signal. For Suspend, the SuspendFlag of the target lwp is set before the Suspend_Message is sent to the Async_MboxPtr of the target lwp. For Signal, the signal object is sent to the SigObj_MboxPtr of the target lwp before the Signal_Mesg is sent to the Async_MboxPtr of the target lwp.

  By the time the special message (Suspend_Mesg or Signal_Mesg) is received from its Async_MboxPtr (in the course of a receive), the lwp may have already processed the corresponding Suspend or Signal. To prevent the arrival of the special message at its Async_MboxPtr from causing the asynchronous operation from being repeated (erroneously), the lwp checks its SuspendFlag (for Suspend_Mesg) or SigObj_MboxPtr.

9 Sprite Features Used

This section describes the features of the Sprite operating system used by the implementation. The system can be ported easily to another operating system that supports multiple processes in a shared address space.

- Proc (proc.h): Proc_Fork, Proc_Detach, Proc_Wait, Proc_Exit
  User-level processes in a shared address space
- Monitor (syncMonitor.h): LOCK_MONITOR, UNLOCK_MONITOR, Sync_GetLock, Sync_Unlock, Sync_Wait, Sync_Broadcast, Sync_SlowWait
  Mesa-style monitors and conditions
- Vm: Vm_CreateVA, Vm_DestroyVA, Vm_PageSize
  Create and destroy virtual memory for a process
- Clock (time.h): Sync_WaitTime
  Interval timer
- Signal (sig.h): Sig_Send, Sig_HoldMask, Sig_SetAction
  Asynchronous signals, disable and enable signals, signal handlers
- C library routines:
  - Input output (io.h): Io_PrintStream, Io_Flush
  - String (string.h): String_Length, String_Copy
  - Memory allocation (mem.h): Mem_Alloc, Mem_Free
  - Non-local exit (setjmp.h): _setjmp, _longjmp

10 Implementation Status

This section describes the major problems that must be solved before a multiprocessing SPUR Lisp system will be available on the SPUR workstation:

- C/Lisp interface: memory allocation in C routines and garbage collection.
- SPUR port: machine-dependent code.
- Lisp system integration: adapting the uniprocessing SPUR Lisp system for multiprocessing.

11 Known Problems

- Signal handlers can only have a maximum of 4 (optional) arguments.
- The Async_MboxPtr mailbox of a receiver lwp is added to the end of the mailbox sequence. This mailbox should be removed from the mailbox sequence after the Receive.

12 Summary

We described an implementation of Multiprocessing SPUR Lisp under the Sprite operating system on Sun workstations. We examined the major parts of the system, including the Lightweight Process Scheduler, the Synchronization module, the Mailbox module, the Future module, and the Asynchronous module.

Our aim was to develop a system that was simple, easy to port to a different hardware architecture, and reasonably efficient for frequently-used operations. Simplicity was achieved by optimizing for frequently-used operations, and by implementing infrequent operations using existing primitives. The implementation strategy was to construct a simple but working prototype, measure its performance, and then optimize its performance for the frequently used operations. As lwp creation and mailbox operations are assumed to be used much more frequently than asynchronous operations, the Asynchronous module was implemented in a completely machine-independent manner using the Mailbox module (Section 8). As the usage pattern of futures has not been established, the Future module was also implemented using the Mailbox module (Section 7). This strategy greatly reduces the number of low-level primitives required in the implementation.

Since SPUR Lisp processes are expected to be used heavily, they are implemented as lightweight processes and not as Sprite processes. This reduces the overhead of Lisp process creation and synchronization (Section 3).

Lwp scheduling is performed in FIFO order, and the lwp scheduler does not distinguish between lwps created by the user (using Make-process) and lwps created by the system (when Send-after-delay, Future or Delay is invoked). More studies on actual usage of these features will be needed to determine if a more sophisticated scheduling strategy should be used (Sections 3.1 and 7.3).

Portability across architectures is important because the system has been developed on Sun-3 workstations running the Sprite operating system, and will be ported to a multiprocessor SPUR workstation running Sprite when the hardware becomes available. To facilitate the SPUR port, the amount of machine-dependent code has been kept to a minimum. Machine-dependent routines are (only) used to allocate and access the process control blocks (saved hardware state) and runtime stacks of lwps. It is estimated that the SPUR port can be accomplished in a matter of days.

Following the above principles, a reasonably clean implementation of the multiprocessing features of SPUR Lisp has been developed in spite of the complexity introduced by the asynchronous operations (in all parts of the system). The effort to improve the performance of the system is described in Appendix F.

A MP Lisp Constructs and C Interface

In this section we list the C procedural interface that implements the multiprocessing SPUR Lisp extensions. [4]

A.1 Process

- `make-process` *closure* `&optional` *process-name* [Function]

  Sch-New-Lwp(FnPtr, StringName)
  FnPtr: entry point of *closure*
  StringName: *process-name*

- `processp` *object* [Function]

  (No C interface. This is done by checking the type of *object*.)

- `process-state` *process* [Function]

  Prcs-External-State(PrcsPtr)
  PrcsPtr: *process*

A.2 Mailbox

- `make-mailbox` `&optional` *mailbox-name* [Function]

  Mbox-New(StringName)
  StringName: *mailbox-name*

- `mailboxp` *object* [Function]

  (No C interface. This is done by checking the type of *object*.)

- `send` *message mailbox* [Function]

  Mail-Send-Atomic(MboxPtr, MesgPtr)
  MboxPtr: *mailbox*
  MesgPtr: *message*

- `send-after-delay` *message mailbox delay* [Function]

[4] C routine names of the form Foo_Bar are formatted as Foo-Bar here.

DSend-Delay-Send(MboxPtr, MesgPtr, Interval)
MboxPtr: *MboxPtr*
MesgPtr: *message*
Interval: *delay*

- `receive` *mailbox-sequence* [Function]

  Mail-Receive-Atomic(MboxSeqPtr, MesgPtrPtr, FromMboxPtrPtr)
  MboxSeqPtr: *mailbox-sequence*
  MesgPtrPtr: message is returned here
  FromMboxPtrPtr: mailbox the message was sent to

- `mailbox-empty-p` *mailbox* [Function]

  Mbox-Empty-P(MboxPtr)
  MboxPtr: *mailbox*

- `mailbox-message-count` *mailbox* [Function]

  Mbox-Message-Count(MboxPtr)
  MboxPtr: *mailbox*

A.3 Asynchronous Interactions

- `kill-process` *process* [Function]

  Async-Kill-Prcs(PrcsPtr)
  PrcsPtr: *process*

  Server-Kill-Sig-Handler(Signal-Number)
  Signal-Number: (internal)

- `suspend-process` *process* [Function]

  Async-Suspend-Prcs(PrcsPtr)
  PrcsPtr: *process*
  Server-Suspend-Sig-Handler(Signal-Number)
  Signal-Number: (internal)

- `resume-process` *process* [Function]

  Async-Resume-Prcs(PrcsPtr) PrcsPtr: *process*

- `signal-process` *signal process &rest arguments* [Function]

  Async-Signal-Prcs(PrcsPtr, SignalPtr,SigArgPtr)
  PrcsPtr: *process*
  SignalPtr: *signal*
  SigArgPtr: *arguments*

  Server-User-Sig-Handler(Signal-Number)
  Signal-Number: (internal)

- `with-signal-handler` *({(name signal-handler)}*) {form}** [Special Form]

  Hdl-Push-Handler(SignalPtr, HandlerPtr)
  SignalPtr: *name*
  HandlerPtr: *signal-handler*

  Hdl-Pop-Handler()

  Hdl-Lookup-Handler(SignalPtr)
  SignalPtr: *name*

- `enable-signal` *signal* [Function]

  EnbSig-Enable-All(SigStackPtr)
  SigStackPtr: (internal)

  EnbSig-Enable-Sig(SignalPtr)
  SignalPtr: *signal*

  EnbSig-Disable-Sig(SignalPtr)
  SignalPtr: *signal*

- `*self-process*` [Variable]

  CUR-PRCS-PTR (PerServerPtr->CurPrcsPtr)

A.4 Future

- `future` *form* [Special Form]

Fut-Create-Future(FnPtr)
FnPtr: entry point of *form*

Fut-Touch-Future(FuturePtr)
FuturePtr: (internal, object returned by Fut-Create-Future)

Fut-Compute-Future(FnPtr, FuturePtr)
FnPtr: entry point of *form*
FuturePtr: (internal, object returned by Fut-Create-Future)

- `delay` *form* [Special Form]

  Fut-Create-Delay(FnPtr)
  FnPtr: entry point of *form*

- `futurep` *object* [Function]

  (No C interface. This is done by checking the type of *object*.)

- `delayp` *object* [Function]

  (No C interface. This is done by checking the type of *object*.)

- `future-eq` *obj1 obj2* [Function]

  (No C interface. This is done by checking the types of *obj1 and obj2*.)

B Modules

The partition of the various modules into files is described as follows.

Process Scheduler (Files: Pcb-stack, Prcs, Stack-sun, Prcsfifo, Switch, Server)

- Pcb-sun: Saved hardware registers and stack base of C stack
- Prcs: Lightweight processes (lwp)
- Stack-sun: C stack for lightweight processes
- PrcsFifo: FIFO implementation of lwp queues used by the scheduler, semaphores and mailboxes. The optimized version is used by the scheduler and semaphores, and the unoptimized version is used by mailboxes.
- Switch: Lwp scheduler
- Server: Server routines and signal handlers

Synchronization (Files: Lock, Schmon, Sema)

- Lock: Sprite-style locks (Sync_GetLock, Sync_Unlock) and Mesa-style monitor routines (Sync_Wait, Sync_Broadcast)
- Schmon: Scheduler monitor routines (lock, unlock, wait, broadcast)
- Sema: Counting semaphores for lwps

Mailbox (Files: Mesg, Mesgfifo, Mbox, Mboxseq, Mbox2, Dsendfifo, Dsend)

- Mesg: Message object
- MesgFifo: FIFO of undelivered message objects in a mailbox
- Mbox: Mailbox object
- MboxSeq: Mailbox sequence
- Mbox2: Send and Receive
- DSendFifo: FIFO of information about unprocessed invocations of Send-after-delay
- DSend: Send-after-delay

Future (File: Future)

- Future: Implementation of future and delay using mailbox routines

Asynchronous Interaction (Files: Sigobj, SigStack, Enbsigfifo, Enbsig, Handler, Signal, Kill, Suspend)

- SigObj: Signal object

- SigStack: Stack keeping track of enabled signals and signal handlers
- EnbSigFifo: FIFO of enabled signals
- EnbSig: Enabled and disabled signals
- Handler: Signal handlers
- Signal: Signal
- Kill: Kill
- Suspend: Suspend and Resume

Clock (File: Clock)

- Clock: Sprite interval timer used by Send-after-delay

Machine Dependencies (File: Mach-sun)

- Mach: Lwp creation, context switching, test-and-set

Main (Files: Main, Cmdline, Debug, Memhack)

- Main: Top-level, miscellaneous initialization
- Cmdline: Command line options
- Debug: Output with monitor lock, error routines
- Memhack: Fast version of memory allocator (Mem_Alloc)

C Size of the Implementation

This section presents the total size of the implementation, as well as the relative sizes of the various modules. Debugging aids and sanity checks in the code have *not* been commented out.

C.1 Source Code Size

Files	Lines	Number of files
C source files	11100	30
C header files	1500	32
Assembly language files	300	1
Total	12900	63

Table 1: Source Code Size

C.2 Relative Module Sizes

This section presents the relative sizes of the various modules, measured in lines of source code.

Module	Lines	Relative Size (%)
Lwp Scheduler	2500	22
Synchronization	1350	11
Mailbox	3000	27
Future	350	3
Asynchronous	2950	26
Clock	50	<1
Main	900	8
Total	11100	100

Table 2: Relative Module Sizes

D Parameters

The current values of these parameters are in parentheses.

- Lwp Stack:

  MAX-NUM-STACK Maximum number of lwp stacks (100)

  VALID-PAGES-PER-STACK Number of valid operating system pages in each lwp stack (3)

  MAGIC Magic number pushed on top of a lwp stack during a context switch (0xFEEDBABE)

- SigStack:

  MAX-NUM-SIGSTACK Maximum number of signal stacks (100)

  VALID-PAGES-PER-SIGSTACK Number of valid operating system pages in each signal stack (1)

- MemHack:

  MEM-HACK-SIZE Total amount of storage that may be allocated by the fast (simple) memory allocator (0x1000000 bytes)

- Server:

  NUM-SERVER-PRCS Number of Sprite server processes (default 1)

E Storage Allocation

This section presents the size of various objects, and the storage allocated by various operations. Fields used for debugging have been commented out.

E.1 Object Sizes

In the current implementation a pointer occupies 4 bytes.

Object	Bytes
Sync_Lock	8
Sync_Condition	4
PerServerType	16
PcbType	68
PrcsType	616
SemaType	32
PrcsFifo	12
MesgType	8
MesgFifo	8
MboxType	68
MboxSeqType	8
DSendInfoType	12
DSendFifo	8
Future	128
EnbSigFifo	8
SigStackType	16
SigObjType	20
jmp_buf	60
link element of Fifo	8

Table 3: Object Sizes

E.2 Stack Sizes

	Pages	Bytes
Sprite Page	1	8192
Lwp Stack	3+1	32768
Signal Stack	1+1	16384

Table 4: Stack Sizes

E.3 Allocation per Operation

This section summarizes the amount of memory allocated dynamically by each operation. Operations not shown do not allocate memory.

Operation	Bytes
Make-process	616
Make-mailbox	68
Send (no blocked receiver)	16
Send (with blocked receiver)	16
Send-after-delay (no blocked receiver)	1268
Receive (non-empty mailbox)	32
Multiple receive (3rd of 5 non-empty)	48
Kill-process	24
Suspend-process	24
Resume-process	0
Signal-process	52
With-signal-handler	0
Enable-signal	8
Future	744
Delay	128

Table 5: Storage Allocation Per Operation

F Speed of Operations

This section presents performance numbers for various multiprocessing SPUR Lisp operations. The current implementation has vast potentials for performance improvement. Possible modifications are identified and discussed regarding the potential performance gain and the ease of change. A major performance tuning effort is currently under progress, and the performance of the tuned system will be reported in an upcoming paper [7].

F.1 Measurement Conditions

The measurements are performed on a diskless Sun-3/75 (16.7 MHz MC68020) workstation with 8 (or 16) MBytes of physical memory running the Sprite operating system. All debugging code has been commented out. Only the C runtime system is measured; the C/Lisp interface is not included. The Sun-3 timer has a resolution of 20 milliseconds.

F.2 Preliminary Performance Numbers

This section presents the performance figures of the implementation. Table 6 shows the performance of the implementation as described. Table 7 shows the performance of the improved implementation in which Sprite signals are not disabled (Section F.3). Performance of the asynchronous operations have not been measured.

Operation	Time (μs)	Trials	Servers	Lwps
Make-process	7200	100	1	1(+1)
Make-mailbox	200	200	1	1
Send (no blocked receiver)	500	200	1	1
Send (with blocked receiver)	1000	100	1	1+1
Send-after-delay (no blocked receiver)	400	200	1	1(+1)
Receive (non-empty mailbox)	800	200	1	1
Multiple receive (3rd of 5 non-empty)	1600	200	1	1
Future	7500	100	1	1(+1)
Delay	350	200	1	1
Create+Touch future (+ Context Switches)	12700	100	1	1(+1)
Create+Touch delay (+ Context Switches)	12000	100	1	1(+1)
Enter+Exit Critical Section	300	10000	1	1
Lock+Unlock Schmon	330	10000	1	1
Lock+Unlock Schmon in Critical Section	40	10000	1	1
Create+Run Lwp (+ Context Switches)	10200	100	1	1(+1)
Setjmp (+save+restore environment)	77	1000000	1	1
MemHack-Alloc(8)	12	10000	1	1

Table 6: Speed of Multiprocessing SPUR Lisp Operations (Old)

Operation	Time (μs)	Trials	Servers	Lwps
Make-process	6600	600	1	1(+1)
Make-mailbox	150	10000	1	1
Send (no blocked receiver)	192	10000	1	1
Send (with blocked receiver)	600	700	1	1+1
Send-after-delay (no blocked receiver)	400	200	1	1(+1)
Receive (non-empty mailbox)	456	10000	1	1
Multiple receive (3rd of 5 non-empty)	1096	5000	1	1
Future	6900	700	1	1(+1)
Delay	280	1000	1	1
Create+Touch future (+ Context Switches)	9800	1000	1	1(+1)
Create+Touch delay (+ Context Switches)	9800	1000	1	1(+1)
Enter+Exit Critical Section	16	10000	1	1
Lock+Unlock Schmon	38	10000	1	1
Lock+Unlock Schmon in Critical Section	36	10000	1	1
Create+Run Lwp (+ Context Switches)	7000	400	1	1(+1)
Setjmp (+save+restore environment)	77	1000000	1	1
MemHack-Alloc(8)	12	10000	1	1

Table 7: Speed of Multiprocessing SPUR Lisp Operations (New)

F.3 Performance Tuning

As discussed in Section 12, the current implementation is optimized for portability and simplicity of design at the price of some performance penalty. This section identifies the main candidates for performance improvement, and estimates the difficulty involved in implementing each change. The changes are chosen to optimize the performance of lwp creation and mailbox operations. No significant effort has been made to identify the performance bottlenecks of the Asynchronous module.

The following items are ranked in decreasing order of the potential performance improvement and the ease of adopting the change.

Not Disable Sprite Signals The synchronous operations (lwp creation and mailbox operations) are expensive because Sprite signals, which are used to implement asynchronous operations, are disabled before a lwp locks the lwp scheduler monitor Schmon or enters a critical section. As Sprite signals are disabled and re-enabled using Sprite system calls, the time required to complete a synchronous operation is increased by the system call overhead significantly. (See Section 8.3.) Table 7 shows the performance of the system after adopting this change.

Pre-Allocation of Objects Lwp creation is slow because of the time spent allocating and initializing the large number of fields and objects associated with a lwp object. If free lists of initialized lwp objects or objects associated with a lwp object (especially semaphores) are maintained, lwp creation can be speeded up significantly without much effort.

Context Switching Currently two context switches are required to switch from one lwp to another: one to switch from the lwp to the server process, and one to switch from the server process to the new lwp. This simplifies the lwp scheduler significantly at a high performance penalty. On the SPUR workstation, this double context switching will lead to even more performance degradation because of the large number of hardware registers on a processor that must be saved and restored. The lwp scheduler should be modified to switch from one lwp to another directly when there are runnable lwps, and only switch to a server process when no runnable lwp exists. (See Section 3.3.)

Reduce Size of Lwp Object Lwp creation is expensive because a lot of time is spent initializing the large number of fields of a lwp object. The size of a lwp object may be reduced by merging some of the special semaphores used by the lwp scheduler to switch out a lwp. This change should be relatively simple to implement, but would complicate the lwp scheduler moderately. (See Section 3.7.)

Future Module The Future module is implemented using the Mailbox module. If futures are found to be used heavily, the Future module should be re-implemented using lower level primitives. (More experience with the actual usage of the multiprocessing SPUR Lisp features in large programs are necessary.) (See Section 7.2.)

Asynchronous Module The Asynchronous module is implemented in a machine-independent way using the Mailbox module and Sprite signals. If the performance of asynchronous operations is important, the module should be re-implemented using lower level primitives (possibly in a machine-dependent way). This modification is expected to be very difficult. (See Section 8.)

Send-after-delay Module The current implementation of Send-after-delay forks a Sprite process for each call to Send-after-delay. If this feature is used heavily, it should be re-implemented so that no Sprite process is forked for each call. This can be done if the Send-after-delay module maintains some form of internal clock. (See Section 6.2.)

Receive and Asynchronous Operations Receive uses the C library routine *_longjmp* to perform a non-local exit if a blocked receiver lwp is awaken by an asynchronous operation. To make this scheme recursive, the state saved by (a previous) *_setjmp* is copied into a local variable before a Receive, and is restored from the local variable if the Receive completes normally. On a SPUR workstation, the large number of hardware registers on a processor that must be saved and restored by *_setjmp* and *_longjmp* make this implementation rather expensive. An alternative way of handling Receive operations that are awaken by asynchronous operations has to be developed. (See Section 8.16.)

References

[1] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In C. A. R. Hoare and Perrott, editors, *Operating Systems Techniques*, pages 72–93. Academic Press, 1972.

[2] Mark Hill, Susan Eggers, James Larus, George Taylor, et al. SPUR: A VLSI multiprocessor workstation. *IEEE Computer*, 19(11):8–22, November 1986.

[3] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[4] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welsh. An overview of the Sprite project. *;login*, 12(1), January 1987.

[5] Benjamin Zorn, Paul Hilfinger, Kinson Ho, and James Larus. SPUR Lisp: Design and implementation. Technical Report UCB/CSD 87/373, Computer Science Division (EECS), University of California, Berkeley, September 1987.

[6] Benjamin Zorn, Paul Hilfinger, Kinson Ho, James Larus, and Luigi Semenzato. Features for multiprocessing in SPUR Lisp. Technical Report UCB/CSD 88/406, Computer Science Division (EECS), University of California, Berkeley, March 1988.

[7] Benjamin Zorn, Paul Hilfinger, Kinson Ho, James Larus, and Luigi Semenzato. Lisp extensions for multiprocessing. In *Proceedings 22nd Hawaii International Conference on System Sciences*, Kailua-Kona, HA, January 1989. To appear.