The M-Machine Operating System

by

Yevgeny Gurevich

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1995

Author ....
Department of Electrical Engineering and Computer Science
August 11, 1995

Certified by ....
William J. Dally
Associate Professor
Thesis Supervisor

Accepted by ....
Frederick R. Morgenthaler
Chairman, Department Committee on Graduate Theses

The M-Machine Operating System

by

Yevgeny Gurevich

Submitted to the Department of Electrical Engineering and Computer Science
on August 11, 1995, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This document details the design and implementation of an operating system written specifically for the M-Machine, a multicomputer currently being designed at MIT. The operating system is designed to be lightweight and flexible, able to support a UNIX-like operating system layer interface to higher-level code, while at the same time exposing machine primitives to user programs in a safe and efficient manner. The operating system's central features are its support for fast and efficient thread creation and built-in memory-coherence to present the view of global virtual memory to user-level programs as well as higher-level protected subsystems. Four core components are presented - the physical and virtual memory managers, the thread manager, and the memory-coherence manager.

Thesis Supervisor: William J. Dally
Title: Associate Professor

Acknowledgments

My participation in the M-Machine project has involved the most exciting and challenging work that I have so far undertaken. I would like to wholeheartedly thank the entire M-Machine team - Nick Carter, Andrew Chang, Marco Fillo, Steve Keckler, and Whay Lee. Special thanks to Nick and Steve for timely feedback on this thesis.

I owe special thanks to Bill Dally, for guiding me through a large and complex project, finding time to give feedback, giving me the opportunity to contribute to the project, and driving me to solve more problems and deal with more complex issues. It has been a true privilege working under your leadership.

To my parents, thanks for hanging in there and supporting me through these last few hectic months. And thanks also to Mark, a terrific brother who's helped out so much.

Finally, to my very good friends at KBL (and visitors), thanks for making KBL such a welcome home to return to after long hours of work. To Hugh, Len, Pete, Steve & Danielle [congrats], and Paulo - many happy smiles.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The M-Machine is a new multicomputer currently being designed at the MIT AI Lab. The machine's hardware features, some radically different from conventional architectures, require a custom operating system. The operating system is meant to define a small collection of powerful primitives which may be used to construct interface layers in order to emulate existing, familiar operating systems. At the same time, this low-level system code attempts to expose novel hardware features to user-level code in a safe manner. For this reason, the low-level OS needs to be efficient and flexible. Flexible, in terms of providing a framework of very general primitives, and efficient in order to allow other system personalities to reside as higher-level layers without significantly impacting performance.

Following the general trend in operating system design, the M-Machine's OS (M-Machine Runtime System or MARS) is a loose collection of managers which work in concert, instead of a single monolithic kernel such as in traditional implementations UNIX. These managers provide the minimum functionality necessary for an operating system - memory-management and process (thread) management. They form the basis for allowing user-level programs to execute on the machine in protected, stable manner. These low-level managers execute on each node of the M-Machine, similar to the microkernel of Amoeba, which performs process and memory management tasks. As in Mach, the collective managers are designed to enable the implementation of a UNIX-like API which sits above the low-level OS layer. Such an implementation can be efficient and at the same time, provide a common and familiar programming environment.

As in Mach and Amoeba, the OS presented in this thesis supports lightweight thread creation which may be used a basis for the much heavier and often inefficient UNIX fork, although newer implementations of UNIX have moved towards this design as well, providing more lightweight process-creation functions.

A novel addition to this operating system is low-level memory-coherence management. Even operating systems such as Mach, which were designed in part to run on multicomputers, do not integrate a core global shared virtual memory system. A distributed shared memory server in Mach would operate at a higher-level. Furthermore, the global shared virtual memory, supported by hardware-based capabilities, allow the OS to employ a single machine-wide memory map which is identical for all processes. This differs from operating systems like UNIX, Amoeba, and Mach, where virtual memory maps depend on the currently-executing process. A single address space simplifies sharing and writing parallel programs. The use of capabilities free the OS from having to use complicated software-based capability schemes to enforce protections on shared memory. Shared virtual memory is as inexpensive to access, and as safe from errant and malicious threads, as a thread's private memory. Like Amoeba but unlike Mach and UNIX, the M-Machine OS does not include a pager. Such an addition requires additional complexity, and design time for a complete I/O system as well.

This thesis is divided into three general sections. In the first, chapter 2 provides a quick overview of particular aspects of the M-Machine architecture which will both shape the design of the operating system and enable it to perform its duties in an efficient manner. Chapter 3 then provides a high-level picture of the M-Machine operating system's structure.

The second section presents more detailed design and implementation of the four central subsystems within the operating system - the physical memory manager, the virtual memory manager, the thread manager, and the memory-coherence manager. These are covered in chapers 4, 5, 6, and 7.

In the last section, the interface for user programs to the system is described in chapter 8, some performance figures are given in chapter 9, and chapter 10 concludes with project status and future work which needs to be done.

Chapter 2

Target Hardware Overview

This section presents a brief description of the machine architecture targetted by MARS - the M-Machine. The M-Machine is a shared-memory superscalar multi-computer being designed at the MIT Artificial Intelligence Laboratory. A detailed architectural design is provided in [4]. At the high level, the M-Machine consists of a mesh of *nodes* serviced by a high-speed network substrate. Each node consists of four *clusters*. Clusters contain an integer, memory, and floating-point unit capable of issuing in parallel on each clock cycle. Multiple register files and other thread state support up to six thread contexts in hardware simultaneously. The clusters may communicate with each other through a dedicated cluster-switch, and to four cache-banks through a memory switch. With this design, the machine's peak issue rate is twelve operations per clock cycle, with up to 12 outstanding memory references being serviced by the individual cache banks at any one time. Several of the machine's distinctive features which greatly affect its operating system design are presented in this chapter. They include (1) hardware primitives to support global shared virtual memory, (2) operations for atomic memory access, (3) hardware-enforced capabilities for memory protection, (4) support for fast context-switching, and the ability to concurrently maintain several thread contexts in hardware, (5) support for message-send primitives at the instruction level, and (6) mechanisms for accessing hardware state through a memory-mapped configuration space.

2.1 A shared-address-space multicomputer

The M-Machine supports a global 54-bit virtual address space across all of its nodes. Local on-node caches are virtually-addressed, a global translation lookaside buffer (GTLB) maintains mappings of virtual addresses to their home nodes, and system software is required to maintain memory coherence between nodes. Memory references which miss in the cache are handled by an external memory interface (EMI) which probes an on-node local translation lookaside buffer (LTLB) to determine which physical page provides backing for the referenced virtual address. Due to the design of the memory system, lines in the cache must be backed by a local physical page so that they may be flushed to external memory by hardware in the event of a cache-line conflict. References which pass to the LTLB but miss there as well result in an event record being generated which allows software intervention.

2.1.1 Hardware primitives for implementing shared-memory

In order to support an implementation of software-based coherent shared memory, the M-Machine architecture maintains two status bits for each 8-word block of virtual memory. These *block-status bits*, signifying whether a line is invalid, read-only, exclusive-clean, or exclusive-dirty, are maintained by hardware in the local translation lookaside buffer and cache. The memory system prevents an access from completing if it violates the block-status bits, instead generating an event record which allows system software to intervene and satisfy the access. There are three fault types: write to an invalid line, read to an invalid line, and write to a read-only line. All events are handled by a dedicated system thread as explained in the next chapter. System software is expected to replicate and manage these block status bits in node page tables when altering entries in the LTLB.

2.1.2 Atomic Test-and-set Memory Operations

In order to support access to global shared data structures in the face of concurrency, each word of the machine's memory includes a lock bit which is referenced in atomic test-and-set memory operations. These synchronizing memory operations allow programs to perform loads or stores conditional upon the status of the lock bit (the precondition), and set the bit to a known value if they succeed (postcondition). Conditional synchronizing memory operations return a condition which is true if the test-and-set succeeded and the memory operation completed, and false otherwise. Unconditional synchronizing memory operations generate events if the preconditions they require are not met. Details of instructions are in [5].

2.1.3 Hardware-supported capabilities

In order to maintain global shared virtual memory without the use of access lists, capabilities are used to enforce memory protection. Words may be tagged as pointers to memory segments of a power of two bytes in length by system software, and given out to user-level processes. User processes are only allowed to copy the pointers as is or to modify their address portion so as to change their offsets within the memory segment that the pointer represents. In this way, it is not necessary for the operating system to maintain separate page tables for each process in a multi-process environment since pointers may not be forged. As explained in [2], this presents a problem when a thread deallocates a segment of virtual memory since the operating system does not know a priori which threads may have been passed this pointer[1]. A global garbage-collection of allocated virtual memory must be performed to find and destroy any clones of pointers to virtual segments before such segments may be deemed clean and available once again for allocation. However, as [2] shows that for large address spaces, reclamation may be performed extremely infrequently. [1] presents a more detailed description of the M-Machine's capabilities. The use of different pointer types to allow efficient user access to system code will be revisited in chapter 8.

[1] Threads may pass pointers around in messages, by writing them into memory shared by other threads, or direct intercluster register-file writes

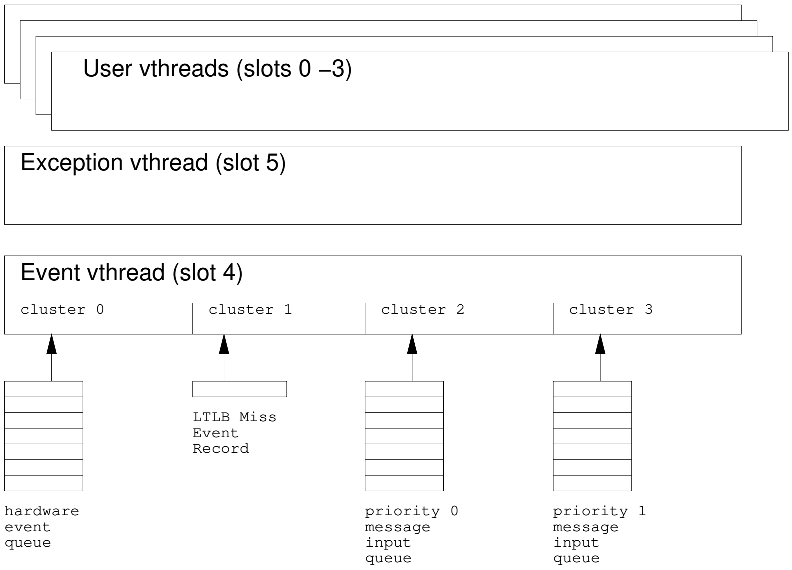

Figure 2-1: Thread Slots and Clusters

2.1.4 V- and H-Threads

A thread which executes on the M-Machine is identified as a V-Thread. It occupies one of six hardware thread slots and may be composed of up to four decoupled H-Threads, each running on a different cluster on the node. H-Threads communicate with each other either through memory or intercluster register-file writes. More details of these mechanisms are given in [4]. At any instant, any combination of four H-Threads from the six different V-Thread slots may be issuing instructions down the piplines of the clusters. V-Threads are round-robin scheduled by the hardware to allow each fair access to machine resources. Four of the hardware thread slots are intended for user-level threads. The remaining two thread slots are meant for system-level handlers. Certain registers in the system-level thread slots are mapped to hardware resources such as the event and network input queues, as shown in figure 2-1. These system thread slots form the core of the M-Machine operating system as will be described in chapter 3. The hardware support for several thread slots allows for efficient context switching among available user threads for better latency tolerance. In addition, there is no expensive penalty for invoking system handlers to respond to events and messages since suspension and eviction of a user thread to make room for a system handler is not required. The handlers are always active, sleeping until an event or incoming message requires their attention. Finally, the controlled manner in which system handlers are invoked - similar to a protection violation invoking kernel mode in traditional operating systems - protects handlers from errant threads since no direct function call is involved.

2.1.5 Support for Efficient Message-passing

Primitive hardware instructions to perform an atomic message send allow threads to inject messages into the M-Machine's internode network without needing to call system software. At the user-level, this allows threads to invoke handlers on a particular node if they obtain (1) an entry pointer into a message-handler routine and (2) a pointer to a virtual memory segment mapped to the destination node. The requirement for a pointer to a message-handler routine ensures that incoming messages are serviced by trusted code which will not lock up the network input queue. At the system level, threads are allowed to send messages directly to physical node numbers instead of using virtual addresses. This message-send primitive is employed by the memory-coherence management software as explained in chapter 7. A ten-word message size limit is adequate for the system software's requirement of shipping 8-word cache lines with some extra status information.

2.1.6 Memory-mapped Access to Hardware State

Threads on the M-Machine may access hardware state through load and store memory operations which target configuration space. Pointers tagged with the configspace type identify such accesses and requests are passed to configuration space controllers in each cluster. Machine state such as the LTLB, portions of the instruction-cache, hardware thread contexts, and status registers may be read and modified by system software. Since the configuration address space is 54 bits, hardware state is laid out sparcely so as to simplify hardware decoding of requested addresses. As will become evident in the rest of this document, configuration-space access will be one of the central tools employed by the runtime system to manage the machine. Since configuration-space pointers allow such powerfull access, these pointers are never given out to user-level threads, and are constructed when needed by priviledged threads or generated directly by hardware state machines on event-record generation.

Chapter 3

Runtime System Overview

The M-Machine Runtime System (MARS) is split into two distinct pieces - system functions which are invoked by user-level threads and execute within the caller's thread slot, and low level handlers which perform physical memory management, memory-coherence, and thread management. System functions allow protected system code execution within a user thread's context with the help of the capabilities mentioned in the previous chapter. A detailed description of system entry and exit is provided in [1]. The operating system presented here is not truly complete, as it lacks a design for I/O, among other components.

The current runtime system implementation uses its own data pointer when invoked as system code, but for simplicity still borrows the caller's stack for intermediate values and spill space. This, however, is actually a potential security leak since a malicious user-level thread may pass a copy of its stack pointer to a confederate (perhaps another H-Thread within its own slot) which may then overwrite portions of the stack or snoop on the stack contents, hoping to encounter a pointer it is not normally allowed to access. MARS can be modified to counter this security problem. A more secure system would employ distinct system stacks inaccessible by the user-level caller. Such stacks may be maintained as linked lists of virtual or physical segments allocated by the OS at boot time, and popped for use by system code when a system function is first invoked. System handlers are invoked indirectly through fault mechanisms and are therefore more secure, using their own dedicated stacks and

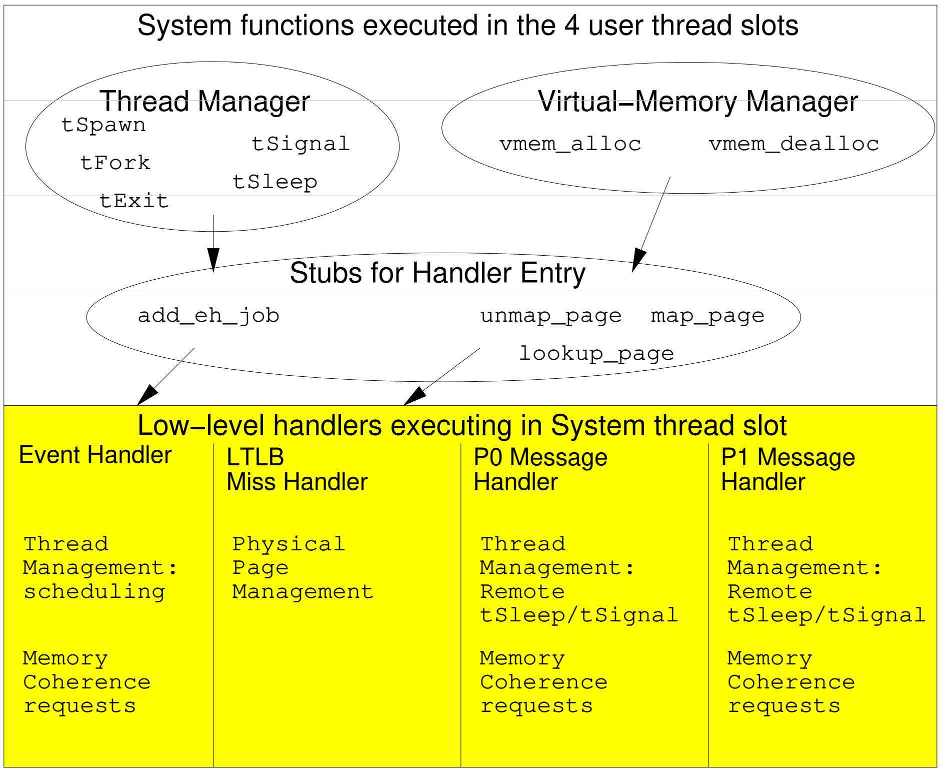

Figure 3-1: OS Components Overview by Hardware

data segment pointers.

All components of the runtime system are designed with several key principles in mind - handlers and system functions are meant to be lightweight, tolerate concurrency, and be flexible and general enough to support a variety of high-level operating systems built from the primitives that they provide.

3.1 Differences from a Traditional Operating System

Unlike traditional operating systems such as the UNIX and Mach variants, MARS is designed from the ground up to support concurrent execution of lightweight threads in a single global virtual address space, and provide a view of coherent virtual memory even to higher-level operating system components. Unlike Mach, where message-passing provides a secure interface for communication, communication among and between user-level processes and operating system components is accomplished through function calls and memory access. This is in great part due to the hardware capabilities which enable protected low-cost shared-memory access, and the underlying memory-coherence protocol. MARS does, however, share several design concepts with Mach, such as the support for synchronization primitives, inexpensive IPC, and lightweight threads/processes.

Instead of a kernel or microkernel of linear code which is invoked by user code through a page-fault mechanism, most system calls in MARS are handled within a user thread slot through protected system entry. This has several key advantages - while system calls are being performed by one user thread, other user threads are not prevented from executing their code. In addition several system calls may be active at any time, even the same system function (system calls must be designed to be re-entrant and use locks when accessing shared data structures.) Finally, more critical services such as the handling of memory protection violations and page faults are still invoked in the traditional fault-reponse manner, but with two key differences. First, while a system event handler is executing other threads may continue issuing until a conflict in a hardware resource arises, in which case the system threads are given higher priority. Second, even the faulting thread may make progress until it requires the use of data which is being serviced by the fault mechanisms. More importantly, even the invocation of critical OS services may be performed concurrently, with three and sometimes four different system-level fault handlers being able to service independent requests at the same time.

A more detailed view of the low level handlers which reside in the system thread slot is provided in the next section.

3.2 System Thread Components

The system V-Thread, running in thread slot four on each node of the M-Machine, is composed of four decoupled H-Threads effectively providing four independent handlers which may concurrently satisfy system events. The four H-Threads are the Event Handler (EH), LTLB Miss Handler, Priority 0 Message Handler (P0MH), and Priority 1 Message Handler (P1MH). All thread components remove events from hardware-based event FIFO queues and process each event in turn. If no events are present, handlers simply block until an event arrives, thereby allowing other threads to issue and not stealing any execution cycles on the machine. An overview of these system components is shown in figure 3-1. Each thread blocks on a different hardware FIFO, allowing up to four events to be handled at a time. Events are usually of fixed-length and are inserted into queues by hardware state machines. Each of the H-Threads in the system V-Thread has integer register 14 mapped to the head of its respective event queue. When that register is used as a source for an operation, the head word of the next event in the hardware queue is used as the data source. Integer register 15 maps to the body of the event, used to access all remaining words in a hardware event. Once an event word is read out of the hardware queue, it is effectively popped from the queue and may not be recovered. Therefore, most handlers store away event words if they are intended to be used multiple times. The following subsections describe each of the four system H-Threads.

3.2.1 Event Handler

The Event Handler responds to block-status miss events, global translation lookaside buffer misses (GTLB Miss), and synchronization misses (SYNC Miss). GTLB misses occur when user-level message-send instructions target virtual addresses which contain no address to node-number mapping in the GTLB. Synchronization misses occur when a synchronizing memory operation fails to proceed because the referenced memory location does not have the requested precondition. The MARS system has not been designed to handle the two latter cases, although future work makes extending the event handler quite simple. In addition, software job queues are used by other components of the OS to request that the Event Handler perform certain tasks, such as evicting or installing threads. A SIGNAL event wakes the event handler to ensure that it gets a chance to examine these software job queues. The handling of block status miss events is a part of the memory-coherence protocol described in chapter 7.

3.2.2 LTLB Miss Handler

The LTLB Miss Handler is the most critical component of the MARS system. Due to the nature of the M-Machine memory system, a miss in the LTLB locks up the external memory interface until that miss is serviced. Other threads may continue issuing operations until such time as they cause a cache miss, in which case memory requests stack up until the EMI is freed by the LTLB Miss Handler. The LTLB Miss Handler itself has a separate path (the *bypass* path) into the EMI which insures that it may always access physical memory. Therefore, when the Miss Handler is executing, there is absolutely no guarantee that any other thread is active and able to make progress on the machine. This makes it especially important that the handler not access any data structures which may be locked by other system threads. Such locked data structures may never be released if the owner of the lock is blocked waiting for the LTLB Miss Handler to free up the EMI for memory accesses. In addition, the handler may only access physically-addressed memory, since a virtual address reference may miss in the LTLB itself, causing deadlock. In its normal mode of operation, the LTLB handler maintains the local page table which contains mappings from virtual to physical pages, and refills LTLB entries on an address miss. The handler may also be called through a faked miss mechanism by system software to create, remove, or lookup the mappings that it creates. This mechanism is decribed in more detail in section 4.5.3. Finally, since the LTLB miss handler cannot guarantee that other portions of the event-handling system are able to make progress, it cannot cause Block-Status, Sync, or GTLB misses, or send messages.

3.2.3 Message Handlers

The P1 and P0 Message handlers receive messages from the network destined for their node and respond by executing message-handling functions. Such functions may implement a variety of mechanisms including remote memory transfer, remote procedure call, and thread spawn. Others form the core of the software memory-coherence implementation. In order to guarantee deadlock-free execution, all request messages are sent via priority 0, and acknowledgements are returned on priority 1. Priority 1 messages are intended to be handled unconditionally, eventually allowing the network to drain of all P1 traffic and allowing all message traffic to make forward progress. For every P0 message received, at most one P1 message should be returned as an acknowledgement.

3.2.4 Availability and Reentrancy

As mentioned previously, since the system handlers reside in an active system V-Thread there is no need to swap in their context and save or restore user thread state before they may begin fulfilling a request. This makes for fast and efficient reponses to what are effectively interrupts without slowing down user code which may be executing concurrently. In addition, there is no time wasted restoring the register-file contents or setting up thread state for the system thread.

The limitations of the LTLB miss handler have already been discussed. In general all event handlers are not reentrant since they are the sole mechanism available for fulfilling their respective event requests. The event handler may not cause and events such as block status, SYNC, or GTLB misses, to occur. In order to maintain the progress guarantees of the network, the P1 message handler may not itself send out messages.

A hardware mechanism prevents user-level threads (those executing in thread slots 0-3) from issuing if the hardware event queue for the event handler rises above a watermark. This mechanism is in place to bound the number of outstanding events in the system and prevent hardware queue overflow. For this reason, it may be possible that protected subsystems which execute within user thread slots may not be able to issue instructions until the event handler has serviced events in the hardware queue. This introduces another constraint upon event handler operation - code executing in the event handler may not wait for locks held by code executing in user thread slots.

3.2.5 Signalling the Event Handler

In some instances other OS components, including message handlers, may request that the Event Handler perform a function, such as a message resend, in proxy for them. This is especially true if a message needs to be sent in response to an ACK arriving at the P1MH. In these cases, a request record is added to a *software job queue* for the Event Handler to fulfill at a future time. A signal event is then issued. In order to avoid overflowing the hardware event queue, only a single signal event may be in the hardware event queue at a time. This is accomplished by keeping a word in memory (the event lockword) on which the handlers may synchronize. System code adding a request for the Event Handler (the producer) first adds the event to a software queue and then attempts to set the lock bit of the event lockword to full. If the word was previously full, the set fails and the producer goes on. If the word was previously empty, the producer adds a new SIGNAL event to the hardware event queue. For its part, the Event Handler always resets the event lockword each time it dequeues the SIGNAL event. This guarantees for the producers that if the lockword is set, a previously-issued SIGNAL is still in the hardware event queue (or recently popped from it) and will be examined by the Event Handler as detailed below.

There are two software job queues because handlers within the system thread slot, and protected subsystems within user thread slots may be attempting to add software jobs for the event handler. All code executing within user thread slots synchronizes access to the job queue so that there is a single producer at a time. A job queue is a ring buffer, with two global pointers into it - the *cur* pointer and the *free* pointer. The cur pointer is read and modified only by the consumer (the event handler thread). It is advanced each time a new event is read out and identifies which events have been read out of the job queue. The free pointer is read by both producer and consumer, but advanced only by the producer. Each time a producer wishes to add a new event to the job queue, it reads the free pointer, and begins storing new event words starting at the free pointer and moving down (wrapping around the end of the event buffer if necessary). As its last action, the producer advances the free pointer with a single store operation. This is the atomic action which signifies that a new event is available. For its part, the event handler checks the cur pointer against the free pointer each time it looks for an event to service. If the cur and free pointers match, no new software events are in the job queue. Otherwise, the event handler may start reading off the cur pointer and advancing it - servicing the next request in the queue.

This mechanism allows the event handler to safely dequeue events without needing to acquire a lock. A second queue is used for the two message handlers to enqueue jobs with the event handler. They also synchronize among themselves to guarantee that there is only one thread adding events to the software job queue at a time.

Given sufficient buffer space to hold all requests, this mechanism is deadlock-free and guarantees that all events in the software queues will eventually be handled. The reason for using the SIGNAL event is to guarantee that requests in the software queue will be examined by the Event Handler if no hardware events are being generated and the Event Handler is blocked waiting for one. The SIGNAL effectively wakes the thread so that it may look at the events stacked up in its software queues. A producer which is unable to set the lockword and therefore add the SIGNAL event is guaranteed that the SIGNAL word is either still in the hardware queue or is just being removed by the Event Handler. In either case, since it had enqueued the request in a software queue prior to attempting a SIGNAL, the producer is guaranteed that the Event Handler will wake up and take a look at the recently added request, as long as the Event Handler runs through the entire software queue before attempting to sleep again. Finally, the lockword also insures that at most one signal has been placed in the hardware queue at a time, preventing queue overflow.

3.2.6 System Call handling in User Thread Slots

Despite the great deal of infrastructure developed for handling events in the dedicated system thread slot, many higher-level system calls may be handled by trusted software running within a caller's user thread slot (V-Thread slots 0 to 3). Such system functions include virtual memory allocation, thread and process creation (but not scheduling), invocation of remote functions or spawning remote threads, bulk memory transfer, and others. In short, most routines made available by high-level operating systems which do not require direct manipulation of low-level data structures such as page tables or memory-coherence directories may be safely executed within a user thread slot. In addition, system calls which work with protected data structures that reside in virtual memory may take full advantage of the memory-coherent global memory supplied by low-level OS components. This layered design provides a lot of flexibility.

3.2.7 Capabilities and Protection

Capabilities enable user threads to enter system functions in a protected manner. The runtime system "exports" a collection of system functions during the loading of user executables. User programs containing references to system functions are patched with entry pointers to runtime system functions by a trusted loader. Entry pointers may be loaded and used in jump instructions, but may not have their addresses changed. This provides a safe entry mechanism since the system functions which are exported are guaranteed to be entered at well-defined points. Since the setting of the pointer bit is a priviledged operation, user programs may not forge entry pointers of their own. This also means that as the OS evolves, the exact entry points and number of available system functions may change, but legacy programs will still execute correctly since patching is performed at load and not link time. In order for system functions to gain access to the runtime's data segment and associated system-level data structures, a system data segment pointer is stored within the system's code segment by the boot code. As a user-level thread enters a system function, the entry pointer is changed to an execute-system-mode instruction pointer which points into the system code segment. This allows the callee system function to load the system data segment pointer by offsetting from its IP (now allowed since the IP is no longer an entry pointer) and performing a load, overwritting the user data segment pointer. On return to the caller, the user's data segment pointer is restored and a jump to a return pointer switches the thread back to user mode. This process is explained in chapter 8.

3.2.8 Page Table Design

The global virtual memory supported by the machine allows the system software to use a single inverted page table to maintain virtual-to-physical page mappings for all allocated memory on each node. The M-Machine uses a 4-Kbyte page size for both physical and virtual pages. An open-hashing page table on a node with 16Mbytes of physical memory requires only 8192 entries to be twice as large as necessary for maximum capacity.[1] Assuming 4 64-bit words per entry, this works out to a 0.2% overhead for an inverted page table. Once again, the advantages of maintaining a single page table for all processes running on the node are clear - no switching of tables is necessary on context switches, speeding up multiprocessing performance. In addition, since capabilities prevent user threads from forging pointers, no additional mechanisms are required to prevent a process from accessing virtual memory allocated for other processes. Finally, no special support is required for shared virtual memory. Once a process gives out a virtual pointer to another thread, the virtual segment may be read and written by both. Several flavors of protected pointers allow processes to set up read-only or read-write shared segments.

Since virtual segments do not necessarily need to be backed by contiguous regions of physical memory, a chained list of physical pages is used by the physical memory manager to dole out backing pages to virtual segments. As physical pages become available for allocation, they are added to the free page chain in a FIFO manner. To speed up allocation, a background process may be used to clean physical pages before they become available for allocation. Physical memory management is detailed in the next chapter.

[1] The M-Machine currently being designed is expected to have 8MBytes of on-node physical memory.

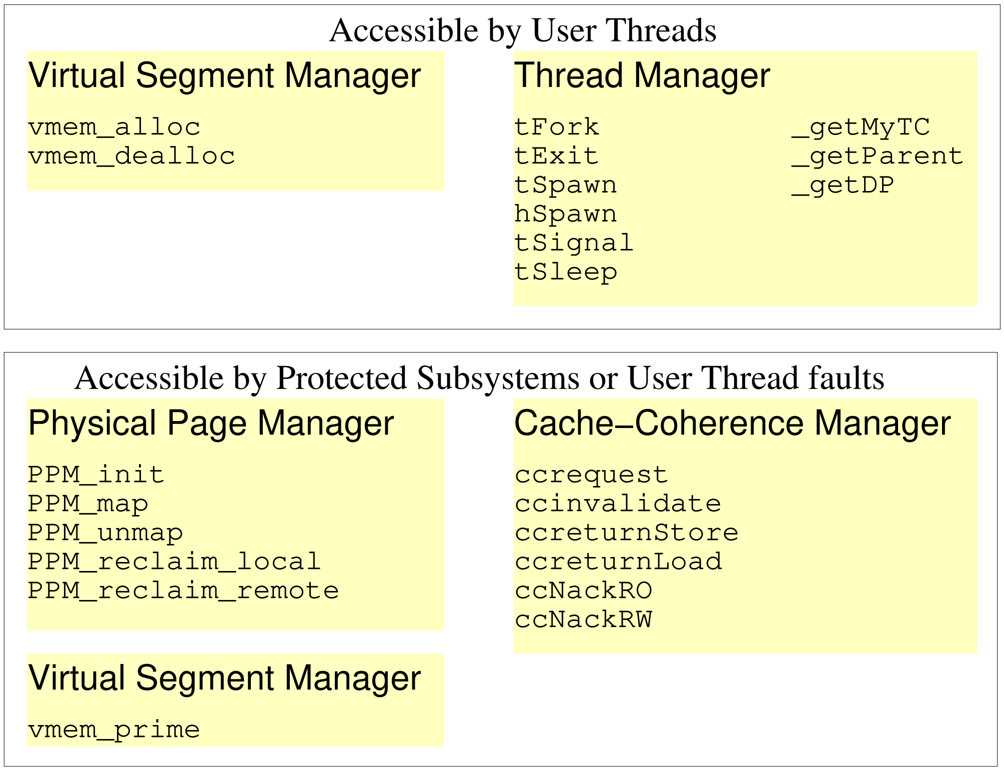

Figure 3-2: OS Components Overview by Function

3.3 Breakdown of MARS into Functional Components

The previous section approached the MARS design from the point of view of hardware resources used for runtime implementation. This section provides an overview of the runtime system as it is broken down into functional components. The runtime system can be viewed as a collection of managers running in a largely autonomous manner to satisfy requests. At times, managers may call upon each other to fulfill certain requests. This is most commonly the case when system threads require access to physical memory - they call upon the physical memory manager to allocate new physical pages or return information on existing virtual-to-physical mappings.

3.3.1 Physical Memory Management

Physical memory management - the maintenance of the local page table and the LTLB - is handled exclusively by the LTLB Miss Handler. Since the handler thread is not allowed access to data structures which may be locked by other threads (as explained in section 3.2.2), there is effectively no overlap in the information which it maintains with that of any other thread. Access to the LTLB Miss Handler is performed in a fault-response manner similar to traditional OS's as described above. In general, misses to reserved virtual addresses which are kept unmapped by the LTLB handler are used as triggers to invoke specific handler functions - such as removal of a particular virtual-physical mapping, creation of a new one, or return of information about an existing mapping.

3.3.2 Virtual Memory Management

Virtual memory is doled out in segments by a Virtual Segment Manager which is composed of a series of system functions accessible by user threads. The VSM does not allocate physical backing to the segments which it gives out, simplifying its design and allowing it to run independent of other pieces of the system software within user thread slots - it does not require access to hardware tables, registers, or other machine state. At boot time, the managers on each node of the M-Machine are primed with virtual segments which they may give out and effectively manage independently. The underlying data structure used for tracking allocated and available segments is the buddy list. Details of the VSM and Buddy list allocation are given in chapter 5.

3.3.3 Memory-coherence Management

The software memory coherence implementation of the M-Machine is centered around the actions performed by Event and Message handlers on each node. The Event Handler on a requesting node sends P0 requests to home nodes in response to local block-status miss events. The P0 message handler on a shared data item's home node receives requests for cache lines, updates a memory-coherence directory and ships out blocks of memory as P1 acknowledgements. The P1 message handler on the original requesting node receives the remote cache line (an implicit acknowledgement to its request) and installs it locally. In the event of cache-line conflicts or flush requests, the Event Handler on a node sharing remote data may be required to invalidate and, in the case of dirty lines, return cache lines to their home nodes. The memory-coherence implementation is detailed in chapter 7.

3.3.4 Process Management

The management of user processes is broken down into two pieces. System calls invoked within user thread slots are used to fork user threads and add them to lists of ready-to-run threads. Other system calls allow threads to sleep, or signal other sleeping threads. The actual manipulation of hardware thread slots for evicting threads and/or installing new ones is performed by the event handler. This localizes access to the machine hardware so that it is performed by a single thread which is guaranteed to be always active. Although not strictly necessary, this localization simplifies aspects of the memory-coherence implementation as detailed in later chapters.

Chapter 4

Physical Memory Management

The M-Machine physical memory manager (PMM) is responsible for maintaining virtual-to-physical page mappings on each node and keeping track of available and allocated physical page frames. Physical memory (sometimes referred to as consisting of backing pages) is usually the ultimate target of memory operations issued on the M-Machine.[1] In the M-Machine memory hierarchy, each node requires a PMM to maintain mappings between virtual pages and their associated physical backing store within a page table. Without a page frame to back it, a virtual address reference cannot be completed. To increase memory-system performance, a 64-entry cache for these mappings is maintained in hardware (the LTLB). The PMM is responsible for keeping the LTLB in sync with the mappings found in the page table. Hardware events notify the PMM when a mapping was not found in the LTLB - an LTLB Miss Event. The PMM must find a mapping within the page table and place it in the LTLB, perhaps evicting a conflicting mapping for a different virtual page. This chapter first introduces a functional interface to the memory-management functions, describes the data structures employed, and details the implementation of the memory manager. As described briefly in the previous chapter, the LTLB Miss Handler is solely responsible for these functions. Section 4.3 explains the rationale behind this design decision.

[1] Exceptions are I/O addresses which are memory-mapped into virtual address space, and configuration-space which is a totally separate address space.

4.1 System Calls

The PMM performs three different functions as part of its management duties - creating virtual-physical mappings, removing these mappings, and returning existing mapping and status information. Interface definitions are shown in table 4.1. These system calls are meant for protected subsystem use and are only exposed to other OS components, not user-level threads.

Function	**Description**
`void PPM_init(int initword)`	Initializes the physical memory manager. The low halfword of *initword* contains the physical page number of the start of unallocated physical memory (the runtime system resides in pages below this page). The high halfword contains the number of pages to add to the local physical page pool (the size of each node's external memory minus the number of page frames consumed by the runtime system).
`int PPM_map(int vpn)`	Creates a mapping between virtual page *vpn* and an available, unallocated physical page frame. There are two pools from which to draw page frames - one for backing local virtual addresses, and one for backing virtual addresses mapped to remote nodes. The page frame is taken from either the local or remote-memory pool, depending on the whether the virtual page is local or remote. Returns the page frame number assigned to the new mapping.
`void PPM_unmap(int vpn)`	Destroys the mapping of virtual page *vpn* with its physical page frame.
`int PPM_reclaim_local(int ppn)`	Returns the page frame *ppn* to the local frame pool.
`int PPM_reclaim_remote(int ppn)`	Returns the page frame *ppn* to the frame pool used for remote backing pages.
`int PPM_lookup(int vpn)`	Returns the number of the frame backing virtual page *vpn*. Returns -1 if no mapping is found.

Table 4.1: Physical Page Manager exported functions

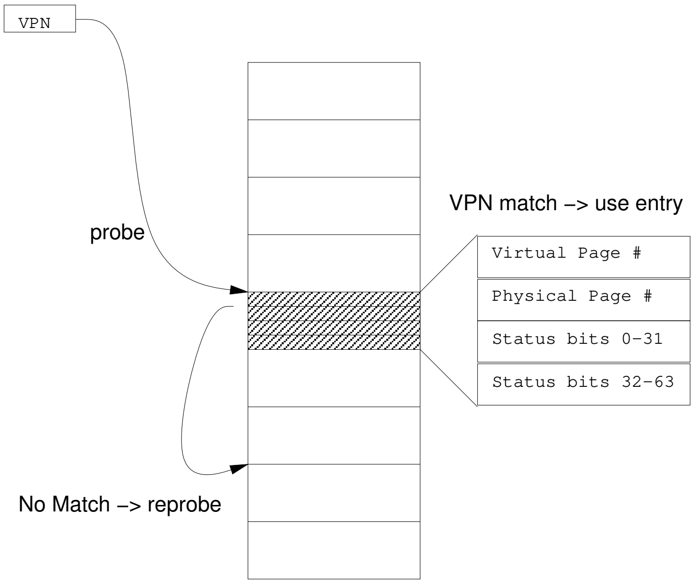

Figure 4-1: PPM Hash Table Structure

4.2 Data Structures

Two main data structures are employed by the PMM. First, the page table is an open hash, used to store virtual to physical mappings and block-status information. The hash table is initialized at machine boot time and sized so that it has room for twice as many mappings as there are page frames on a node. Since the page table is not hierarchical but a hash table, having a large, potentially sparse table is critical for performance reasons - too small a table will result in many conflicts and longer lookup times. Open hashing tables, as described in [3], tolerate entry conflicts without employing chains (linked lists of entries which map to the same location in the hash table) thereby increasing average-case performance. Each hash table entry consists of four words - the actual virtual page number used to define the mapping, its associated physical page number, and 128 block status bits[2] packed into two words (see figure 4-1.)

In order to actually allocate backing pages for virtual pages, the PMM needs to

[2]Each page contains 512 words divided into 64 8-word cache lines. 2 Block status bits for each of the 64 cache lines require a total of 128 bits.

maintain a list of all unallocated page frames. The most efficient data structure is a chain of page frame numbers which resides within the unallocated frames themselves. The PMM maintains a single 64-bit word which contains the page frame number of the next unallocated frame which may be used as a backing page. The first word within that frame itself contains the page frame number of the next frame to use. Thus, a chain of available frame numbers is maintained within the unallocated frames. A page frame chain is terminated by a `-1` which is never expected to be a valid page frame number. In figure 4-2, the free page frame chain starts at page 15, and terminates at page 2. There are a total of 6 frames in the chain. Popping a new frame for use simply requires reading out the frame number from the frame about to be used and substituting it in the pointer to the next available frame. A list of allocated frames is not required since that information is implicitly stored within the hash table. Any frame popped from the free frames chain must be used in a hash table entry. Conversely, removing a frame from the hash table requires that it be added to the free frames chain. A second 64-bit word stores the frame number which is the last available frame in a chain. This makes returning pages to the page frame chain very simple - the tail frame's next frame entry is modified from *-1* to the frame being added to the chain. The tail frame number is then changed to reflect a new end-of-chain page frame number.

Since the memory-coherence system is so closely tied to the OS, special provision for frames which are used as backing for shared cache lines is made in the PMM. Instead of maintaining a single chain of available page frames, two chains are employed. The first is used to allocate normal backing pages for local data. The second is a limited collection of frames, perhaps some fraction of total on-node memory, which may be used as backing pages for shared cache lines. Once this pool is exhausted, shared cache lines must be evicted until an entire frame is freed up, at which time it will become available for allocation as a backing page of remote data again. The `PPM_reclaim_remote` call explicitly tells the PPM that a particular frame has been cleaned and should be added to the remote backing page pool. This particular aspect of page management is discussed further in chapter 7.

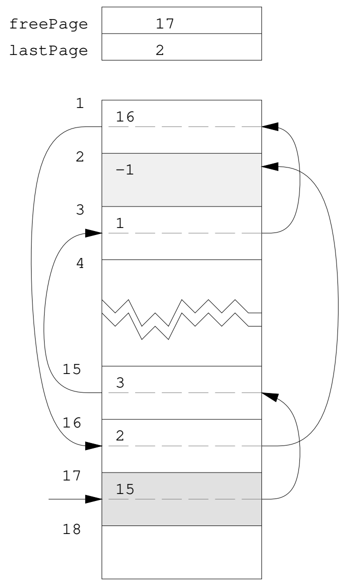

Figure 4-2: PPM Free Page List

4.3 Design Rationale

The reason for placing physical memory management in the hands of a system level handler instead of trusted code which may execute in user-level thread slots is tied closely to the M-Machine's memory system design. Since the LTLB miss handler needs access to the page table in order to insert and remove LTLB entries, no other software components may lock the page table data structure (as explained in section 3.2.2). Since locking the local page table is not allowed, there is only a single software component which remains able to access the page table - the LTLB miss handler itself. Because access to the miss handler is restricted to hardware-generated events which occur only on TLB misses, a few system-level routines act as wrappers around the special miss-response interface. These wrappers allow other system components to make standard function calls which in turn result in forced LTLB misses to reserved virtual page numbers. This implementation is detailed in section 4.5.3.

4.4 Page Allocation Policy

The PMM employs on-demand page frame allocation. That is, if the LTLB Miss Handler does not find a virtual-physical mapping in the on-node page table for a memory access which touched a particular page, it is assumed that a new mapping needs to be created. This allows efficient use of very large and sparse virtual segments - allocation of a virtual segment does not mean that physical backing needs to be created immediately. Instead, individual LTLB misses to virtual pages cause page frames to be allocated. In fact, the current runtime system only employs the `PPM_map` function when performing memory-coherence management, since the common case is for mappings to be created on-the-fly by this automatic allocation policy.

In standard operating systems, a memory reference to an unmapped virtual page is considered a disallowed memory access (a segmentation violation or bus error) which needs to be terminated. On the M-Machine, capabilities are used to control memory access. Since threads may not generate pointers on their own, they may not access arbitrary memory locations. All memory accesses which are issued by the memory functional unit on each node's cluster have had their capabilities verified. Therefore, a page fault is not considered a disallowed access on the machine, but rather an access to previously unmapped memory which is still a valid memory reference.

4.5 Implementation

The PMM is implemented as a low-level handler written in assembly which pops LTLB Miss events off the hardware event queue and passes them on to the handler body, which is written in C. Once a new event has come in, the assembly stub moves the four words of the event (referenced address, event header word, associated data, and configspace pointer to the faulting thread) into argument registers as defined by the M-Machine compiler and runtime system, and calls the body function. The handler body then determines whether the virtual address reference which caused the LTLB Miss is a fake virtual address and requires that special handling be employed, or whether it is a standard reference. Upon return, the stub restores its stack and returns to waiting for the next event to arrive.

The body of the handler is written in C, as shown in appendix D (`ltlb_body.c`). It calls on functions which manipulate the data structures outlined at the beginning of this chapter. The data structure code is also written in C and shown in appendix D. Initialization code is assumed to set up these structures when the LTLB handler is first spawned.

4.5.1 Event Format

The hardware-composed LTLB Miss Event consists of four words, shown in table 4.2. The address word identifies the referenced address which caused the LTLB miss. The header word encodes information such as the opcode which was used in the address reference, the issuing V-Thread slot, cluster, and source and destination registers. If the operation was a store, the opdata word contains the data which was attempted to be stored. Finally, the faultcp is the configspace pointer to be used by the software to write thread registers when fulfulling memory requests in software. If the faulting operation was a load op, the pointer offsets directly into the configspace-mapped location of the destination register of the load operation. If it was a store operation that faulted, the configspace pointer identifies a location which updates the faulting thread's *membar* counter. Conditional synchronizing operations have the faultCP identify the their destination cc register.

Word	Description
Header	Encodes information regarding the operation which caused the LTLB Miss and the issuing thread
Virtual Address	Virtual address which was not found in the LTLB
Opdata	Contains the 64-bit value which was attempted to be stored if the faulting operation was a store op.
faultCP	Configspace pointer to thread state for the faulting V-Thread.

Table 4.2: LTLB Miss Event Format

The low-level handler simply moves the message header and body words into argument registers and calls the manager's C-based handling function.

4.5.2 Initial Page Lookup

The handler body code extracts the virtual page number from the miss address that it is passed. This is a simple procedure of shifting off the 12 least significant address bits and masking off the 10 high protection/length bits to retain just the 42-bit page number. The virtual page number is then used to probe the page table by calculating the hash function and indexing into the table. A thorough study of good hash functions has not been performed. In the current implementation, the hash function is an XOR of a 16-bit constant and the rearranged bytes of the low 32 bits of a virtual address. C code is shown in figure 4-3.

```
result = (
          (((vpn >> 24) & 0xffL) << 16) |
          (((vpn >> 16) & 0xffL) << 24) |
          ((vpn >>  8) & 0xffL) |
          ((vpn & 0xffL) << 8)
         ) ^ 0x134aL;
```

Figure 4-3: Hash Function Calculation

Using the algorithm of open hashing, if the handler finds an entry marked *deleted*, or a valid entry whose vpn does not match the vpn being probed, the vpn is rehashed and probing continues. If a vpn match is found, the LTLB is accessed through configspace to determine which existing LTLB entry is to be evicted to make room. Since block-status bits for the virtual page whose mapping is to be evicted may have been modified, they have to be written back into the page table before the mapping can be evicted. Therefore, the existing entry is read from the LTLB through configspace load operations. The vpn of the evicted entry is used to probe into the page table and the block-status bits for that page are copied back into the hash table. Finally, the vpn-ppn mapping and associated block status bits for the page which is to be added to the LTLB are written into the LTLB through configspace stores, overwritting the evicted entry. The EMI is then unlocked through a configspace store, allowing the instruction which caused the miss to be retried automatically by the EMI, this time presumably hitting in the LTLB. Throughout this entire procedure the actual faulting operation is not retried by the handler, and the virtual address never used as the target of a memory operation - the LTLB Miss Handler operates only on physical addresses or configspace addresses.

If continual probing does not find a virtual page match, the page is determined to have no physical backing, and a new backing page needs to be allocated. A GTLB probe is performed to determine whether the virtual address which was referenced is mapped to the node handling the LTLB Miss (a locally-mapped page). If it was a local page reference, local handling is invoked. Otherwise, remote handling is performed. These two distinct cases are described below.

Local Handling

The free page chain pointer of normal (not memory-coherence) backing page frames is read to determine the next available frame which may be allocated. The first word of that page is copied into the free page chain pointer, effectively popping off the backing page. This page number is then added to the page table, creating a new virtual-physical mapping. Block-status bits are set to `exclusive` (read/write) for all lines in the page and also written into the page table. Finally, this entry is added to the LTLB so that the next memory access to this page does not cause another LTLB miss.

Remote Handling

An initial reference to a remote virtual page requires that a physical page from the pool of memory-coherence pages be used for the mapping. In the simple case, a physical page is available and is popped off the memory-coherence backing chain, in a manner similar to that described in the section above. The only difference is that the block-status bits for the page are set to `invalid` since the node does not yet contain any remote data. When the memory operation is retried, the memory reference no longer causes an LTLB Miss (since the mapping was written added to the LTLB and page table) but causes a Block-Status miss instead, which then results in the invocation of software leading to the local installation of a remote cache line.

If no physical page frames are available in the backing pool, a special physical frame number, `-1`, is used as a marker, identifying the fact that no backing frames are available. Since the block-status bits for the new mapping are still set to `invalid`, there are no problems involved in using the same mapping for all virtual pages which do not have available backing pages.[3] As section 7.3.2 details, this marker page is used as a trigger for the memory-coherence manager to perform a cleanup of existing shared pages and make room for new ones.

4.5.3 Fake Miss Interface

As explained in the beginning of this chapter, certain virtual pages which are always unmapped on the machine (`trigger pages`) are used to request direct manipulation of the PMM data structures by the LTLB Miss Handler. Since user threads are never given pointers to these special pages (and cannot create ones on their own), the miss handler is guaranteed that calls to it through misses are made by trusted subsystem code.

This mechanism, which involves threads generating memory faults to trigger actions by low-level components of the operating system, is similar to standard kernel-entry methods of other operating systems. As outlined in the previous chapter, performance is improved on the M-Machine because no actual thread swapping and context switching is performed.

The actual virtual page numbers used as triggers for the `PPM_map`, `PPM_unmap`, and `PPM_lookup` functions are compile-time constants in the kernel source code and may be picked rather arbitrarily, so long as they are not a subset of the virtual pages which may be allocated by the Virtual Segment Manager (see chapter 5). In the current runtime implementation, these virtual page numbers start at `0x80000`. To invoke the

[3]The only occasion, in fact, when multiple virtual pages may be mapped to the same physical page within a single node is when a physical backing page is unavailable, in which case all block status bits are set to `invalid`.

LTLB Miss Handler, a thread issues a conditional synchronizing store instruction, targetting one of the three trigger addresses as shown below:

```
                 /* cause a fault               */
instr memu stscnd <data-register>, <trigger-address-register>, <cc>;
                 /* block until fault completes */
instr memu ct <cc> ...
```

The store instruction allows the thread to pass 64 bits of data to the LTLB Miss Handler. In most cases, this contains the virtual page number which is to be used as an argument to the PMM functions. By issuing an instruction conditioned on the value of the cc register in the trigger instruction, the requesting thread blocks until the LTLB Miss Handler has completed the request and fills the cc register. Since the functions all have full 64-bit integer return values, the LTLB handler needs to have a simple way to return data to the requesting thread. One mechanism is to overwrite the integer register conventionally used as the return argument register by the compiler - integer register 6 - with the return value. This may be done through a configuration space store operation. In this case, the functional wrapper around the `PPM_unmap` primitive may look like:

```
PPM_unmap::
    /* i6 contains the argument to this function */
    instr memu stscnd i6, <trigger>, cc1;          -- trigger the LTLB Handler
    instr ialu ct cc1 jmp RETIP;                   -- wait for completion
    instr ;                                        -- i6 (the return register)
    instr ;                                        --   is already set properly
    instr ;                                        --   at this point
```

The current implementation instead writes a single physical memory location (called `_ltlb_data_for_mh`) which is then loaded by the caller to retrieve the appropriate value. In order to prevent concurrent accesses to this location, a lock is used by callers to serialize access.

4.5.4 Reclaiming Pages

The local and remote reclamation functions simply take the supplied page number and add that page to tail of the proper physical page chain. Cleaning needs to be performed before pages may be considered reclaimed and ready for reallocation.

4.5.5 Unmapping

When mappings need to be destroyed, the virtual page number argument to the `PPM_unmap` function is used to probe the page table until its entry is found. At that time, the entry is removed from the page table and replaced by a *deleted* marker, as necessary for an open-hashing table. In addition, cache lines may need to be flushed, and the LTLB modified to remove the virtual-physical mapping.

Note that no provisions are made for when virtual-physical mappings should be torn down. Higher level OS components make calls upon the PMM to create or eliminate mappings, but the PMM does not need to employ any policy for when mappings should be removed from the page table. Usually, this will be the work of the Virtual Segment Manager, which needs to deallocate physical backing once a virtual segment has been freed. See section 5.3.2 for more details.

Chapter 5

Virtual Memory Management

The Virtual Segment Manager (VSM) doles out segments of virtual address space for use by user threads and other portions of the operating system. Segments are a power of two bytes in length, with length protections enforced by the segment length field in pointers. A buddy list allocator is used for the implementation of the underlying allocation mechanism. Since a copy of the low-level operating system runs on each node of the M-Machine, each node's VSM executes independently of all others. This section describes the interface to the VSM, explains the data structures which are employed, and then details the rather simple implementation. Design issues conclude this chapter.

5.1 System Calls

The VSM exports a total of three functions. The first, `vmem_prime`, is accessible only to other protected subsystems and allows the bootstrap to initialize the allocator with segments of virtual memory which are available for allocation. The `vmem_alloc` call returns a segment of a requested size, while the `vmem_dealloc` call accepts a segment for deallocation. Table 5.1 provides a brief overview.

Function	Description
`void vmem_prime(void *segment_ptr)`	Identifies the virtual segment pointed to by *segment_ptr* as available for allocation. The pointer length field in the pointer's capabilities explicitly identifies the size of the segment.
`void *vmem_alloc(int bytecount)`	Returns a pointer to a clean segment of virtual memory of at least *bytecount* bytes in size. Returns a NULL pointer if no such segment may be allocated.
`void vmem_dealloc(void *segment_ptr)`	Deallocates the segment of virtual memory identified by *segment_ptr*.

Table 5.1: Virtual Segment Manager exported functions

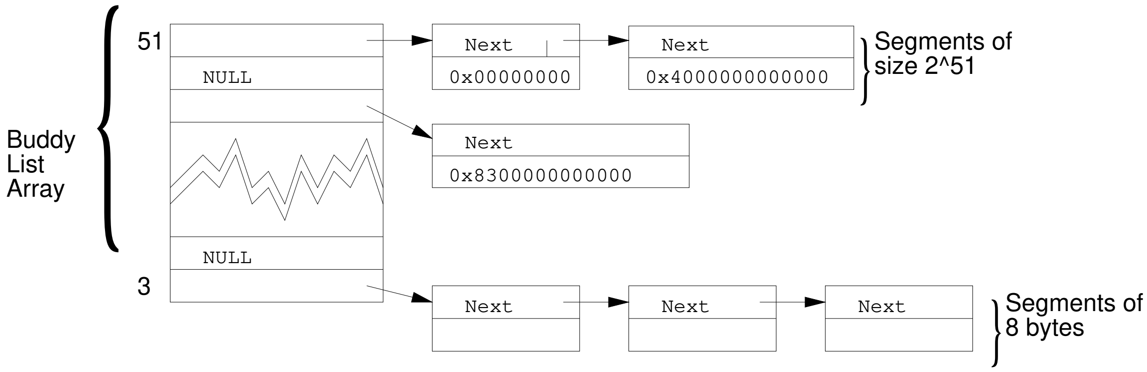

Figure 5-1: Buddy List Structure

5.2 Data Structures

The VSM maintains two buddy lists for memory allocation and deallocation. A buddy list is essentially an array of sorted linked-lists of segments (see figure 5-1). The array has as many entries as there are segment sizes available on the machine. On the M-Machine, an array of 51 entries allows segments to range from 8 bytes to 2^{54} bytes in length.

The free-segment buddy list stores information on segments which are available for allocation. Initially empty, the list is first primed with segments by the `vmem_prime` call. Subsequent `vmem_alloc` calls remove entries from this free segment list, perhaps modifying it in the process, and return newly-available segments.

The dirty-segment buddy list records segments which user programs or protected subsystems have asked to be deallocated - those that have been passed to the `vmem_dealloc` call. This allows deallocated segments to be collected and coallesed into larger segments for bookkeeping pending garbage-collection. A deallocated segment cannot be moved directly from the dirty to the clean list unless a garbage-collection phase has ensured that there are no clones of this virtual address remaining on the entire machine. Therefore, the dirty buddy list essentially stores segments which are candidates for a garbage-collection phase. The current implemention of MARS does not perform garbage-collection. Therefore, the dirty list is just a repository for segments which will never be given out. In fact, it is possible to make the `vmem_dealloc` function a null function.

Finally, a statically-defined linked-list of nodes for use in both buddy lists allows the allocator to function without needing dynamic memory-allocation itself, although this limits the number of memory segments which may reside in the buddy lists to a compile-time constant.

5.3 Implementation

The MARS bootstrap splits the M-Machine's global virtual memory into segments and assigns them to different nodes. Each node's boot code calls `vmem_prime` with its assigned segment, priming the free-segment buddy list data structure and allowing subsequent allocation calls to use that memory.

Once the priming is complete, the VSM accepts calls from user-level as well as system-level code. Since the VSM code may be running within many thread slots at once, a lock is used to serialize access to global data structures.

5.3.1 Allocation

When an allocation call is made, the requested number of bytes is used to calculate the smallest power-of-two-byte segment which contains at least that many bytes. As [7] and [6] explain, this may lead to wasting both virtual and physical address space since a little less than half of the entire segment may go to waste (e.g. a call to allocate a segment of 129 bytes will return a segment of at least 256 bytes in size). The waste of virtual address space is not a great problem, given the large size of the machine's address space. Physical address waste is limited a maximum of a single page, due to the policy of on-demand page allocation. That is, since only those virtual addresses which are targetted by a memory operation require physical backing, having a large number of allocated but unused virtual pages at the end of a segment does not cause wasted physical page frames to be allocated.

A search of available segments in the clean buddy list then begins for a segment of the appropriate size. This is a simple procedure given that the requested segment size is known - the free-segment array is indexed to find if any segments of the needed size exist. If the array entry is non-NULL, the linked list of segments of the requested size is modified as a segment is popped off the list. The pointer to the newly-allocated segment is returned to the caller. If no segments of the needed size exist, the allocator begins looking for larger segments, simply by moving up in the free-segment array, looking for linked-lists of larger and larger segments. Any larger segment that is found can be repeatedly split into two until the correct size segment is once again available. Leftover segments are added to the buddy list in the process, for later allocation.

The on-demand allocation of page frames simplifies VSM implementation since no mappings from virtual pages within the allocated segment to frames need occur - the LTLB Miss Handler will perform those tasks as each virtual page is touched.

5.3.2 Deallocation

As mentioned previously, virtual segments which are deallocated are added to the dirty-segment buddy list and need to pass a garbage-collection phase before being added to the clean list. However, an initial unmapping phase must occur to remove any virtual-physical mappings used by the segment, in order to free up physical memory. The need to deallocate physical backing from a segment actually poses a problem because the only data structure which lists all allocated physical page frames is the on-node page table. It may be inefficient for the VSM to run through all of the virtual pages within a segment and make `PPM_unmap` and `PPM_reclaim_local` calls for each, especially if the segment is large and the number of actual pages provided as backing for it is unknown. For small segments, the VSM cannot deallocate the mapping because other segments within the same virtual page (and hence mapped to a common physical page frame) may be still active. The VSM splits the deallocation problem into three cases.

The dirty-segment buddy list is used in the reverse manner of allocation - segments which are deallocated are coalesced with their buddies to try to form a single segment of as large a size as possible. In the case of deallocating segments smaller than a single virtual page, no unmapping is performed by the VSM initially. Instead, as the segment is coalesced with other dirty segments, the VSM waits (perhaps for many deallocations to follow) until a segment the size of a virtual page is finally formed. This means that many small segments which had resided within the same virtual page have all been finally deallocated. The VSM can make a single pair of calls (`PPM_reclaim_local(PPM_unmap(vpn))` - that is, return the page previously used to back the deallocated virtual page to the local page pool) at this point, passing to the Physical Memory Manager the virtual page number of the segment.

For segments of moderate size (smaller than the number of physical pages on a node) which are deallocated, the above design will not work since the segment already spans many pages. Instead, unmap and reclaim calls are made for each page within the segment. Finally, for very large segments, the PMM must be called to unmap the entire segment, which requires that the PMM search for all entries in the page table which match a *range* of virtual pages (not just a single one) and remove all such mappings. This is especially efficient for very large segments, since the number of pages which need to be tested is limited by the amount of on-node physical memory.

5.4 Design Issues

For a machine with a large virtual address space, such as the M-Machine, buddy list allocators are quite efficient because they can quickly manage segments of memory which vary greatly in size. The fact that segment-size is encoded directly into all M-Machine pointers make this scheme even more efficient - a call to deallocate a segment uses the segment-size field to determine which low address bits of the pointer to ignore, and which high bits to use when searching for a segment's buddy.

The unmapping of backing page frames for segments seems the most inefficient aspect of the VSM design, and can be improved if a bitmap of which virtual segments actually have physical backing is maintained for each segment which is allocated. This increases overhead, however, and requires more storage space for such bitmaps. It is not clear, for example, how to maintain a mapping-bitmap for a segment of 2^{24} bytes. Such a segment spans 4096 page frames, requiring 64 words for a bitmap. The advantages of the current design lie in the fact that once a segment has been allocated and returned to a user thread, all information pertaining to its existence is no longer maintained by the VSM, until a call is made to deallocate it. At that point, the segment itself is provided to the VSM, which may add it to the dirty-segment list and start keeping track again. In fact, the reason for maintaining the dirty segment list (as opposed to a more simple linked-list of deallocated segments) is not for performance improvements, but rather for storage efficiency - fewer individual segment-information nodes need to be maintained if dirty segments are naturally coalesced. If two buddies are combined to form a larger segment in the dirty segment list, this frees up one more segment-information node which may be reused by the system.

Chapter 6

Thread Management

In traditional operating systems, a *process* represents a basic vehicle for executing code. Processes may be composed of threads which cooperate and share an address space and any special structures assigned to their collective process by the operating system. In MARS, there is no real concept of a heavyweight process. Since all privileges are granted through pointers given out by the system, all threads are protected from each other, yet any subset may cooperate on a task as well.

The MARS thread mananger is responsible for allocating and destroying user-level threads, scheduling threads to run in the available user thread slots, and managing interthread synchronization through the `tSignal` and `tSleep` interfaces. Threads which synchronize through explicit message-passing or shared-memory have no need for the thread manager to aid in their communication. The sleep and signal interface allows multiple threads to sleep on a single signal and be all awakened when it arrives, and even for a single thread to provide a signal mask, so that it is possible to group signals into categories and allow threads to pick which types of signals they wish to receive.

Instead of using some integer to identify each process (thread) which has been created by the thread manager, a *context pointer* is used instead. A context pointer is a pointer of type `key` whose address portion names a virtual memory segment which contains state information about the thread (the thread context.) Since this pointer cannot be used to read or write memory, it may be returned to user-level threads as a magic cookie, identifying a particular thread. When an operation needs to be performed on the underlying thread state, a privileged system function may simply modify protections on the pointer from *key* to *read-write*, without the need to index into a process table. As will become evident, context pointers are used extensively within the thread management system to identify and track threads.

6.1 System Calls

This section describes the system calls available to user threads for accessing thread manager functionality. All of these system calls may safely execute as priviledged code in user thread slots since they do not modify any hardware state. Table 6.1 lists the common thread manager calls.

This set of system calls provides a great deal of functionality to user threads with a very simple interface. An example of how these calls are used is given in figure 6-1. In this example, the main parent thread spawns off a child to execute the function `foo` and then sleeps on a T_CHILD_EXIT signal, waiting for the child to complete. There is no explicit `tSignal`, because the signal is performed by the `tExit` function which the parent passed to the child. The `_getDP` function returns the parent thread's data pointer so that the child may share all of the parent's data structures.

More complex examples of system call use will be given later in this chapter.

In addition to the above system calls, several internal functions of the thread manager are invoked by the Event Handler and Message handlers. These include actual scheduling, and low-level signal reception.

6.2 Data Structures

The thread manager uses a structure called a thread context to store information about each live thread on its node. A signal table is used to manage the signal/sleep interface. Finally, pointers to chains of thread contexts maintain information on active threads. These structures are described in this section.

Function	Description
`void *tFork(void *IP, void *DP, void *retIP, int numargs, void *parent, ...)`	Creates a new thread which will begin execution at address `IP`. The thread's data pointer is set to `DP`. When the thread exits, it will jump to `retIP`. The number of arguments passed to the function at `IP` is given in *numargs*, followed by the arguments themselves. *parent* is usually left NULL. This function returns a key pointer identifying the newly-created thread.
`void tExit(int retval)`	Standard exit procedure usually passed as the retIP to `tFork`. Signals its parent thread with a `T_CHILD_EXIT` signal and return-value *retval*.
`void *tSpawn(int numargs, void *IP, int node, ...)`	Forks a thread on remote node given by *node*. The data pointer is the same as the thread which called this function. The forked thread will start executing at `IP` and signal its parent when done. The number of arguments being passed to the function at `IP` and the argument list itself is also given. Returns a key pointer identifying the spawned thread.
`int tSleep(void *sigword, int mask)`	Puts the calling thread to sleep until a signal arrives which targets the *sigword*. The *mask* allows the calling thread to only be wakened by a subset of all signal arriving for the signal word. A mask of `0` will always match a signal. Returns the data which was send to the signal word (see `tSignal`.) The signal word must be a key pointer.
`int tSignal(void *sigword, int data)`	Attempts to wake all threads sleeping on *sigword*. The *data* is the data returned to all matching sleepers. If no sleepers are found, a dormant signal is recorded. The signal word must be a key pointer.

Table 6.1: Thread Manager system calls

6.2.1 Thread Contexts

The thread manager defines a thread context data structure which is used to store information about each live thread. Several linked-lists of thread contexts group these threads into collections of running, pending, and kill threads. Running threads are the user-level threads actually occupying V-Thread slots on the manager's node. Pending threads are waiting to be scheduled to run on the hardware. Blocked threads are sleeping on a signal and should not be swapped into a thread slot until wakened

```
int foo(int i, int j) {
    int x = 0;

    printf("This is function foo!\n");

    printf("let's calculate i + j : %d\n", i + j);
    printf("foo exiting");
    return i + j;
}

int main(int argc, char **argv) {
    char *mydp;
    void *child;
    int i;

    mydp = _getDP();      /* _getDP returns the thread's own data pointer */
    child = tFork(foo, mydp, tExit, NULL, 2, 1, 10);
    printf("main: forked foo (child pointer is %p)\n", child);

    i=  tSleep(child, T_CHILD_EXIT);
    printf("main: woken with signal 0x%x\n", i);
    return 0;
}
```

Figure 6-1: Sample Thread Management system call usage

(they are stored implicitly in a signal table described later). Kill threads are waiting to be garbage-collected and removed from service. Together with running threads, kill threads may occupy hardware thread slots, but should be evicted by the thread scheduler.

Figure 6-2 shows a C structural definition of a thread context. The main sections of the context structure are the individual H-Thread contexts, (which define the entire register state of the H-Threads that compose the user thread), global thread state information, and linkages to other contexts.

The HContext structure simply contains space for all of the integer, floating-point, and condition registers of a particular H-Thread, the four restart instruction-pointers (used when installing a thread for execution), hardware and software memory-barrier counters (count how many memory references the thread still has outstanding in the system), and a scoreboard of which registers are vacant.

```
struct ThreadContext {
    struct ThreadContext *Next;
    struct ThreadContext *Parent;
    struct ThreadContext *Sibling;
    struct ThreadContext *Children;

    struct HContext hthreads[4];  /* register state for each H-Thread */

    int VSlot;
    int flags;                    /* hFull and hIssue bits IIIIFFFF  */
    int SCC  ;                    /* stall-cycle counter             */
    int SCL  ;                    /* stall-cycle limit               */

    int signalData;               /* data passed when thread woken   */
    int need_to_block;            /* thread is blocked for a signal  */
    int need_to_wake;             /* signal has arrived              */
    int need_to_sleep;            /* thread has asked to sleep       */
};
```

Figure 6-2: Thread Context data structure

Global state information records which H-Threads of the user thread are active and may issue. When a thread is first forked, only the first H-Thread is active. If the thread spawns other H-Threads to neighboring clusters, this value will change. Thread flags are composed of eight bits in two 4-bit bitmaps - called hFull and hIssue. The hFull bitmap records which H-Threads are part of the V-Thread represented by the thread context. The hIssue bitmap is used as a mask to tell hardware which H-Threads may issue operations down their cluster pipelines. Special state information used in the signal/sleep implementation is also part of global thread state. The `signalData` field records the data word with which a thread was wakened. The three state bits of need_to_block, need_to_wake, and need_to_sleep are used by the scheduler to help decide which of the pending/running lists is to receive this thread. These state bits will be discussed in detail in the section on signalling. Finally, the thread Stall Cycle Limit (SCL) and Stall Cycle Counter (SCC) are used by the the M-Machine hardware to generate events if a particular user-level thread has been stalled and unable to issue for a certain number of cycles.

The linkages (`Next`, `Parent`, `Sibling`, and `Children`) allow thread contexts to

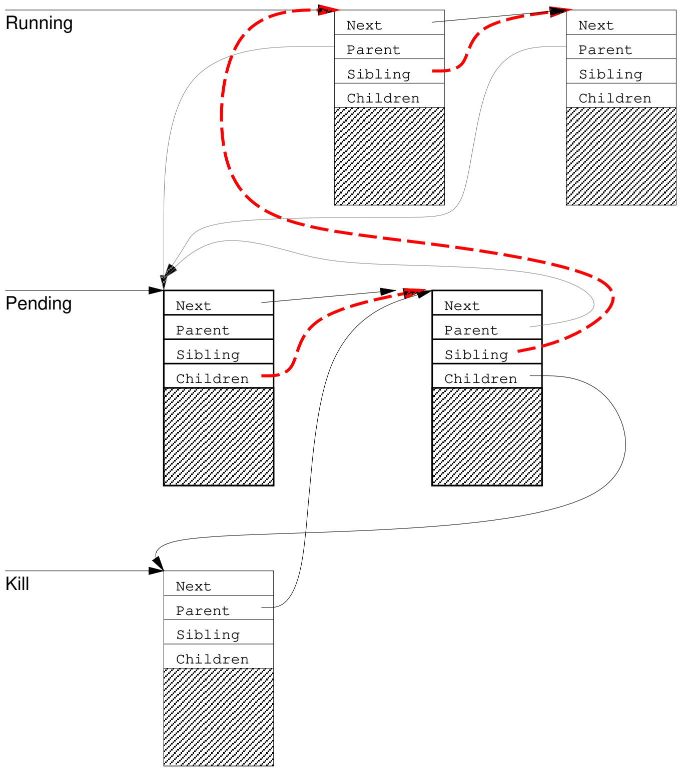

Figure 6-3: Context Linkages

be threaded onto several linked-lists at once. The main pointer is `Next`, which is used in the running, pending, and kill lists mentioned above and described in detail in a later section. The Parent pointer points to the thread's parent. Usually, the parent is the thread which tFork'ed the thread, although a different parent may be substituted (this is the `parent` argument to the tFork call). The Sibling pointer is a secondary linked list, which winds itself through all of the children of a particular parent thread. That is, even if the children of a particular parent are strewn around different pending/running/kill lists, this single list can identify all of the children of the parent regardless of where they are. This makes it easy to find and kill all children of a particular parent thread, without needing to look through all lists of threads (looking for contexts with a particular parent). Finally, the children pointer is the head of the Sibling list, which resides with the parent. Figure 6-3 makes this structural arrangement more explicit.

In this example, the first thread on the pending list is the parent of three threads - one also on the pending list, and two others that are running. One of its children is the parent of a thread which is on the kill list.

Thread contexts reside in virtual address space, and are dynamically allocated by the `tFork` call. Since all virtual addresses are unique across the entire machine, a thread context unambiguously identifies a thread to all operating system components across all nodes of the machine. All threads may access their own context pointer through a call to `_getMyTC`, and the context pointer of their parent with `_getParent`. The pointers that are returned are key-type pointers, to prevent user threads from actually modifying thread state.

6.2.2 Signal Table

In order to maintain information on which threads have performed signals and which threads have tried to sleep, the thread manager uses a chained hash table of signal entries.

A signal entry records information about a thread which has asked to be put to sleep, or a signal which has been made before any thread has slept on it (see figure 6-4.)

```
typedef struct se {
    struct se *next;
    int signal_word;
    int signal_data;
    struct ThreadContext *sleeper;
} signal_entry;
```

Figure 6-4: Signal Entry

If a thread has slept on a signal word, the two arguments to the sleep call (signal_word and mask) are recorded along with the thread context of the thread making the sleep call (*sleeper* in the signal entry. If the entry is recording a signal for which no thread has slept yet, sleeper is NULL and the signal_data is the actual data passed to the tSignal call.

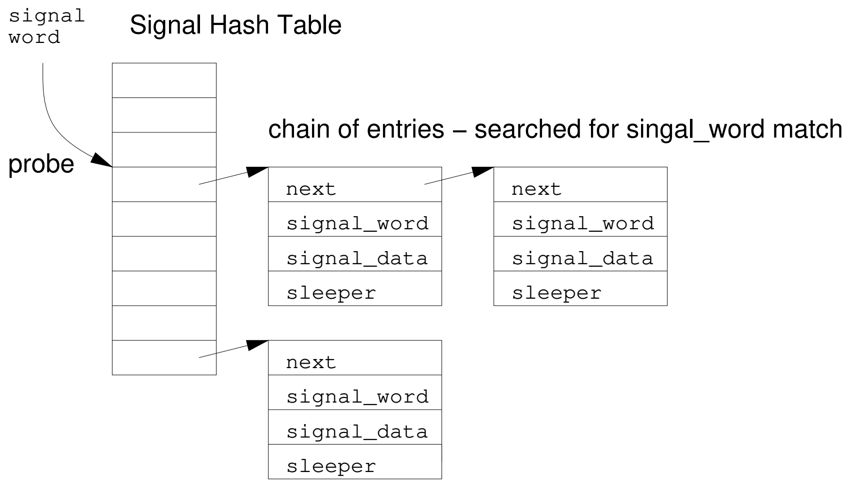

Figure 6-5: Signal Hash Table Structure

Signal entries are split into chains and referenced from the signal hash table, to improve lookup speed. The unique signal_word is hashed and identifies the chain, which may then be searched for matching entries. Signal entries may be dynamically allocated in a manner similar to thread contexts, or a fixed number may be statically allocated at compile-time into the runtime system (similar to what is done by the virtual segment manager.)

6.2.3 Thread Lists

The low-level scheduler employs thread lists, headed by pointers to *Pending*, *Running*, and *Kill* lists. All threads active on a node belong to one of these lists, or have *sleeper* entries in some signal table (effectively the collection of blocked threads). This guarantees that thread manager components have a way to find all active threads on the node by following these structures. The *Next* pointer in a context lets it be threaded in one of these lists. A thread may be in only one of these lists at a time.

6.3 Implementation

When the thread manager is initialized, it sets up a blank signal table and resets the running, pending, and kill thread lists to contain a single running thread - the bootstrap. Calls to the manager's system calls will begin modifying these structures. It was briefly noted that the thread manager is really composed of system calls executing in user thread slots and a low-level scheduler tied into the event-handler system. For this reason, the thread management implementation uses a producer-consumer model for servicing requests. User-accessible system calls invoke functions which set up and sometimes modify thread state. After certain global data structures are modified, the event handler is signalled through its software job queue to perform the low-level scheduling tasks. This two-phase design simplifies the implementation of individual thread manager components. It also allows thread manager subsystems to execute in conjunction with the scheduler without relying on locks to serialize access to common data structures - all data structures which are modified by the portion of the thread manager which runs in the event handler slot do not interfere with other thread manager functions.

In general, producers create or modify thread contexts which are then added to the running, pending, and kill lists by the scheduler (this list modification is performed when the event handler responds to certain signals). The scheduler examines these lists each time it is invoked and performs lowl-level functions such as thread eviction and installation. The following sections describe the producer's contribution to handling system calls. It is important to note here that in all critical sections of the portion of the Thread Manager that runs in user thread slots, a lock called the `userthreadLock` is used to serialize access to global data structures among user threads. This lock is not accessed by the low-level scheduler, and hence does not cause it to block in any of its activities.

6.3.1 tFork

The tFork function needs to allocate a new thread context by calling on the virtual segment manager, and fill an initial H-Thread with information passed to it. It allocates a new thread stack, again calling on the VSM, and pushes arguments on the stack exactly as the called thread expects to see them. The return pointer is set up as well, so that the exit function passed to the tFork is the last function executed. Parent/child/sibling linkages are updated to reflect the fact that a new thread has been created and that it belongs to some parent. If the parent pointer passed to the fork call is NULL, the thread executing the fork call is considered the parent (this is the common case). Remote parents are a special case, which are handled within the tSpawn implementation. Finally, the event handler is signalled to add the new thread context to the pending list. This signifies that the thread is ready to execute and is waiting to be scheduled into an available thread slot.

6.3.2 tExit

The tExit call must mark its own thread for termination since it is executed within the very user thread which is trying to exit. First, the thread calls tSignal on its own context pointer with a return value of T_CHILD_EXIT. Any thread waiting for this particular child to exit (most likely its parent) will be wakened.

The sibling list is modified to reflect the termination of this thread. The event handler is then signalled to add the thread to the kill list. The event handler removes the thread from the running or pending lists and adds it to the kill list. Finally, tExit blocks on an empty register to prevent stealing any more execution cycles. Eventually, the scheduler will be invoked and terminate the thread which had been added to the kill list.

6.3.3 tSignal

The tSignal system call is used by a thread to signal another thread, passing it a 64-bit data word. Signals are made upon *signal words*, which are key pointers given out by the operating system. The most common signal words are the thread context pointers exchanged by the parent and child during a tFork call. Other signal words may be obtained simply by calling on the operating system to demote the protections of a virtual-memory read-only or read-write pointer to key.

The tSignal call takes a signal word and a 64-bit data word as arguments and determines which signal table to examine. If the address defined by the signal word is mapped to the thread manager's own node, the local signal table is examined. Otherwise, a message is sent to the node where the signal word is mapped, and a TM local to that signal table is invoked. The TM determines whether an address is remote or local by making a call to `_sysGPRB`, a function which performs a GTLB probe and returns the node number to which an address is mapped. This allows threads on different nodes to signal each other and for all thread managers to quickly decide which signal table needs to be referenced.

Once a local TM is invoked to examine the signal table, the signal word is used as the input to a hash function and an index into the signal hash table is calculated. This index identifies a chain of signal table entries which is to be searched to find a match or matches (for multiple sleepers) on the signal word. In order for a signal to match an entry, it must meet three criteria.

1. the *signal_word* field of the entry must match the signal word passed to tSignal
2. the signal_data [mask] field in the entry bitwise ANDed with the signal_data passed to tSignal must be nonzero (unless the mask is `0`, in which case this criterion is always considered satisfied)
3. the *sleeper* field of the entry must be non-NULL.

For each match that is made, the thread identified by the sleeper context pointer is wakened (this process is described below.) Once all sleepers have been wakened, the signal operation has completed. If no sleepers were found, a *dormant* signal entry is added. This means that the signal is added to the signal hash table and waits for a sleeper to come along, at which point the thread which attempted to sleep on the signal is automatically wakened. Such dormant signals are added to the ends of the signal chains, to handle cases where multiple dormant signals for the same signal word are added. In these cases, the signals are meant to be popped off in a FIFO manner, until they are all used up.

For each thread context which needs to be wakened, the tSignal system call must decide whether the wakening occurs locally or remotely. Once again the TM probes the GTLB, this time to determine whether or not the sleeper thread context is mapped to the local node. If the thread context is remote, a *Wake* message is sent to the appropriate home node of the thread. Otherwise, the event handler is signalled to set a thread's wake data. This causes the thread context's *signalData* field to be written with the signal data passed to tSignal, and the *need_to_wake* field set to true, signifying that if the thread happens to be blocked, the scheduler should move it to the pending list.

6.3.4 tSleep

A user thread calls tSleep when it wishes to block, stopping execution until a signal wakes it. This is especially useful when a thread has spawned off some children which are to perform long-latency operations and wishes to be informed when these operations have completed. Although it is possible for the parent thread to spin on global memory locations waiting for child thread to modify them, this is extremely inefficient if the child processes are expected to take a long time to complete their operations, and the parent has no other work to perform.

For this reason, the calling thread identifies itself as sleeping on a particular signal word, and also passes a mask as data. This mask is used to filter out certain signals to the signal word which the sleeping thread does not wish to see (as described above). As in the case of tSignal, the signal word is used to probe the GTLB to find the home node of the signal table. If the signal table is remote, the thread asks to be put to sleep locally and sends a message to be added to the remote signal table. It is important to note here that it is possible for the message to arrive and a dormant signal to be found which would cause a wake message to be returned, all before the local TM is able to put this thread to sleep. The need_to_sleep, need_to_wake, and need_to_block

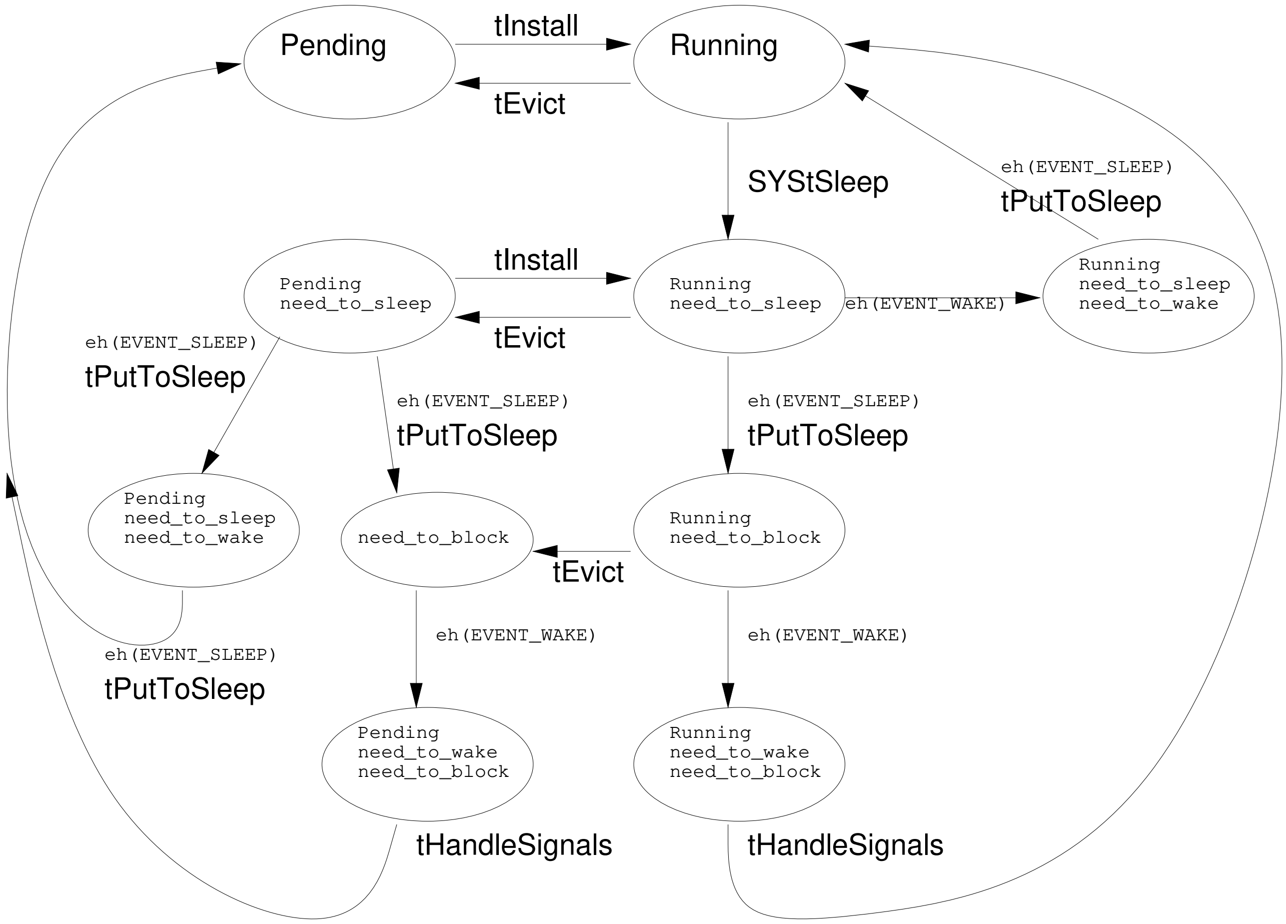

Figure 6-6: State Transitions in Signal/Sleep Implementation

bits define state-transitions to handle such cases. Figure 6-6 shows a state-transition diagram where a thread state is a function if its *need_to_xxx* bits and whether it is running, pending, or neither. Transitions occur as a result of the low-level scheduler performing routine scheduling tasks, or being invoked as a result of signals to the event handler (`EVENT_SLEEP` and `EVENT_WAKE`). Certain functions are automatically invoked as a result of these signals (such as `tPutToSleep` and `tHandleSignals`).

Finally, whether as a result of a Sleep message from a remote TM or the fall-through case of a local tSleep call, the TM needs to add a sleeper for the signal word to the signal table. Again as in the tSignal case, the signal word is used as the hash input to find a chain of signals. The chain is examined for any *dormant* signals to this word. If a dormant signal is found and the data within it filters through the mask provided by the calling thread, the thread is immediately wakened. If the thread was local to the signal table, the data is returned directly to the thread without the thread having ever been put to sleep. Otherwise, a wake message is sent to the home node of the sleeper thread.

If no dormant signal entries are found, a new sleeper entry is made. Finally, if the TM is still executing locally, it makes a call to `sysSignalSleep`, which asks the scheduler to move the thread off the running list (if possible) and consider it blocked until a signal arrives. At the same time, this action causes the thread to empty the return-value register and block on it. Whenever this register is written (as a result of the scheduler restarting a thread which is being wakened by a signal) the thread will resume execution and return a value to the caller of tSleep.

Figures 6-7 and 6-8 show an example of the use of signal and sleep calls for interthread synchronization. The parent thread forks a child called longprint, which in turn forks off longprint_child. Longprint then waits for its child to signal it. Meanwhile, the main parent sleeps on a signal from longprint. longprint_child signals its parent and then goes to sleep, waiting for longprint to signal it. At this time, both main and longprint are sleeping on the same signal word. When longprint is wakened by its child's signal, it signals to its own threadcontext pointer, waking both its child and its parent. Finally, longprint waits for its child to exit before exiting itself. The main thread waits for longprint to exit.

6.3.5 tSpawn

The tSpawn system call is a good example of how lower-level thread manager primitives may be composed to form a more useful function. A tspawn is essentially a request by the user to fork a thread on a remote node and still have the child's thread context be returned to the parent. The tSpawn implementation first creates a nonce which will be used for a signal/sleep pair. In the current implementation, this nonce

```
#include <stdio.h>
#include "syscalls.h"
#include "tsignal.h"

int main(int argc, char **argv) {
    void *child;
    int i;

    printf("Sample signal/sleep program\n");
    child = tFork(longprint, _getDP(), tExit, 6, NULL, 1, 2, 3, 4, 5, 6);
    printf("main: forked off %p\n", child);
    i = tSleep(child, T_ALL_SIGNALS);
    printf("main: woken with 0x%x\n", i);
    /* wait for a while */
    for (i = 0; i < 900; i++) ;
    i  = tSleep(child, 0x100);
    printf("main: woken with 0x%x from child %p exit\n", i, child);
    return 0;
}
```

Figure 6-7: Sample signal and sleep system call usage: main thread

is simply a newly-allocated segment of virtual memory used and then discarded.[1] A message is then generated, and the nonce and arguments to the spawn are sent to the destination node. Finally, the calling thread performs a tSleep on the nonce, waiting to be notified when the new thread has been created. It expects the return value of the tSleep (the data when it is signalled with tSignal) will be the thread context of the new child.

On the receiving node, a message-handler dispatch function processes the tSpawn request. The Spawn message is unpacked and arguments formatted for a tFork call. This time, instead of a parentTC of NULL being passed, the TC of the remote parent is substituted (this was passed in the message, along with argument list, IP, and so on), allowing linkages to be set up correctly. After the tFork completes and returns a thread context, the message-handler performs a tSignal on the nonce passed within the spawn request message, passing the child thread context as data. This eventually

[1]Since the VSM returns pointers as Read/Write, a *demote* call is made to change the protections to key pointer.

```
int longprint_child(int i, int j) {
    int sleepval;

    printf("longprint3_child: i * j = %d\n", i * j);
    tSignal(_getSelfTC(), 0x112);

    /* now wait until longprint signals me */
    printf("longprint_child: going to wait for longprint to signal me\n");

    sleepval = tSleep(_getParent(_getSelfTC()), T_ALL_SIGNALS);

    printf("longprint_child: woken with 0x%x and exiting\n, sleepval);
    return 4;
}

int longprint(int i, int j, int k, int l, int m, int n) {
    int x = 0;
    void *child;
    int sleepval;

    printf("longprint: %d, %d, %d, %d, %d, %d\n", i, j, k, l, m, n);

    child = tFork(longprint_child, _getDP(), tExit, 2, NULL, 5, 11);
    if (child) {
printf("longprint: forked off %p, and sleeping on it\n", child);
sleepval = tSleep(child, T_ALL_SIGNALS);
printf("longprint: woken with 0x%x from child %p\n", sleepval, child);

        for (x = 0; x < 200; x++)
  if (!(x % 20))
    printf("longprint: %d\n", x);

tSignal(_getSelfTC(), 0x223);

/* sleep on child exiting */
sleepval = tSleep(child, T_CHILD_EXIT);
printf("longprint: child %p exited\n", child, sleepval);
    }

    printf("longprint exiting");
    return 1;
}
```

Figure 6-8: Sample signal and sleep system call usage: child threads wakens the calling parent who receives the child thread context just like the return value of a tFork.

6.3.6 Scheduler

The scheduler portion of the Thread Manager runs as part of the event handler - responding to requests placed in the software job queues. Requests are summarized in table 6.2. The generic EVENT_SCHEDULE is the most interesting to cover because it encompasses the important tasks of installing and evicting threads.

Request	Arguments	Description
`EVENT_SCHEDULE`		Perform generic scheduling: wakes threads which have `need_to_wake` set. Terminates threads on the kill list. Attempts to install threads on the pending list, perhaps evicting running threads to make room.
`EVENT_SLEEP`	`tc`	Puts thread identified by thread context pointer `tc` into a blocked state. If the thread is already running, it is moved to the front of the running queue so it is the first to be swapped out if an eviction is necessary. If thread is on the pending list, it is removed from the list so as not to be mistakenly installed during scheduling. Sets thread's `need_to_block` state bit.
`EVENT_FORK`	`tc`	Adds thread identified by thread context pointer `tc` to the Pending list.
`EVENT_WAKE`	`tc data`	Sets the `need_to_wake` state bit of the thread identified by `tc`. Sets the thread's `signalData` field to `data`. If the thread is not currenly occupying a thread slot (running) it is added to the pending list.
`EVENT_KILL`	`tc`	Adds the thread identified by `tc` to the kill list.

Table 6.2: Thread Manager system calls

The scheduler completes three tasks when asked to perform scheduling.

Cleaning Killed Threads

First, all threads in the kill list are popped and terminated, if possible. Their thread context is freed, the hardware thread slot state that they occupy (if they are still installed in a thread slot) is reset and the thread slot marked as unoccupied. If threads which are popped off the kill list still have outstanding memory events which are to be resolved in software or outstanding hardware events, the threads may not be terminated and are added back to the kill list. A check in the code which runs through the kill list makes sure that recirculating threads into the kill list does not cause an infinite loop of pushes and pops.

Signal Handling

The thread scheduler then deals with outstanding signal-handling. A thread which is (1) in the pending or running lists, (2) has its `need_to_wake` state bit set, and (3) has its `need_to_sleep` bit unset, is set active by copying signalData into the appropriate return register. If it is occupying a thread slot, the thread's return-register (i10) is written with the contents of the context's `signalData` field directly (using a configuration-space write). Otherwise, the register is modified within the thread context and the empty bit for that register set to full so that the register can be read the next time that the thread is installed into a thread slot. In both cases, the `need_to_wake` bit is reset.

Installing Threads

In its third task, the scheduler pops a thread off the pending list (the candidate) and attempts to install it into a free user thread slot. If no free thread slots exist, a thread is popped off the running list and evicted (if possible). Eviction involves halting all H-Threads which are issuing within the V-Thread - accomplished by writing to the thread flags region of configuration space mapped to the hardware thread slot which the thread occupies. The thread flags are modified to zero out the hIssue bits for the thread. Then, for each active H-Thread within the V-Thread, all of the register-file state is copied into the thread context. Four H-Thread IP's for use in the thread-restart process are read out from each cluster. Finally, state like software and hardware membar counters are updated. Once eviction succeeds, the thread context is pushed to the end of the pending list.

When a free thread slot has been found for the candidate, a reverse of the eviction process begins. First, the candidate's hFull thread flags are written into the configuration space mapped to the thread slot into which it is being installed. These flags set the hFull bits for all H-Threads which are to run within the candidate. This has the effect of resetting all thread state within individual clusters. This is a safe procedure since no hIssue bits are set, so the thread will not attempt to issue from a non-existing IP. Then, individual H-Thread state is updated by reading thread context data and writing into the thread slot through configspace. After all register-file and membar counter state has been written, a series of 4 IP writes are made for each H-Thread. These writes prime a hardware restart engine which fetches instructions and can restart a thread. Lastly, the candidate is pushed to the end of the running list.

Chapter 7

Memory-Coherence Management

This chapter details the M-Machine's software-based memory-coherence protocol. As mentioned in previous chapters, the software implementation is closely tied to other OS components, such as the Physical Memory Manager and Thread Manager. The memory-coherence system provides the view of a single globally-shared virtual address space which is accessible by user threads independent of the node on which they execute. That is, any thread which performs a memory-reference to a word of virtual memory will have that request satisfied even if the segment of virtual memory is not mapped to the thread's *home node*. Each word of virtual memory is mapped, through the GTLB and a software Global Page Table (not implemented in the current runtime system), to an M-Machine node - the home node of that data. For purposes of the memory-coherence protocol described in this paper, the granularity is on an 8-word block basis (words in each 8-word block of memory must have the same home node in common). The term "memory block" (or just "block") refers to an 8-word section of virtual memory, the size of an individual cache-line, which may be shared among several nodes. In the rest of this chapter, the *home node* means the node to which a particular block of memory is mapped, and a *requesting node* is used to identify a node which wishes to access data from the home node. In rare instances, the home and reqesting nodes may be the same.

In broad terms, the memory-coherence manager allows threads to transparently read and modify blocks of memory which are not mapped to their local nodes. Load and store operations which attempt to access off-node data fault to software with *block-status* misses (BSM). A portion of the memory-coherence manager (MCM) which runs in the event-handler thread enqueues BSMs into a software event table, and sends out request messages for accessed blocks. Message-handing functions in the P0 and P1 Message Handler threads respond to request messages by modifying local coherence directories, local cache, and the LTLB, and send blocks to requesting nodes. Local message handlers on requesting nodes accept responses to the MCM requests sent out by the event handler and install blocks locally. The cache and LTLB of the requesting node is modified, and events pending to the block which were enqueued in the software event table are popped and satisfied at this time.

The following sections briefly describe the internal functions used by the MCM, present data structures employed by the home and requesting sides of the coherence protocol, and details the MCM implementation, including a state-machine model for tracking individual memory blocks.

7.1 Internal Functions

The MCM is split into three components which run as part of the event handler, and the two message handler threads. Table 7.1 lists the functions executed by the event handler thread. These functions may be grouped into three categories - functions which are executed as part of the requesting node's initial handling of blocks-status misses, functions which are executed in proxy for a requesting node's P1 Message Handler, and functions which are executed in proxy for a home node's P1 Message Handler. The proxy functions are actually wrapped up in the event handler's routine which services the software job queue, and are therefore shown in a stylized manner which does not actually appear in the source code.

The home node's MCM handles incoming requests for blocks, as well as acknowledgements for block invalidations which it sends out. These functions are outlined in table 7.2.

Lastly, the requesting node's MCM handles home node responses to the requests that were sent out by its own event handler. It also responds to invalidation messages coming from the home node. These functions are outlined in table 7.3.

7.2 Data Structures

Each node's MCM uses two data structures - one for managing blocks for which the node is a home node, and the other for tracking requests for blocks which the node makes in its capacity as a requesting node. The home-node information is stored in a coherence directory, while requested blocks are stored in a software event table.

7.2.1 Coherence Directory

The coherence directory is simply a linked list of lists. Each toplevel entry in the list contains the address of a block of memory which is shared by at least one node, state information about the block, and a list of nodes which share that block (these are nodes to which this block has been sent). Blocks may be in one of three states. Read shared blocks may have multiple nodes which share them. Exclusive shared blocks may only be held by a single node. Transitioning blocks are in the process of being revoked from all sharers because a conflicting request for them has been made (a request for a readonly or exclusive copy for a block which was held exclusive by a different node, or a request for an exclusive copy if the block was held readonly by at least one node).

Functions are provided to add a new sharing node for a particular block (`CCDirectory_addSharing`) to the directory, and remove a sharing node from the list of nodes sharing a particular block (`CCDirectory_popSharing`). Other functions access and modify block state.

This current implementation is not efficient in terms of search time. Future implementations of the directory should use a chained hash table to access shared addresses with greater speed.

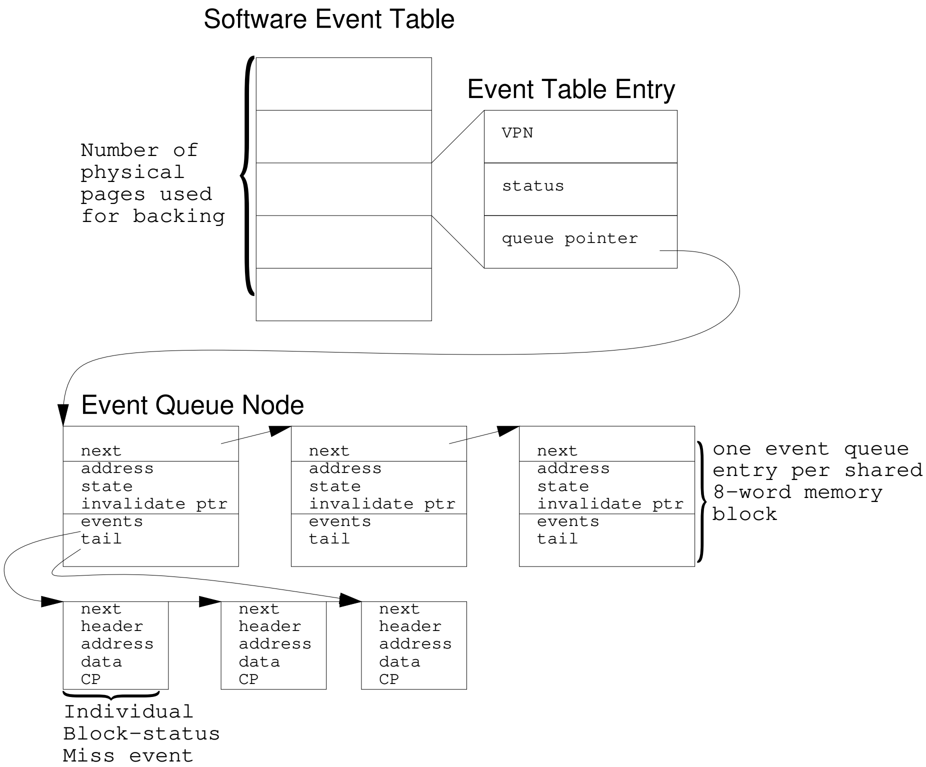

Figure 7-1: MCM Software Event Table

7.2.2 Software Event Table

The software event table is used by the cache-coherence manager to record block-status miss events which are being handled in software and maintain information about the status of blocks which have been requested from a home node. The table contains three-word entries and implictly maps physical page frame numbers to virtual page numbers and queues of requests. That is, the *ith* entry in the table refers to the *ith* page frame on the local node which is used as backing for remote virtual memory blocks. This table is statically-sized at link time, or at the time that the Physical Memory Manager is asked to reserve a range of frames for backing of remote memory with the `PPM_local2remote` function. Figure 7-1 shows event table layout.

The event table is probed with both a virtual address and a physical page frame number to access event queues for that block. The frame number is used to directly index into the table and locate a table entry. The table entry's virtual page number field is compared against the page number portion of the virtual address. If the numbers match, the pointer to the entry's queue of requests is followed (the structure of the queue is described below). If no vpn match is made, the frame number is considered stale, and a page-table probe (`PPM_lookup`) must be performed. In this way, the software event table functions almost like a reverse page table, except that information that it holds may be stale and inconsistent with the local page table. State information is associated with each table entry as well. Currently the only state information is a bit which informs the caller that the physical frame associated with the entry is marked for eviction, and no new events should be added to its queue.

The last component of the event table entry is the software queue entry pointer. This identifies the head of a linked list of queue entries. Each queue entry represents an 8-word memory block for which event information is stored. There may be at most 64 such entries in any linked list since there are at most 64 different blocks within a virtual page. Each entry contains information on the state of the block (to be discussed later), a 64-bit invalidation pointer if the home node has requested that this block be invalidated and returned [1], an address field which is used to identify which of the 64 blocks this block represents [2], and pointers to the head and tail of an event list for this block. The event list is a collection of entries which represent block-status miss events which have been removed from the hardware event queue by the event handler. Each miss event entry contains all four words which compose a block status miss, and a *next* pointer for use in linked lists.

Use of these data structures will be explained when implementation is detailed. To obviate the need for dynamic memory allocation of these structures, a collection of software queue entries and miss event entries are statically allocated at compile-time and initialized into lists of available entries at runtime. Entries are popped from the lists of free entries when needed, and returned to these lists when no longer used in the event table. Since the event table is statically-sized at compile time, it also does not need any dynamic memory allocation.

[1]Invalidation pointers are pointers to a *yankbuffer* structure, described later in this chapter.

[2]Although the current implementation uses a full 64 bits, only 6 are necessary since the rest may be reconstructed from the virtual page number of the containing event table entry.

7.3 Implementation

A memory-coherence protocol needs to handle a variety of common-case memory-sharing requests, and deal properly with a number of more unusual cases which are a result of the asynchronous nature of multinode execution. This section first presents a simplified view of common-case operation of the coherence protocol, introducing how the different handlers interact and employ the data structures that were presented in the last section. The motivation for employing a state-machine model of block states is presented, along with the model. Further sections then explain handling of more subtle coherence cases.

7.3.1 Simplified Roundtrip Coherence Path

Figure 7-2 is helpful in clarifying the mechanisms introduced in this section.

All nodes initially start execution without sharing any remote data. Threads which reference off-node data begin the process of remote-block fetching and installation. The process begins when a thread causes an LTLB Miss, since while a page of virtual address space may have physical backing on its home node, a remote node will not have such backing. A thread (called the *faulting* thread in the rest of this section) which references off-node memory will cause an LTLB Miss with its memory reference which will invoke the Physical Memory Manager as described in chapter 4. The PMM will determine that the virtual address is a remote-address and create a new page-table entry mapping the virtual page to a new backing page frame take from the remote backing pool. Block status bits for all blocks within the page will be set to *invalid*. When the hardware retries the memory-reference, an LTLB entry will be found, but block-status bits for the block containing the referenced address will be invalid. The hardware will therefore generate a Block Status Miss event and add it to the hardware event queue. The event, similar to the LTLB Miss Event, will contain a header word, faulting address, source data if the operation was a store, and a configuration space pointer into thread state for the faulting thread. A 20-bit field within the header word contains the frame number retrieved from the LTLB at the

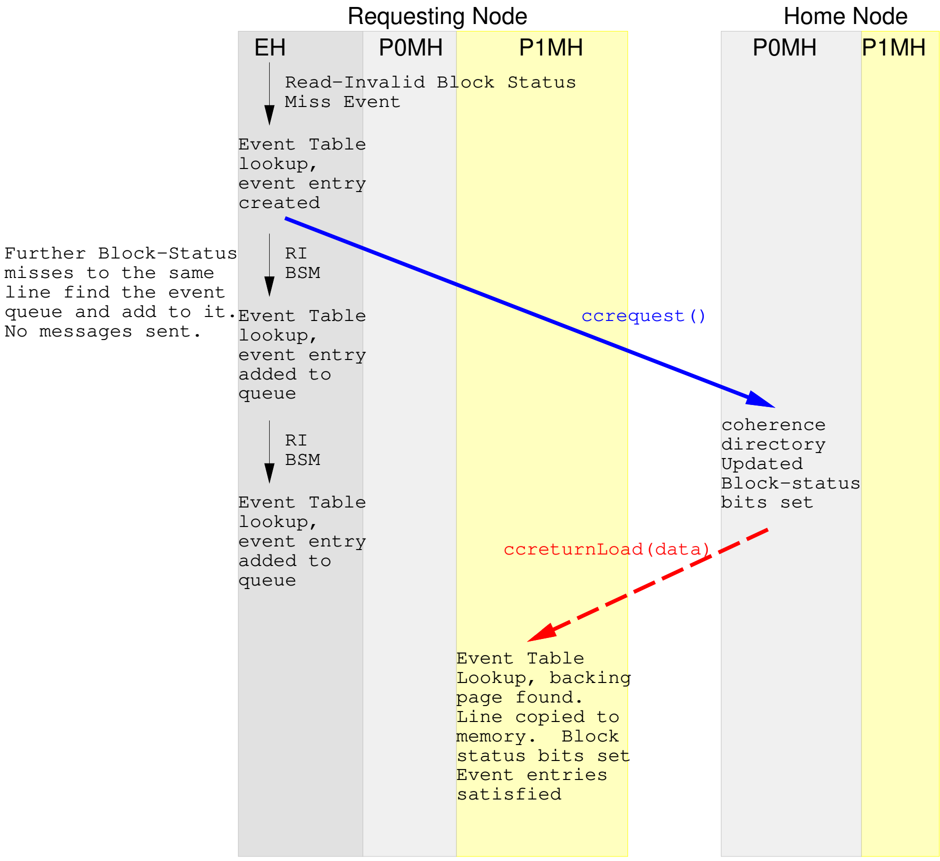

Figure 7-2: End-To-End Communication in Simple-Path Coherence Protocol

time that the block-status miss was generated. See table 7.4 for the event header format.

Sending a Request

When the event handler pops the block-status miss event from the hardware queue, it determines the type of the event from the low four bits of the header word. Finding that it is a block-status miss, the event handler dispatches the event to the `_BSM_xx` functions which interface assembly-coded portions of the event handler with higher-level functions written in C. The assembly code then calls `_EH_handle_bsm`, passing it all four event words. This function uses the header's encoded physical page frame number to index into the event table and find an entry. Initially, all entries within the table will contain invalid mappings (virtual page numbers of `-1`). Therefore, the event handler will not find a match between the faulting address' page and the page in the table entry. At this point, the handler decides that the page information is stale (it could have been changed between the time that the hardware determined the mapping from the LTLB and the time that the event handler had removed the event from the hardware queue) and performs a page table lookup (calls `PPM_lookup`). The resulting page is again used to probe into the table and again a match will not be found. At this point, the handler must deduce that the event table entry is not current, and creates a mapping, simply by writing the faulting address' virtual page number into the entry's vpn field.

Having found a valid table entry for the fault, the event handler examines the backing page's state information, to make sure that the page is not marked for eviction. Since it is not (the table is initialized so), the handler attempts to enqueue the block-status miss event. Since the page table's queue pointer is null, a new software queue entry is popped from the list of free entries and added as the head of a new list. Its address field is set to that of the faulting address with the low 6 bits masked off (indicating an entire 8-word block). A new miss event entry is also popped, initialized with the event words, and added to the event queue for the block in which the faulting address resides. The function returns certain flags which enable the caller to determine what actions to take. The *send_message* flag is set because a new software queue entry was added, and therefore this was the first reference to this block. The calling function (the event dispatch handler) then decides to send a message to the home node of the faulting address, requesting that the remote block be sent back. A `MSG_ccrequest` priority 0 message is sent, containing the header word and virtual address. At this point, the work of the requesting node's event handler is complete. The node must now wait for an acknowledgement to its request.

All further events targetting the block in the meantime are added to the event queue for that block so that spurious request messages are not sent. As long as there are events remaining in the software queue for a particular block, new events are added but no messages are sent.

Fulfilling Requests

When it receives a `MSG_ccrequest` message, the home node's priority 0 message handler removes the message arguments from the message queue, packages them as function arguments, and calls the `ccrequest` function of the MCM. ccrequest examines the event header which was sent in the message and determines whether the request was for a readonly or an exclusive block based on the opcode of the operation that faulted on the requesting node. A `ld` operation results in a call to `ccrequest_ld` while a `st` or any of the synchronizing ld/st variants result in a `ccrequest_st`, `ccrequest_stsu` or `ccrequest_ldsu` being called.

In any case, the home node checks the coherence directory to determine what is the state the requested block. Assuming that this is the first coherence request to be serviced, the directory will return the fact that the block is unshared. In this case, the directory is modified to have the requesting node as a sharer for the block in question. If this was a store request, the store which was requested to be performed is performed locally (the opdata passed in the request message is used as the data source of the store operation). Block-status bits for the block are then changed to INVALID, and the block is read out and sent as an acknowledgement to the requesting node. In response to a load request, the block-status bits are changed to READONLY since the home node's thread can continue reading the block, and the block is read out and sent to the requesting node.

Installing Remote Data

On the return path, the acknowledgement to the a block request returns to the requesting node as a `ccreturnLoad` or `ccreturnStore`, depending on the type of sharing which was granted (exclusive or readonly). In either case, the address and header which return in the acknowledgement are used by the MCM to index into the event table in the same manner as performed by the event handler. This time, there is a match between the entry's vpn and the vpn of the requested address (since this was correctly updated by the event handler prior to the request message being sent) and the entry's software queue pointer is followed and the queue entry for the appropriate block is found. The block contents are read out of the message queue by an assembly function and written into local memory (a backing page exists since there is a mapping in the event table from the ppn listed in the header, and the vpn in the faulting address). Block-status bits for the virtual address of the block are set properly (READONLY or READWRITE, depending on the type of sharing allowed). All events stacked up for the requested block are then handled in turn, by performing the faulted memory operations, this time on memory which has been installed locally. After all events have been processed, the event entries and software queue entry are returned to their free pools, and, if no other cache blocks have been requested for that particular virtual page, the pointer to the software queues in the table entry for the backing frame is reset to NULL.

7.3.2 Diverging from the Simple Case

This section begins to explore the more interesting cases which must be dealt with by the MCM. Each section will identify a case not covered in the above simplified example and ammend the actions taken by affected components. The cases will parallel the previous section in the order of the components that are introduced - starting with the event handler.

Out of Backing Pages

In the previous section, the page frame number located by the M-Machine memory system was assumed to be a valid physical page frame. As mentioned in section 4.5.2, the ppm will create a mapping of a virtual page number to physical page frame `-1` if no backing frames for remote data remain. This information may be returned in the event header of a block-status miss. **It is the policy of the MCM not to send requests for remote blocks unless physical backing is obtained first**. Therefore, the MCM first performs a `PPM_lookup` to make sure that a mapping hasn't been created since the block-status miss first occured. If the lookup returns a valid page, the event handler can perform the probe as before and continue processing.

On the other hand, if an invalid mapping is returned again, the event handler makes note that cleaning of shared pages must be performed to free up a backing frame, and adds the entire event to a local software queue, effectively recirculating it so that it may continue taking a look at the event from time to time and being able to finally satisfy it when physical backing is obtained. Meanwhile, to prevent user threads from continuing to cause block-status misses and overfilling the recirculation queue, all user threads are prevented from issuing instructions (the event handler turns off their `hIssue` thread state bits).

In order to find pages suitable for reuse, the event handler may run through the event table, looking for entries which have no pointers to software queues of events. Such pages are ripe for eviction since no outstanding requests to their pages remain and therefore all of the shared blocks within these pages may be evicted (and sent back to their home nodes if dirty). In order to evict a shared page, the event handler performs the following actions:

1. Performs 64 *putcstat* operations, setting block-status bits for each block within the page to *invalid*. Putcstat's return value, the previous block-status bits, are used to check whether each block was dirty. Every dirty block is shipped back to the home node with a `sysPushDirty` call, which sends the address and the 8-words of the block to the home node in a `MSG_ccreturnDirty` message.
2. Calls `PPM_unmap` to remove old virtual-physical mapping for the virtual page being evicted.
3. Returns the backing page to the backing page chain with a call to `PPM_reclaim_remote`.

After a virtual page has been evicted and the backing frame is returned for reuse, the event handler makes a `PPM_map` call to give physical backing to a new virtual page, which was missing backing previously. Finally, the entry corresponding to the newly-acquired backing page is modified to reflect a new virtual page number, and the process of adding a new software queue entry may continue as before.

If no pages may be evicted right away (each entry in the event table has a valid software queue pointer, signifying that there is at least one outstanding event per page waiting for a block to be returned), some pages are chosed for eviction and their state bits in the event table are set, indicating that no new events are to target these pages since they must be evicted.

In order to prevent running out of backing page frames, the event handler is designed to examine the number of page frames remaining after each event is handled. If the frame count is below a watermark, the handler must perform preemptive page eviction to free up backing frames. This may be accomplished by keeping a pointer into the event table which is advanced until a suitable candidate frame (one with a valid VPN mapping, but no queue pointer) is found. This frame undergoes the eviction process described in the steps above and may be added to the backing pool.

Backing Page is Marked for Eviction

The case in the previous section presents another problem for the event handler. If it finds an event table entry for the faulting address and the virtual pages match, the physical page frame may be locked. If the state bit for that entry is set, the event handler is prevented from adding the new event to the software queue (although one optimization is to allow it to add the event if it targets an existing block, so that the event will be handled with all other events for that block as soon as the home node returns the necessary data) and must recirculate it. This case becomes analogous to the event handler not having an appropriate backing page, although in this particular instance no search for new backing pages is required.

A modification to the priority 1 message handler which deals with returning blocks must be made as well. When the last software queue entry for a particular virtual page has been freed and the event table entry's software queue pointer set to NULL, the message handler must check the status bit of that entry. If the status bit is set, the page is ready for eviction. Since the P1 message handler is not allowed to send out messages (this is to avoid deadlock in the machine's network) and message-sends of dirty blocks may be required when performing a page eviction, the P1MH enqueues an eviction job with the event handler in the handler's software job queue. Some time in the future, the event handler will respond to the eviction request and perform the same type of operations in evicting a page as mentioned in the previous subsection.

Invalidations Required

Moving to the home node of requested data, the case of incompatible block sharing arises. As mentioned briefly at the beginning of this chapter, when the home node probes the coherence directory, it may discover that several nodes are sharing a block which has just been requested as an exclusive copy; or a node other than the requesting node may have an exclusive copy of the block. In both cases, all of the nodes currently sharing the block must have their shared copies revoked, before the latest request can be satisfied.

The home node performs the invalidation with the help of a new data structure - the yankbuffer. The yankbuffer records information about the request which caused the invalidation to be performed, and the number of invalidation messages outstanding. A circular buffer of pointers to free yankbuffers is accessed to acquire a new yankbuffer. This circular buffer is then used to return a yankbuffer for reuse once the invalidation process has completed. The invalidation protocol begins as follows: a new yankbuffer is acquired and the four words of request information written into it. The requesing node number is written as well, so that the MCM knows which node sent this request. Lastly, the number of nodes which currently share the block is written into the yankbuffer.

With the yankbuffer initialized, the message handler sets the state of the block in the coherence directory from shared exclusive or shared readonly, to TRANSITIONING, signifying the fact that an invalidation of this block is in progress. The message handler begins popping nodes from the coherence directory list of sharers for the requested block and sends an `MSG_ccinvalidate` message to each. The block address and yankbuffer address are sent in each message. Once all messages have been sent, the message handler's immediate task is complete, and it is ready to handle the next incoming message. Other portions of the MCM will respond to the invalidations and cause the block to be sent to the requesting node which caused the invalidations.

As acknowledgements to the invalidation messages arrive at the P1 message handler (invoking the `ccreturnYank` and `ccreturnyankFull` functions), the yankbuffer pointer that is sent along is used to decrement the invalidation count within the buffer. Dirty blocks which are returned in acknowledgements are copied into home node local memory.

All requests for blocks which come in while the blocks are in the transitioning state are NACKed back to their senders. This frees the home node from buffering requests for blocks locally, and instead places the burden of buffering on the network and requesting nodes, as NACKs are returned to home nodes, buffered, and new requests sent out.

Once the invalidation count reaches zero, the state of the requested block on the home node may be returned to the exclusive-copy state since (1) all node which had previously shared the block have acknowledged that they no longer hold copies, and (2) no new copies were given out since any new requests are met with a NACK. The state in the coherence directory remains transitioning, however. The original event is read out of the yankbuffer and added as a job to the event handler so that the full block-request code may be executed. The event cannot be handled directly by the P1MH since a reponse to a block request involves a message-send, which it not allowed for the P1MH. The state of the block in the coherence directory remains transitioning, to make the window of vulnerability when another request may come in an acquire rights to the block ahead of the original request as small as possible. The yankbuffer is returned to the circular buffer of free yankbuffers.

As the event handler performs the request procedure, it removes the block from the coherence directory (since no node is sharing the block) and calls the `ccrequest` function (normally called by message-dispatch code) directly, passing it the event information enqueued in its software job queue entry. At this point, the entire invalidation procedure is complete and the request which originally caused the invalidations gets another chance to acquire the block.

Receiving NACKs

In the previous subsection, the home node was shown to be capable of sending NACKs in response to block requests. This section describes how the requesting node's P1MH must deal with NACKs. Since the events which caused the request messages to be sent are still enqueued in software, the MCM does not need to perform another lookup in the event table when it receives a NACK. Intead, it needs to add a job for the event handler to resend the NACKed request. The actual NACK message which is sent by the home node contains the entire contents of the original request message. This makes it quite a simple task for the P1MH to add a resend request for the event handler - it passes all of the words of the NACK message to the EH. The event handler will dequeue the request some time in the future and retransmit the request. Once again, the reason that the P1MH cannot retransmit the request on its own is to avoid deadlock in the network - the P1MH is not allowed to send out any messages. Figure 7-3 summarizes the invalidation protocol.

Performing Block-Invalidation

Another task that the MCM must now perform is invalidating shared blocks in response to invalidation requests from the home node. When an invalidation message arrives, it bears only the virtual address, and does not contain any physical page frame information as events do. Therefore, the P0MH which handles the invalidation request must perform an explicit `PPM_lookup` to determine the local page frame which is used for backing the virtual page in question. The `putcstat` operation is performed on the virtual address to set block-status bits to *invalid* and return the previous state of the block. If the block was dirty, the page frame number is used along with the low 12 bits of the virtual address to determine the offset within the page frame where the block resides, and to read the block out into an acknowledgement message to be sent to the home node. In any case, the invalidation is acknowledged with either a simple ACK or an ACK bearing the contents of a dirty shared block. The invalidate ACK also contains the yankbuffer pointer which was passed in the invalidate message. As described above, this yankbuffer pointer is used on the return trip by the home node's P1MH to decrement the invalidate counter and decide when all nodes which shared the block have relinquished their copies.

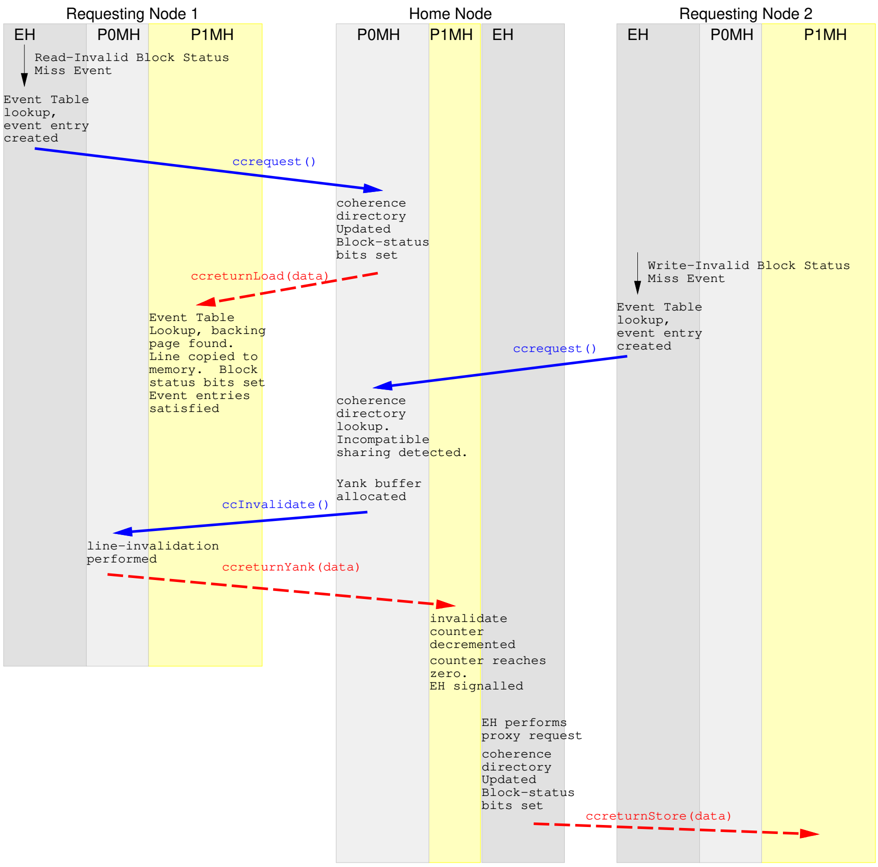

Figure 7-3: Block Invalidation in Memory Coherence Protocol

Dealing with Orderless Messages and Asynchrony

The protocol design presented so far seems to handle a variety of special cases, but the more interesting remain to be covered in this section. Particular problems arise when guarantees on message-ordering don't exist[3], and when asynchronous invalidation and NACK messages must be dealt with.

A requesting node may receive an invalidation message while it is still installing a newly-acquired block. Should the original ACK message to the block request be crossed with a later invalidation message, the requesting node may even receive the invalidation message before the actual data ACK arrives. To handle these cases, the

[3]At the time of the coherence protocol design, the M-Machine did not guarantee message ordering. The machine hardware has since been ammended to allow in-order messages to be used.

coherence protocol employs a state-machine model for memory blocks. That is, each block which has an entry in the event table has associated with it a state. This state helps MCM components decide what to do when messages or events concerning that block arrive. A block state is represented using five bits which encode the history of requests and responses targetting that block. These bits are:

1. `PX` : Pending Read/Write Request
2. `PR` : Pending Read Request
3. `I` : Block Needs to be Invalidated
4. `AX` : ACK to R/W Request Received
5. `NX` : NACK to R/W Request Received

Initially, a software queue entry for a block gets its state set to PX or PR, depending on whether a readonly or readwrite copy of the block was requested from the home node. This records the fact that a request for the block has been sent to the home node and the requesting node is waiting for a NACK or ACK to return. In some instances, both PX and PR bits will be set - this occurs when first a read-invalid block-status miss is handled and the event handler sends out a request for a readonly copy of the block. Later, store to the same block will cause a write-invalid miss which will require that an exclusive copy of the block be requested. The EH will alter the state of the block from `PR` to `(PR | PX)` to note that two requests have been sent.

Should an invalidate message arrive before the actual data returns, the `I` state will be added to the block state. This will allow the MCM to keep track of the fact that after the request is ACKed or NACKed, an invalidation should be performed. The MCM cannot invalidate its block immediately after the invalidation message arrives because the invalidation and reponse to a block request could have gotten crossed, resulting in a block coming back later which should have been invalidated. If an invalidate message arrives when the block state is zero (meaning no software queue entry even exists for the block), it is safe to perform the invalidation immediately since no request messages for that block have been sent.

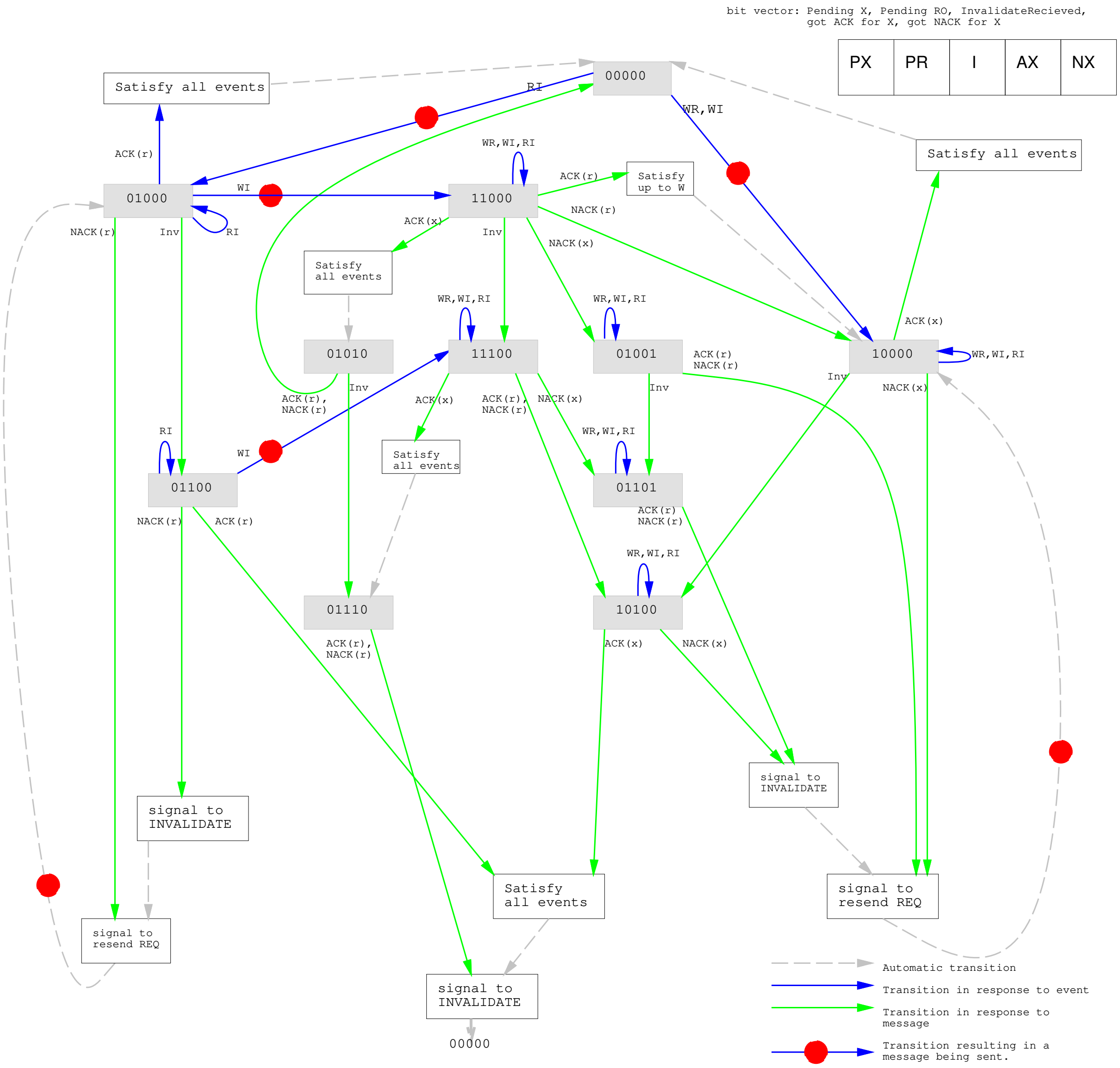

Figure 7-4: State Transition Diagram for Requested Blocks

Transitions which are performed when new messages arrive at the requesting node, or when events occur, can then be defined on these states. A state-transition diagram which is to be followed for each block is shown in figure 7-4. In this figure, transitions are triggered by the arrival of messages (NACK(X), ACK(X), NACK(R), ACK(R)) and new events (RI for read to an invalid block, WI for write to an invalid block, WR for write to a readonly block). This transition diagram clarifies the job of the MCM. When an invalidate message arrives for a block whose state is PR or PX, for instance, the invalidate bit is added to the state and the yankbuffer pointer is added to the software queue entry for that block (hence the need for an invalidate pointer entry in the event queue data structure). If an ACK for the block is returned, the block will be installed, all of the events pending to it will be resolved, and then a job enqueued with the event handler will request that the handler perform an invalidation phase. The previously-stored yankbuffer pointer will be used when the block is invalidated and shipped, if necesary, to the home node. If a NACK arrives, a job for the event handler will be enqueued so that first the block is invalidated, and then a new request for the block is sent (since the MCM always resends requests if it receives a NACK).

This state-machine model tolerates out-of-order messages and asynchronous invalidations by imposing a rigid flow of control on the MCM and only allowing actions to be taken if the block is in a known and consistent state with the action being performed: for instance invalidated if no more messages are pending for that block.

With the state-machine model in place, the MCM design becomes more complete. When an event entry is enqueued for a particular block, the software queue entry's state is updated with proper PX and PR bits. An invalidation message handler (the code in ccInvalidate) first checks the state of a block to determine which state-transition to perform. Similarly, the P1MH checks block state when ACKs and NACKs arrive, to determine which actions to take. Usually in response to ACKs, this involves installing the block and then transitioning to a completed state or enqueuing jobs with the event handler to perform latent invalidations. In response to NACKs the actions are to enqueue jobs with the event handler, the nature of the jobs dependent on the block state - either invalidate-and-send-request, or send-request if no invalidation is required.

Dealing With Concurrency

Since all of the threads which work in concert to provide memory-coherence need to access the MCM data structures (the event table on the requesting node, and the coherence directory on the home node) locks are used to enforce serialized access. The current implementation uses extremely coarse interlocks - a single lock is assigned to each data structure (`sqlock` for the event table, and `ccdirlock` for the coherence directory). These locks must be acquired before functions which access and/or modify their associated data structures may be called. It is important to note here that regardless of lock granularity, the system must be implemented in such a way, that the event handler and P0 message handler may not hold locks which will prevent the P1MH from making progress at the time that they send out messages. As mentioned several times before, this is to prevent deadlocks from occuring - the P1MH must always be able to make progress and service its message queue even if other OS components such the the MCM running in an event handler slot are blocked, waiting to send a message into a saturated network.

When blocks are being installed, it is sometimes worthwhile for the P1MH to unlock the event table each time that it pops a new event from the event table which targets that block. This allows the event handler which is popping block status misses from the hardware event queue to add the event to the event table even while previous events are being popped off. This prevents spurious messages from being sent for blocks which are already installed locally. However, the system must be able to handle the case that a spurious message is sent. This may occur if the P1MH runs through all of the event table entries for a block that it received and then removes all traces that the block was installed, by deallocating the software queue entry for that block. Meanwhile, a latent block-status miss to the newly-installed block may be popped by the event handler, and a new event entry will be created. Since no software queue entry will have been found, the event handler decides to send a request message for the block. The home node will notice that the requesting node had already been listed as a sharer of the block, but will oblige with another copy. This allows requesting nodes to flush their shared blocks without having to inform the home node. The only side-effect of this flushing is that unnecessary invalidation messages may sometimes be sent by the home node.

Normally, installing a duplicate block is not a problem. However, if the original block was installed as exclusive, it may already be dirty by the time the second (and stale) home node's copy of the block arrives. This means that the requesting node, when executing the code in `ccreturnStore` may not blindly install the block that it received in the ACK message. Instead, it must check the existing block-status bits of the block which was previously installed (local block) and determine whether the block is dirty or not. If the local block is dirty or readwrite, the stale copy is not installed. If, however, the local block is in the a readonly or invalid state, the block in the message is installed. In either case, all events pending to the block are satisfied as before.

Function	Type	Description
`INVALIDATE(int node, void *address, void *yankbuffer)`	Request Proxy	Invalidates the memory block identified by *address* from the local cache and sets block-status bits to *invalid* in the LTLB and/or page table. Sends an acknowledgement to the home node *node*, sending along the *yankbuffer*. If the block was dirty, sends the dirty block within the acknowledgement.
`RESENDSTORE(int header, void *address, int opdata, void *faultCP`	Request Proxy	Sends a ccrequest message to the home node of *address*.
`RESENDLOAD(int header, void *address, int opdata, void *faultCP`	Request Proxy	Sends a ccrequest message to the home node of *address*.
`INV_STORE(...)`	Request Proxy	Combination of the INVALIDATE and RESENDSTORE/RESENDLOAD cases above. First invalidates a block and returns it to the home node. Then sends a request for it.
`INV_LOAD(...)`	Request Proxy	Same as above
`REQUEST(int header, void *address, int opdata, void *faultCP)`	Home Proxy	Executes the function ccrequest as if a request message for the block identified by *address* was received.
`EH_handle_bsm(int header, void *address, int opdata, void *faultCP`	BSM Handling	Responds to a local Block-Status Miss event. Enqueues the event (composed of the 4 argument words) into the software event table and returns status flags which tell the calling function what type of request message (if any) to send out. Returns 0 on error. A flag of 0x1 means no failure was detected. A flag of 0x2 means that a ccrequest message should be sent to the home node of *address*. A flag of 0x4 requests that the thread which caused the event be prevented from issuing any more instructions. A flag of 0x8 means that teh event request should be recirculated and tried again later.

Table 7.1: Event Handler's MCM functions

Function	**Type**	**Description**
`ccrequest(void *address, int header, int opdata, void *faultCP, int node)`	priority 0	Processes a request for the block containing *address* from node *node*. Dispatches to helper functions `ccrequest_st` and `ccrequest_ld` depending on the type of operation encoded in *header*. May also call `ccyankline` if a shared block needs to be revoked from current sharing nodes. Sends a response to *node*, bearing the requested memory block or a NACK, or has the event handler do so in proxy at a later time.
`ccreturnyankFull(int *yank_buffer)`	priority 1	Processes an acknowledgement to an invalidation message. The acknowledgement contains a dirty block which must be installed locally. Once installed, the original request which lead to the invalidation is processed in proxy by the event handler.
`ccreturnYank(int *yank_buffer)`	priority 1	Processes an acknowledgement to an invalidation message. Decrements an invalidation counter for each such acknowledgement received. If the counter reaches zero, the block is considered unshared again and the request which lead to the invalidation is processed in proxy by the event handler.

Table 7.2: Home Node MCM functions

Function	**Type**	**Description**
`ccNackRO(void *address, int header, int opdata, void *faultCP, int node)`	priority 1	Deals with a NACK returned by the home node in response to a readonly sharing request. Usually, the event handler is asked to resend the original request, to the home node, *home*. The first four arguments to this function are the arguments which were returned in the NACK, and used in the repeat request by the event handler.
`ccNackRW(...)`	priority 1	Same as above, except that the NACK is in response to an exclusive block request.
`ccinvalidate(void *address, void *bufPtr, int node)`	priority 0	Responds to an invalidation request from the home node, *node*, of the block identified by *address*. Takes steps to invalidate the block locally and, if dirty, to ship it back to the home node.
`ccreturnLoad(void *address, int header, int node)`	priority 1	Installs the block which is returned in response to a readonly sharing request from node *node*. The 8 words of the block remain in the hardware message queue and are read out by an assembly-level helper function.
`ccreturnStore(...)`	priority 1	Same as above, except that the block is installed for read/write as an exclusive copy.

Table 7.3: Requesting Node MCM functions

Description	bits
OP_ACTION	56 - 63
issuing thread slot	48 - 55
issuing functional unit	42 - 47
issuing cluster	40 - 41
target register file	36 - 39
target register	32 - 35
target cc	28 - 31
precondition	26 - 27
postcondition	24 - 25
physical page frame number	4 - 23
event type	0 - 3

Table 7.4: Event Header Format

Chapter 8

Exposing System Calls to User Threads

The runtime system managers mentioned in previous chapters need to export certain system calls to user programs. This is accomplished through the use of jump tables and load-time program patching - mechanisms described in this chapter.

In order to allow user programs to safely access certain system function entry points, the programs need to be given *entry* pointers into runtime system code which they may then use to perform `jmp` instructions. The runtime system currently uses an object file called `syscall.o` which is linked with every user-level executable. This file contains stubs for all exported system calls which the program may wish to use. The stubs are simply functions which load system entry pointers from memory and jump on them. Entry pointers are loaded from locations in the data segment which are flagged to the loader as needing to be patched. This simplifies interfacing with the M-Machine compiler, since the compiler has no notion of which functions are system functions. Therefore, it expects to be able to place references to external system functions and have them resolved at link time. Again, this is already accomplished by having syscall.o contain stubs for all system functions, which means that from the point of view of the compiler and linker, a user-level executable has all of its symbols resolved before it is loaded. Figure 8-1 shows an example of a stub written in M-Machine assembly.

```
_tFork::
        GET_FRAME
        LOAD_FAR_LABEL(_tFork_ptr, itemp0, DStart)  /* load ldptr value */
        instr ialu jmp itemp0;                      /* jump to system code */
        instr ;
        instr ;
        CALC_RETIP

        RETURN                                      /* return to caller */
```

Figure 8-1: Sample syscall.m stub

Since stubs load system entry pointers from memory and the values of these entry pointers are known only at load time, the syscall.o object file contains magic numbers and relocation entries within it which signify that certain locations of its data segments need to be patched with pointers at load time. These pointers are called `ldptr` in the assembly language, and have their own relocation type. The trusted loader reads the object file, looking for ldptr relocations and replacing the contents of the data segment where the ldptrs are stored with entry pointers into system code. The magic numbers stored where the ldptr's are defined are used to determine which system function entry pointer needs to be stored there. The trusted loader is passed a table of associations between magic numbers and system entry pointers. This allows the syscall.o to create a table of ldptr values in its data segment, and use the stubs to load these values and jump on them. This patching is safe, since the user cannot trick the loader into giving out privileged information - any entry pointer which can be given out defines a protected entry point, and only entry pointers which the OS is willing to give out are passed to the loader. Examples of ldptr usage from syscall.m are shown in figure 8-2.

The entry pointer table passed to the loader is constructed at boot time, with values which are taken from system call function stubs, offset from the runtime IP. These are usually physical addresses. System call function stubs exist for each actual system function and act as an interface to the system function. Once called, the stubs perform two tasks. First, they issue an *mbar* instruction, which insures that

```
data;
align 0 mod 8;
_tFork_ptr::     ldptr 0x0000ffffaaaaaab0;
_tExit_ptr::     ldptr 0x0000ffffaaaaaaab;
```

Figure 8-2: Sample syscall.m ldptr usage

```
_tForkX::
       instr memu mbar;     -- issue mbar right away to keep registers safe
       GET_FRAME
       PUSH(DStart)         -- save caller's data segment pointer
       instr ialu imm __SYSTEM_UDAT_PTR, itemp0;    -- offset where system's
                                                    --   data pointer is stored
       instr ialu leab IP, itemp0, DStart;          -- create a pointer
                                                    --   to this offset
       instr memu ld DStart, DStart;                -- load system's data
                                                    --   pointer off the IP
       FCALL(_SYStFork)                             -- call the actual runtime
                                                    --   system function
       SPOP(DStart)                                 -- restore user's data ptr
       RETURN                                       -- return to caller
```

Figure 8-3: Sample runtime stub

any memory operations which the caller performed will complete and overwrite any registers before the stub continues execution. This prevents a malicious user from issuing memory operations which may overwrite the register set of structed code as it begins execution. Secondly, while the IP of the executing system code points into runtime system space (as opposed to the user-level caller's space), the data segment pointer still points to the caller's data segment. The system function stub saves away the existing data segment pointer, and then loads the runtime system data pointer off its IP. The runtime data segment pointer is stored there at system boot time, for the express purpose of making it available to system-function callees. A runtime stub for the tFork system call is shown in figure 8-3.

Chapter 9

Performance Measurements

This chapter presents performance measurements of some runtime system components. It should be noted that although cycle-counts are included, these numbers are the result of executing a runtime system which was compiled with a compiler still under development and with absolutely no optimizations being performed. The more interesting numbers to examine are the breakdown of cycle-counts within long-latency operations to determine where most of the time is being spent.

9.1 The LTLB Miss Handler and Physical Memory Management

Tables 9.1 and 9.2 list the cycle counts of performing physical memory management tasks by the LTLB Miss Handler. Note that table lookups are quite fast, but the time to create a new mapping, which involves acquiring a new page frame from the free page list, is the largest component of an LTLB Miss.

9.2 Virtual Memory Allocation

The virtual segment manager takes an average of 950 cycles to allocate and return a virtual segment. A selected run is shown in table 9.3.

Subcomponent	Cycles	Notes
Initial LPT lookup	283	Lookup fails
Create new mapping	1398	Creates new virtual-physical mapping
Second Lookup	236	Added entry now found
Find conflicting LTLB Entry	266	For evicting existing LTLB entry
Writing new LTLB Entry	231	Evict old entry and write new one
Other	1423	
Total	3837	Total time to handle a miss to an unallocated page

Table 9.1: Cycle count breakdown of LTLB Miss Handling

Function	Cycles	Notes
PPM_lookup	1281	Lookup a mapping in the page table
PPM_unmap	1789	Remove a mapping from both LTLB and the page table

Table 9.2: Cycle counts for selected PPM functions

9.3 Thread Management

Table 9.4 shows that aside from thread context allocation and initialization, forking off a thread is quite inexpensive. This suggests that keeping available thread contexts around after they are destroyed may help improve performance.

9.4 Memory-Coherence

Table 9.9 shows a cycle-breakdown for handling a block-status-miss by the event handler. Note that while the event table is being updated, the update is not being directly simulated. It is expected that this time will be quite substantial. Cycle counts

Subcomponent	Cycles	Notes
Jump to protected subsystem	95	Including an mbar and restoring system data ptr
Allocate new segment	801	Actual buddy list allocation
Return from subsystem	31	Includes restoring user's data ptr
Total	927	Total time to allocate a virtual segment

Table 9.3: Cycle count breakdown of Virtual Memory Allocation

Subcomponent	Cycles	Notes
Subsystem entry	89	
Allocate new thread context	935	See VSM times in previous section
Initialize thread context	7279	
Allocate thread stack	1248	
Add job to EH job queue	1033	Tells EH to add thread to pending list
Add job to EH job queue	1025	Tells EH to perform scheduling
Return from subsystem	126	
Other	1837	
Total	13572	Total time to fork off a thread

Table 9.4: Cycle count breakdown of tFork

Subcomponent	Cycles	Notes
Pop from pending list	162	Get a new candidate
Install candidate	2725	Includes copying entire register state
Other	348	
Total	3235	Total time to install a thread into empty slot

Table 9.5: Cycle count breakdown of tInstall

Subcomponent	Cycles	Notes
Subsystem entry	104	
Signal T_CHILD_EXIT	4685	Includes allocating signal entry
Add EH job	788	Add EXIT signal
Other	1316	
Total	6893	Total time for a thread to call tExit and block

Table 9.6: Cycle count breakdown of tExit

Subcomponent	Cycles	Notes
Send spawn message	2018	Includes nonce allocation (1718 cycles)
Perform tSleep on nonce	2800	
Return Signal Message Processing	4453	Time to wake from when signal arrives
Total	9217	Does not include time that thread was sleeping

Table 9.7: Cycle count breakdown of sender tSpawn

Subcomponent	Cycles	Notes
Perform local fork	9267	
Perform signal on nonce	564	Sends message to spawner's node
Other	326	
Total	10157	Doesn't include time that remote caller was sleeping

Table 9.8: Cycle count breakdown of receiving tSpawn request

for handling BSM's which don't require message-sends average about 410. This means that there is about a 700-cycle premium to sending out a request message, putting a thread to sleep, and performing other bookkeeping.

Subcomponent	Cycles	Notes
Assembly prologue	37	Time to call C handler function
Add to event table	174	This is not simulated
Stop thread from issuing (icache miss)	261	
Request Message send	126	Read out data and send request message
Other	495	
Total	1093	Time to handle a block-status miss

Table 9.9: Cycle count breakdown of handling a BSM

Table 9.10 shows cycle breakdowns for handling a coherence request by the home node. Note that as above, the coherence directory code is not being simulated and is expected to be a substantial portion of the total execution time. The total roundtrip time from block-status-miss to completion of line installation is about 8400 cycles, or about 1050 cycles per event to that line (the cycles of adding events after the initial request has been sent overlap the response times).

Subcomponent	Cycles	Notes
Page-table lookup	1471	
Reading and sending cache line	106	
Other	1182	Includes coherence directory modification
Total	2759	Time to handle a cc request

Table 9.10: Cycle count breakdown of home node's handling a ccrequest

Subcomponent	Cycles	Notes
Read line from message and install	75	
Pop and satisfy 8 events to line	3326	(415 cycles/event)
Other	1116	
Total	4517	Time to handle a cc ACK

Table 9.11: Cycle count breakdown of requesting node's handling an ACK

Chapter 10

Status and Future Directions

In this chapter, I present a broad overview of the currently-implemented MARS components and chart a course for what work remains to be done to develop MARS into a truly robust system.

10.1 Key OS Features and Contributions

The operating system presented in this thesis is quite novel. This is in great part due to the unique hardware platform to which MARS is tailored. The M-Machine's support for multiple thread contents, hardware-based capabilities, and configuration-space access to hardware state has been presented. What sets MARS apart most strongly from existing operating systems is its reliance on a collection of concurrently-executing managers to perform OS functions, instead of a single monolithic kernel or even microkernel. Most systems, regardless of light or heavyweight nature of the kernel, still require user-level programs to fault into a single-threaded kernel. With up to four system-level handlers able to execute at the same time, (and several additional protected subsystems in user slots) MARS is a truly decentralized operating system. The highest priority thread - the PMM - is still just a single thread performing only physical page management.

The use of capabilities by the OS can dramatically enhance performance. By turning thread context pointers used by MARS into Key pointers and giving them away to user-level threads, the OS is able to obviate the use of more levels of indirection in order to protect threads. At the same time, once the thread context pointer is passed to a trusted OS component, the conversion of pointer type allows the system to access thread state very quickly, without requiring a lookup table. Capabilities are also used by the loader and runtime system to export system calls to user threads. Again, because no fault is required to enter a trusted subsystem, and because system-level code may execute in a user-level thread slot, performance of other threads is not affected.

By coupling a single virtual address space (in itself not a novel idea) with capabilities, MARS is able to provide efficient shared memory for all user and higher-level system threads. No special provisions are required to map virtual address spaces independently for each thread. A single virtual address map simplifies page and thread management. Context switches need only deal with register contents and other localized thread state.

Finally, the low-level support for coherent memory across the nodes of a multiprocessor makes MARS quite unique. Although operating systems like Mach may rely on hardware-based coherence, or allow a software coherence layer to be built independently using add-on memory managers, MARS takes a middle-ground. This results in memory-coherence more flexible than if built into hardware, at a performance cost. Because the coherence system is built on such a low level - within the message and event handlers - higher-level components are free to execute in such an environment. For example, the system-level loader can easily distribute a data segment of a newly-loaded executable over several nodes without requiring explicit message-passing. Simply storing a large array into virtual memory striped across several nodes will transparently distribute it. This makes the task of writing not only user-level programs, but also other system routines much simpler. This is certainly demonstrated by the ease with which multithreaded shared-memory code may be written under this OS (as shown by the example programs in appendix E).

10.2 Existing Components

The MARS system is composed of a collection of assembly and C source code files which compiled, assembled, and linked into a single executable. This executable is loaded into the M-Machine Simulator for testing and development work, and runs completely in physical memory.

The bootstrap - the `boot.m` assembly file - is the first to execute. It spawns off remaining system threads and performs initialization of the four managers presented in previous chapters. Each of the four system-level handlers contains an assembly-level portion which sets up arguments by popping events from hardware-mapped registers and calls on higher level functions written in C. The handlers are the event handler (`event.m`), P0 and P1 message handlers (both in `message_event.m`), and LTLB miss handler (`ltlb_event.m`). Sthe `syscall.m` assembly file contains stubs which allow user-level programs to call on exported system calls. This file is assembled and linked with user programs and is not linked into the runtime system.

The components written in C are divided on a roughly functional basis.

The physical memory manager is composed of the `ltlb_body.c`, `ppm.c`, `lpt.c`, and `pplist.c` files.

The virtual-memory manager is composed of stubs in `vmem.m` and actual routines in `buddy.c`.

The thread manager is divided into `tmanager.c`, `tmanager2.c`, and `tsignal.c`, with certain stubs written in `boot.m`.

Cache-coherence code is in `cc_home.c` and `cc_request.c`, with stubs in `boot.m` to handle message-sends and line-installation.

The cache-coherence data structures are actually compiled into the M-Machine simulator instead of being part of the runtime system. Both the cache-coherence directory and the event table are some of the largest components which remain to be fully implemented within the runtime system. Table 10.1 shows the breakdown of OS components by source file.

File	Description
`boot.m`	Main system bootstrap. Also includes several assembly stubs for special instructions and message sends.
`event.m`	Event handler H-Thread source code. Marshalls arguments before calling code in `eh.c`
`ltlb_event.m`	LTLB Miss Handler H-Thread/PMM source code. Marshalls arguments before calling code in `ltlb_body.c`
`message_event.m`	P0 and P1 Message Handler source code. Interfaces to routines in memory-coherence and thread management functions.
`sysloader.m`	Loads user programs into memory and executes them.
`vmem.m`	Assembly stubs for VMM. Calls VSM functions in `buddy.c`
`buddy.c`	Virtual Segment Manager source code.
`cc_home.c`	Home node end of memory-coherence functions.
`cc_request.c`	Requesting node end of memory-coherence functions.
`eh.c`	Event Handler source code - for dealing with the software job queue, as well as responding to block-status miss events.
`lpt.c`	Local Page Table management tasks of the physical memory manager.
`ltlb_body.c`	Core LTLB Miss Handler code written in C.
`pplist.c`	Code to manage free page chains. Written by Andy Shultz from design by the author.
`ppm.c`	Physical Page Manager code for dealing with individual map/unmap/reclaim calls. Written by Andy Shultz from design by the author.
`sq.c`	Event Table code for use in memory-coherence. Currently incorporated directly into the M-Machine simulator and not linked into the runtime executable.
`tmanager.c`	Code for the thread manager dealing mostly with forking, evicting, and installing threads.
`tmanager2.c`	Additional code for the thread manager, dealing mostly with exiting a thread and maintaining parent/child linkages.
`tsignal.c`	Thread manager code dealing with signal/sleep.

Table 10.1: MARS Sources Files

10.3 Future Work

Additional debugging and testing still needs to be performed on the runtime system to iron out bugs, although several test programs which have excercised all aspects of the runtime system, from memory-management to thread creation and communication to memory-coherence, have been successfully executed. These programs include the tfork suites (tfork2.c, tfork3.c, and tfork4.c), the matmul parallel matrix multiply programs (matmul1.c and matmul2.c), and iterative Jacobian relaxation programs (jacoby.c, jacoby2.c, jacoby3.c, jacoby4.c, jacoby5.c, and jacoby6.c).

10.3.1 Loader

The system's loader is an assembly stub which calls into the M-Machine simulator to perform actual program-loading. This component should be implemented as a protected subsystem able to run completely in virtual memory and load other processes without requiring low-level interaction with the runtime system - aside from the I/O aspect of accessing an executable's raw contents, calls to `vmem_alloc` and `tFork` are all that are required.

10.3.2 Memory-Coherence

The memory-coherence data structures and code for manipulating them should be moved out of the simulator and into the runtime system directly. This includes porting the implementations of the `SSQEnqueue`, `SSQDeqeuue`, `SSQGetState`, `SSQSetState`, and other such functions. The work should be relatively simple because the existing implementation is already written in C. The more involved development work must deal with the implementation of the backing-page invalidation and eviction strategy which was presented in the memory-coherence chapter. This will also require that the event handler call upon the physical page manager to determine the number of available backing pages. A low watermark will require preemptive evictions of shared lines to make more pages available should they become necessary. Speed optimizations to improve average-case performance for directory lookups will require modifying the existing memory-coherence directory code to use a chained hash table instead of a simple linked-list of memory-block addresses.

10.3.3 Virtual Memory Management

The deallocation of virtual segments and underlying garbage-collection phase needs to be designed. This involves collecting dirty virtual segments in the dirty buddy list on each node and then performing a garbage-collection phase at very infrequent intervals. The actual garbage-collection will involve several phases. First, an initial round of communication needs to be performed so that all nodes enter into a garbage-collection phase, and prevent user threads from issuing any operations. In addition, all event and message queues need to be drained to remove any latent events and messages which may contain pointers to dirty segments. In a second phase, all local register files and physical memory needs to be examined to look for references to dirty segments. Any pointers which are found need to be replaced (perhaps with *errval* pointers) or NULL pointers). The system must be careful to avoid physical memory used by the OS itself. After local cleanup is completed, references to dirty segments must also be removed from all other nodes on the machine, so the garbage-collector needs to contact all other nodes and ask them to perform a local cleaning. Upon completion of the cleaning phase, another round of communication needs to inform nodes that garbage-collection is complete, and user threads may issue.

10.3.4 UNIX Personality

An entire UNIX system-call layer may be written using the low-level system primitives. This will present a familiar system-level interface for programmers to target without sacrificing general system performance. Thread and process-creation calls would be most interesting to implement in terms of MARS calls. Process creation calls like `fork` and `exec` would require little additional work and may be written in terms of primitives like `tFork`. The `signal` and `waitpid` would perhaps be the most challenging. The UNIX idea of letting programs install system handlers to dispatch on signal events can be extended in the MARS system to allow dispatch threads to run, which absolves the runtime system of needing to save away current program state when handling a signal. Synchronization between the main thread and its signal handlers will need to be designed, however.

In terms of memory-allocation, it is quite likely that the UNIX `sbrk` call may be a NULL call if user threads are given enough virtual address space for code, data, and stack at the outset. Giving threads very large address spaces does not introduce a tremendous inefficiency problem since on-demand backing of virtual pages with physical page frames allows threads to have access to large address spaces without wasting physical memory.

Appendix A

MARS Messages

This appendix chapter lists the messages employed by MARS.

Message IP	Message Words	Description
MSG_ccreturnDirty	address word1 ... word8	Returns the a dirty block named by *address* to the home node. Words 1-8 are the contents of the block.
MSG_ccreturnyankFull	yankbuf word1 ... word8	Returns a dirty block as a response to an invalidation message. The block is named by the address stored at the home node in the *yankbuf*. Words 1-8 are the contents of the block.
MSG_ccreturnyank	yankbuf	Acknowledges an invalidation request with the information that a shared line is no longer at the requesting node. The *yankbuf* sent in the original invalidation message is returned to the home.
MSG_ccinvalidate	address yankbuf	Sends an invalidation for a block identified by *address* to a node which shares that block. The *yankbuf* pointer to a local yankbuffer structure is passed as well. This pointer is returned in the ACK to the invalidation.
MSG_ccNackRO	address header data fcp	Sends a NACK message to a requesting node in response to a request for a readonly copy of a line. The contents of the request message are bounced back to the sender.
MSG_ccNackRW	address header data fcp	Similar to above, except message is in response to a request for an exclusive copy of a line.
MSG_ccreturnLoad	address header word1 ... word8	Sends a readonly copy of a block from a home node to a requesting node. The block starts at *address* and consists of the 8 data words. The *header* sent in the original request is returned as well.
MSG_ccreturnStore	address header word1 ... word8	Same as above, only an exclusive line is returned.

Table A.1: Memory Coherence Messages

Message IP	**Message Words**	**Description**
`MSG_tWake`	`tc signal_data`	Sends a message to invoke the `SYStWake` function on the home node of the context *tc*. The thread identified by *tc* is to be wakened with the *signal_data*.
`MSG_tSleep`	`signal_word tc data_mask`	Invokes a `SYStSleep` function on the home node if *signal_word*, adding a sleeper entry for thread context *tc* with a mask of *data_mask*.
`MSG_tsignal`	`signal_word signal_data`	Invokes a `SYStSignal` function at the home node of *signal_word*.
`MSG_tspawn`	`nargs dp ip arg1 ... arg5`	Spawns a thread executing the function at *ip* with up to *nargs* number of arguments. The thread's data pointer is *dp*.

Table A.2: Thread Management Messages

Appendix B

MARS Header Files

This chapter contains the header files used by the assembly routines and C functions in the M-Machine runtime system.

Appendix C

MARS Assembly Code

This chapter contains the assembly routines for the M-Machine runtime system.

Appendix D

MARS C Code

This chapter contains the C source files for the M-Machine runtime system.

Appendix E

Sample User Programs

This chapter contains the source code for two user-level programs which make calls on MARS primitives. Matmul1.c is a parallel-matrix-multiply program. Jacoby6.c is a version of an iterative jacobian-matrix relaxation.

Bibliography

[1] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 319–327. Association for Computing Machinery Press, October 1994.

[2] Jeffrey S. Chase, Henry M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. How to use a 64-bit virtual address space. Technical Report 92-03-12, University of Washington, 1992.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1993.

[4] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. M-Machine architecture v1.0. Concurrent VLSI Architecture Memo 58, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, January 1994.

[5] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee. The MAP instruction set reference manual v1.3. Concurrent VLSI Architecture Memo 59, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, February 1995.

[6] Abraham Silberschatz, James L. Peterson, , and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, third edition, 1992.

[7] Andrew S. Tannenbaum. *Modern Operating Systems.* Prentice Hall, Englewood Cliffs, NJ, 1992.