Scheduling Support for Concurrency and Parallelism in the Mach Operating System

David L. Black
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
(412) 268-3041
David.Black@cs.cmu.edu

Abstract

Changes in the use of multiprocessors are placing new demands on operating system schedulers. This article describes some of the new challenges posed by parallel and concurrent applications, and introduces techniques developed by the Mach project to meet these challenges. An overview of the techniques of timesharing scheduling and a description of the Mach scheduler are also included.

This article describes work to incorporate processor allocation and control into the Mach operating system. The design approach divides the implementation into three components:

1. Basic mechanisms implemented in the kernel.
2. Long term policy implemented in a server.
3. Optional user implementation of short term policy.

Isolating long-term policy in a server yields the advantages of policy-mechanism separation, while avoiding the drawbacks encountered in previous applications of this principle to multiprocessor scheduling. The design and implementation of a processor allocation server for a gang scheduling policy is also described.

This article also describes work to support the effective multiprogrammed use of multiprocessors. The approach taken to this problem implements the scheduler in the kernel, but encourages users to provide hints. This allows the scheduler to take advantage of user knowledge without requiring users to implement sophisticated scheduling modules.

Keywords: Multiprocessor, Scheduling, Mach, Operating System Multiprogramming, Synchronization, Parallelism, Concurrency, Gang Scheduling, Handoff Scheduling

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251.

1. Introduction

Significant changes occurring in the use of multiprocessors require new support from operating system schedulers. The original use of multiprocessors for increased throughput (more applications run at once, but no individual application runs faster) is giving way to the use of parallel programming for reducing the run time of individual applications. Programming models and languages for parallel programming often anticipate dedicated use of processors or an entire multiprocessor, but few machines are used in this fashion. Most modern general purpose multiprocessors run a timesharing operating system such as Unix[1]. The shared usage model of these systems conflicts with the dedicated use model of many programs. Resolving this clash by restricting usage of a multiprocessor to a single application at a time is often not an attractive solution.

Another impact on schedulers comes from the increasing use of concurrency. We define the *application parallelism* for an application on a multiprocessor as the actual degree of parallel execution achieved. The corresponding definition of *application concurrency* is the maximum degree of parallel execution that could be achieved if an unlimited number of processors were available. For example, an application consisting of ten independent processes running on a six processor multiprocessor has an application parallelism of six based on the six processors, and an application concurrency of ten because it could use up to ten processors. Application concurrency beyond the hardware parallelism can improve hardware utilization by allowing other portions of the application to proceed if one portion blocks for some reason, such as a disk or network operation. The use of concurrency can also simplify programming of concurrent applications by allowing the state of ongoing interactions to be captured in the local variables of executing entities instead of a state table.

Application parallelism and concurrency complicate two areas of scheduling: making effective use of the processing resources available to individual applications, and dividing processing resources among competing applications. The problem of scheduling within applications usually does not arise for serial applications because the schedulable entities that compose an application can often be scheduled independently with little impact on overall performance. In contrast, the medium to fine grain interactions exhibited by programs employing parallelism and concurrency may necessitate scheduling portions of these programs together to achieve acceptable performance. Parallel applications that require a fixed number of processors complicate scheduling across applications by making it more difficult to divide up machine time. These applications may also introduce situations in which fair sharing of the machine may not be an appropriate policy. For example, an application may be configured to require a fixed numbers of processors for efficient execution, and be unable to cope efficiently with less than that number.

This article discusses new approaches to these areas developed as part of the Mach operating system[1] at Carnegie Mellon University. Mach is a new operating system for uniprocessors and multiprocessors that provides flexible memory management and sharing, multiple *threads* (locus of control, program counter, registers) within a single address space or *task* for concurrency and parallelism, and a network transparent communication subsystem (IPC). The Mach kernel incorporates compatibility code derived from 4.3BSD Unix[2] that provides complete binary compatibility. Mach runs on a variety of uniprocessor and multiprocessor architectures including the VAX[2], Sun 3, IBM RP3[3], and Encore Multimax[4].

[1]Unix is a trademark of AT&T Bell Laboratories.

[2]VAX is a trademark of Digital Equipment Corporation

[3]An experimental NUMA multiprocessor

[4]Multimax is a trademark of Encore Computer Corporation

This article concentrates on the shared use of general purpose uniprocessors and shared memory multiprocessors. Support is emphasized for the common Uniform Memory Access (UMA) architectures that have all memory equidistant from all processors in terms of access time. This work is also applicable to Non-Uniform Memory Access (NUMA) machines that have access times to memory depend on the location of the processor with respect to the accessed memory, but it does not provide a complete solution to the load balancing problems for this class of machines.

This article begins with a review of timesharing schedulers and a description of the Mach scheduler as a basis for improvements. This is followed by a short survey of parallel and concurrent programming models. Section 5 discusses the techniques of Handoff and Discouragement Scheduling for support of concurrency. Section 6 discusses the design and implementation of the Mach processor allocation system, and a server that implements a simple but useful allocation policy. Section 7 covers related processor allocation work. Section 8 concludes with a summary of results.

2. Timesharing Scheduling

A major goal of timesharing schedulers is fair sharing of processor resources so that applications competing for processor time will receive approximately equal portions. The notion of 'approximately equal' applies over periods of a few seconds to ensure interactive response in the presence of compute-bound jobs. In practice, this sharing requires some form of processor usage tracking and the use of this information in scheduling decisions. The simplest form of this is a decreasing priority scheduler where the priority of a process decreases continuously as it uses processor time, and the scheduler favors higher priority processes. Multics[3] used such a scheduler and discovered its major disadvantage, that on a heavily loaded system with significant numbers of short-lived jobs the priority of a lengthy job can decrease to a point where little or no further processor time is available to it. The automatic depression of priority for lengthy jobs in some versions of Unix also exhibits this drawback.

To avoid the problem of permanently depressed priorities it is necessary to elevate them in some manner. There are two major choices for elevation methodologies: event-based elevation, and processor usage aging. Event-based elevation is used to deliberately favor interactive response over compute-bound jobs because process priority is elevated by the occurrence of events such as I/O completion. The elevations associated with the events of interest must be determined in some manner, such as tuning under expected workloads to produce desired behavior. This methodology assumes that jobs are either distinctly compute-bound or distinctly interactive and that interactive jobs are more important. Users whose interactive work consumes large amounts of processor time may not do well under this methodology, and it may be necessary to retune the priority elevations in response to workload or hardware changes. The VAX/VMS[5] scheduler employs this elevation methodology[4].

The second priority elevation methodology is processor usage aging. A scheduler that uses this methodology elevates priorities by gradually forgetting about past processor usage, usually in an exponential fashion. An example is that usage from one minute ago is half as costly as usage within the past minute, usage from the minute before is half again as costly, and so on. As a result, the scheduler's measure of processor usage is an exponentially weighted average over the lifetime of a process. A simple exponential average is not desirable because it has the unexpected side effect of raising priorities when the load on the system rises. This happens because under higher loads each process gets a smaller share of the processor, so its usage average drops, causing its priority to rise. These elevated priorities can

[5] VAX/VMS is a trademark of Digital Equipment Corporation.

degrade system response under heavy loads because no process accumulates enough usage to drop its priority. The 4.3BSD version of Unix solves this problem by making the aging rate depend on the load factor, so that aging is slower in the presence of higher load, keeping priorities in approximately the same range [2]. An alternative technique is to use an overload factor to alter the rate at which usage is accumulated. Under this technique the usage accumulated by the scheduler is the actual usage multiplied by a factor that reflects the increased load on the system.

For a multiprocessor scheduler, the concept of fair sharing is strongly intertwined with that of load balancing. The goal of load balancing is to spread the system's computational load evenly among the available processors over time. For example, if three similar jobs are running on a two processor system, each should get two-thirds of a processor on average. This often requires the scheduler to shuffle processes among processors to keep the load balanced. There is an important tradeoff between load balancing and overhead in that optimal load balancing causes high scheduler overhead due to frequent context switches for load balancing. A uniprocessor timesharing scheduler encounters similar issues in minimizing the number of context switches used to implement fair sharing.

3. The Mach Scheduler

This section describes the Mach timesharing scheduler to establish a basis for the improvements to be discussed. Although Mach splits the usual process notion into the task and thread abstractions, the Mach scheduler only schedules threads. The knowledge that two threads are in the same task can be used to optimize the context switch between them. This information is not used to select the threads to run because there may be little improvement in performance depending on the hardware and application involved, and because favoring some threads on this basis can be detrimental to load and usage balancing. Some scheduler functions are driven by periodic clock or timer interrupts that occur on all processors every 10 to 100 milliseconds.

3.1. Data Structures

The primary data structure used by the Mach scheduler is the *run queue*, a priority queue of runnable threads implemented by an array of doubly-linked queues. Thirty-two queues are used in Mach, so four priorities from the Unix range of 0–127 map to each queue[6]. Lower priorities correspond to higher numbers and vice-versa. A hint is maintained that indicates the probable location of the highest priority thread. The highest priority thread cannot be at a higher priority than the hint, but may be at a lower priority. This allows the search for the highest priority thread to start from the hint instead of the highest priority queue, and can avoid searching a dozen or more queues because the highest priority possible for most user threads is 50 to match Unix. Each run queue also contains a mutual exclusion lock and a count of threads currently enqueued. The count is used to optimize testing for emptiness by replacing a scan of the individual queues with a comparison of the counter to zero. This eliminates the need to hold the run queue lock when the run queue is empty, reducing potential contention for this important lock. The use of a single lock assumes that clock interrupts on the processors of a multiprocessor are not synchronized because significant contention can be expected in the synchronized case, and the use of a shared global run queue may not be appropriate. Figure 1 shows an example of a run queue with 3 threads. The queues

[6] More recent Mach kernels use a priority range of 0–31 so that queues and priorities correspond.

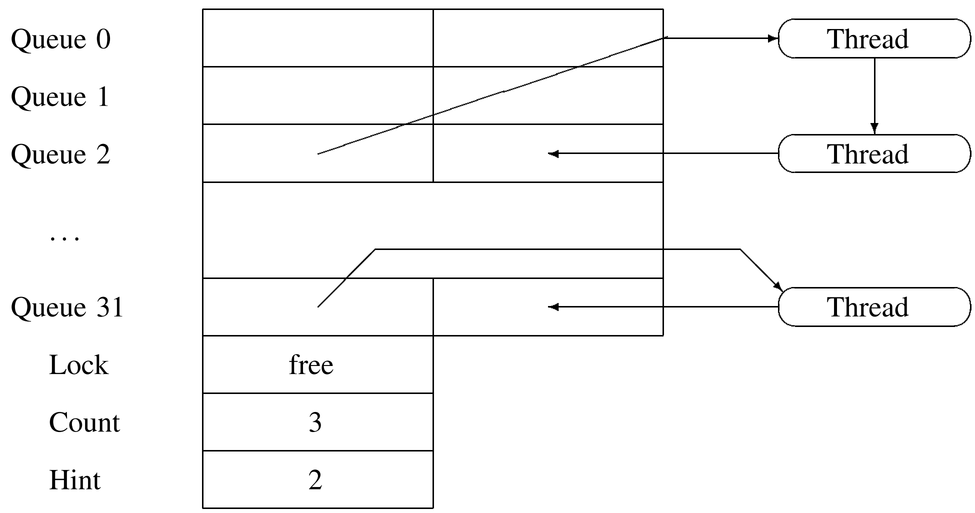

Figure 1: Mach run queue structure

containing the threads are doubly-linked, but only the forward links are shown for clarity. The hint is 2, indicating that queues 0 and 1 are empty and can be skipped in a search for the highest priority thread.

Each processor consults the appropriate run queues when a new thread is needed for execution. The kernel maintains a *local run queue* for each processor and a shared *global run queue*. The local run queue is used by threads that have been temporarily bound to a specific processor for one of two reasons. Some of the Unix compatibility code in current Mach kernels has not been converted for parallel execution[7] so threads executing in the unparallelized portion of this code are temporarily bound to a single designated *master processor*. Interrupts that invoke Unix compatibility code are also restricted to this processor. The remaining use of the local run queues is made by the processor allocation operation described in Section 6. Mach is self scheduling in that processors consult the run queues individually when they need a new thread to run instead of having a centralized dispatcher assign threads to processors. The local run queue is examined first to give absolute preference to bound threads over unbound threads. This avoids bottlenecks by maximizing throughput of the unparallelized Unix compatibility code[8], and improves the performance of processor allocation by causing it to preempt other operations. If the local queue is empty, then the global run queue is examined. In both cases the highest priority thread found is dequeued and run by the processor. If both run queues are empty, the processor becomes idle.

Special mechanisms are used for idle processors. Most uniprocessor systems borrow the kernel stack of the thread or process that ran most recently to execute the idle loop. On a multiprocessor this can be disastrous if the thread resumes execution on another processor because the two processors will corrupt the thread's kernel stack. Mach avoids this stack corruption by using a dedicated *idle thread* for each processor. Idle processors are dispatched by a mechanism that bypasses the run queues. Threads that become runnable are matched with idle processors. The idle thread on each processor picks up the corresponding new thread and context switches directly to it, gaining performance by bypassing the run queues. This mechanism is one example of a general technique called Handoff Scheduling that context switches directly to a new thread without searching for it on a run queue.

---

[7]Most of this code has been parallelized, see [5] for details.

[8]See [6] for an extensive study of this area

3.2. Priority Calculations

Thread priorities consist of a base priority plus an offset derived from recent processor usage. The base priority is set low enough to allow internal kernel threads to run at higher priorities than user threads because the internal threads perform critical kernel functions such as pageout. The offset derived from usage is weighted by the load factor to preserve system responsiveness under load. If an adequate hardware source of timestamps exists, such as a 32-bit microsecond counter, the scheduler can be configured to base usage calculations on timestamps from this counter. This eliminates the inaccuracies and distortions caused by statistical usage calculations (see [6] for details).

Mach uses the overload factor technique for processor usage aging. The aging overhead is distributed by making each thread responsible for aging of its processor usage. Clock interrupts and events that unblock a thread cause it to check its local copy of a counter against a global value that is incremented once a second. If these values differ, the thread ages its usage by a factor of $5/8$ for each unit of difference. This results in each thread's accumulated usage being multiplied by $5/8$ once a second. $5/8$ was chosen to produce behavior similar to other timesharing systems[9], and because multiplication by it can be efficiently without using multiplication or division. Mach uses an overload factor of the number of runnable threads divided by the number of processors with a minimum of 1. This factor is calculated as an exponential average to smooth the impact of abrupt load changes.

This scheduling algorithm requires that all threads check their copy of the counter on a regular basis. The clock interrupt handler performs this check for running threads, and threads that are not running defer this check until they become runnable. Runnable threads with low priorities may spend long periods of time on a run queue while higher priority threads monopolize the processor. Such threads would run if they could check their copy of the counter and age their usage, but their low priority positions on the run queue prevent them from running to perform this check. To avoid the resulting starvation situation an internal kernel thread scans the run queues once every two seconds to perform the aging and resulting priority elevation for any threads in this situation. The two second period was chosen to produce acceptable behavior based on experience with versions of the system that exhibited this starvation behavior.

3.3. Managing Context Switches

The scheduler utilizes time quanta to minimize the number of context switches it causes. When a processor begins to run a thread, that thread is assigned a time quantum. Periodic clock interrupts decrement this quantum, and a rescheduling check occurs when the quantum expires if the thread does not stop or block before then. This rescheduling check causes a context switch if there is another thread to run of equal or higher priority, otherwise the original thread continues running with a new quantum. The context switch mechanism uses an AST (Asynchronous System Trap) to avoid context switches in interrupt service routines. An AST is a general mechanism that allows an interrupt service routine to force the current thread to trap into the kernel from user mode. AST's may be directly supported in hardware, implemented entirely in software, or a combination of the two. Priority reductions due to processor usage accumulated during a thread's first quantum do not cause context switches. This feature reduces the number of context switches for load and usage balancing by extending the time period between context switches.

Context switches are also used to implement preemption of lower priority threads when higher priority threads become runnable. Preemption occurs immediately on a uniprocessor where an AST is requested

[9] The number used by 4.3BSD Unix depends on load. For the range from 0.5 to 1.0, it varies from $1/2$ to $2/3$[2].

Relationship	Model
$MR = VP = PP$	Pure Parallelism
$MR \geq VP = PP$	User Concurrency
$MR = VP \geq PP$	System Concurrency
$MR \geq VP \geq PP$	Dual Concurrency

Table 1: Programming Models for Parallel and Concurrent Programming

as part of making the higher priority thread runnable, but it may take up to a clock interrupt period on a multiprocessor if another processor must notice the presence of a higher priority thread on a run queue. Mach does not use interprocessor interrupts for preemption because they haven't been necessary to achieve acceptable timesharing behavior. A consequence of this is that the scheduler does not maintain data structures that would permit the efficient identification of the lowest priority processor when preemption is needed. The use of interprocessor interrupts for preemption is expected to be added in the future as Mach evolves to include support for some real-time applications.

4. Programming Models

Programming models for applications can introduce concurrency beyond the hardware parallelism at two levels. An operating system can introduce concurrency by providing independently schedulable entities such as Unix processes or Mach threads that it multiplexes onto the hardware processors. This article uses the term *virtual processor* or VP for these entities. A user level library or language runtime may choose to introduce further concurrency by multiplexing language level entities onto these VPs. This article refers to these entities as *multiroutines*; they may be thought of as multiprocessor generalizations of coroutines. Special cases of multiroutines include coroutines (only 1 VP), and the common programming notion of multiple threads (1 VP per multiroutine) as found the Mach Cthreads library[7]. Multiroutines and Virtual Processors can be identified in almost any parallel programming language implementation or environment. One example is that an Ada[10] runtime on Unix would use Unix processes as its virtual processors and Ada tasks as its multiroutines. Another example is that the Mach Cthreads library uses Mach threads as its virtual processors and Cthreads as its multiroutines.

Programming models for parallel and concurrent programming can be classified based on the relationships among the numbers of multiroutines (*MR*), virtual processors (*VP*), and physical processors (*PP*) supported by the model (Table 1). For *pure parallelism* models, the programmer's notion of concurrency is identical to the hardware parallelism. Compiler generated code for parallel execution of loop bodies is a common example. *User concurrency* models introduce additional concurrency at the user level. An example is programs based on a queue of user-defined tasks that are dequeued and executed in parallel by virtual processors. A *coroutine* model is a user concurrency model with exactly one physical processor and hence also one virtual processor. The *system concurrency* models introduce additional concurrency only at the system level. This class of model is used by most multithreading packages and many parallel language runtime systems. Finally, the relatively new class of *dual concurrency* models introduces concurrency at both the system and user levels. The programming model for an application depends both on the programming language or library and how it is used. For example, pure parallelism applications can be written using a library or language with a system concurrency model by only creating as many virtual

[10] Ada is a trademark of the US Government.

processors as physical processors.

Programming models that do not employ system concurrency are required by fine grain applications that execute tens to hundreds of instructions between interactions with other multiroutines. Because virtual and physical processors correspond in these models, every virtual processor is always executing. This assumption is incorporated into the design of synchronization primitives for these models, such as spin locks, and performance suffers if this assumption is violated. Hence programs that use these models require dedicated physical processors. The only exception to this is coroutine models because they use only one physical processor. The major disadvantage of these classes of models is that they handle blocking system operations and page faults inefficiently because blocking a virtual processor in the operating system also blocks a physical processor. This results in wasted time when the physical processor is dedicated to the application.

The remaining classes of models support blocking operations efficiently but are inappropriate for fine grain applications due to potentially large synchronization overheads. Blocking operations are efficient because system concurrency can make an additional virtual processor available to utilize the time relinquished by a blocked virtual processor. The blocking nature of synchronization in languages such as Ada and programming paradigms like message passing forces the use of models from these classes. The need for dedicated processors in these models is application-dependent rather than being inherent in the model. Applications for which parallel execution is important may need dedicated processors, while others may not. Communication or synchronization with virtual processors that are not running can be expensive because the operating system may not understand which virtual processors are involved. Operating system support for such synchronization and communication can improve performance. Even with this support, synchronization can still consume tens to hundreds of instructions, making these models inappropriate for fine grain parallel applications.

5. Scheduling Support for Concurrency

The performance of applications that use more virtual processors than physical processors can benefit from user input to scheduling decisions. Users may have application-specific information about which virtual processors should or should not be running. The Mach scheduler is implemented in the operating system kernel, but allows users to provide hints that influence its decisions. These hints consist of local information involving the thread that provides the hint and possibly one other thread so that users can avoid maintaining information on the overall state of their applications. The hints are based on two pieces of scheduling information that may be available when a thread attempts to communicate or synchronize with another thread that is not running: the first thread may be unable to make further progress until the communication or synchronization is complete, and it may know the identity of the thread it is trying to communicate or synchronize with. For example, in a synchronization based on a message exchange, the initiating thread must block and the identity of the thread that must complete the synchronization is often known.

The Mach scheduler has been enhanced to accept and use hints that provide this information. There are two classes of hints:

1. **Discouragement:** Hint that the current thread should not run. There are three levels:
   (a) Mild: Give up the processor to any other thread if possible.
   (b) Strong: Give up the processor if possible and temporarily depress priority.

(c) Absolute: Block for a specified time period.

2. **Handoff:** Hint that a specific thread should run instead of the current one.

A new primitive, **thread_switch**, allows hints from both classes to be provided simultaneously.

Discouragement hints are useful for optimizing shared memory synchronization in applications that employ system concurrency. The identity of the lock holder is not recorded by the common 'test-and-set' instructions used to implement shared memory locks, making a Handoff hint impossible. A Discouragement hint yields the processor in the hope that the holder of the lock will run. This can cause problems if more than one thread is yielding because they may yield to each other with the result that no useful computation occurs. This can be caused by the timesharing scheduler giving the yielding threads higher usage-based priorities than the thread(s) they are waiting for. Absolute Discouragement prevents this by blocking the threads, but the available time resolution based on clock interrupts is usually too coarse for medium to fine grain synchronization. Strong Discouragement is a compromise that avoids the weaknesses of the other alternatives. A Strong Discouragement hint causes the scheduler to favor threads doing useful work over those that are unable to proceed without the overhead of actually blocking the latter threads. A thread that issues a Strong Discouragement hint may explicitly cancel it when the desired event occurs, otherwise it expires based on a timeout supplied with the hint.

Handoff Scheduling 'hands off' the processor directly to the specified thread, bypassing the internal scheduler mechanisms. Handoff scheduling may designate a thread within the same task or a different task on the same host to run next. A shared memory lock based on a 'compare-and-swap' instruction can identify the target thread, or it may be available from the structure of an application. For example a buffer may be empty and only one thread fills it. One promising use of this technique is to address the problem of 'priority inversion' where a low priority thread holds a lock needed by a high priority thread. The high priority thread can detect this situation and hand off the processor to the low priority thread for the purposes of dropping the lock. When used from outside the kernel, Handoff Scheduling causes the specified thread to be removed from its run queue and run, avoiding a run queue search. Handoff Scheduling is also used extensively by Mach's message passing subsystem inside the kernel to immediately run the recipient of a message. Uses inside the kernel gain additional performance by avoiding the run queue entirely.

5.1. Performance

This section presents results from two experiments that demonstrate the performance benefits of using scheduling hints. These experiments were performed on a Encore Multimax with NS32332 processors, which have a speed of approximately 2 MIPS. The first experiment investigates the use of hints for synchronizing with a thread that is not running. It uses a multithreaded test program than synchronizes with randomly selected threads. A shared variable contains a thread number. That thread replaces it with some other randomly chosen thread number, which replaces it with another randomly chosen thread number, and so on. This program is restricted to run on a single processor, so that it repeatedly synchronizes with a thread that isn't executing. The threads that are not the target of the synchronization can use a scheduling hint to encourage the operating system to run the target.

Table 2 shows the elapsed time per synchronization in microseconds for different scheduling hints and different numbers of threads as mean $\pm$ standard deviation. The Mild Discouragement hint exhibits two different behaviors. If all of the threads are at the same usage-based priority, then the 'Mild-ok' line applies and the synchronization is accomplished in a millisecond or less. If some of the threads are at

Threads	3	4	5	6	7
Mild-ok	570±67	723±125	837±26	920±103	1132±48
Mild-bad	1052±36	1794±512	No data	> 189ms	> 195ms
Strong	708±46	1029±36	1276±45	1587±24	1867±37
Handoff	402±12	406±9	411±4	417±4	429±3

Table 2: Synchronization Experiment Results

different priorities, the 'Mild-bad' line applies, and the synchronization can take much longer. Multiple runs of the program with this hint exhibit both behaviors unpredictably. The only exception is the 5 threads case in which no 'Mild-bad' behavior was observed. . This is probably due to the number of threads being a divisor of the 10Hz clock frequency that drives the scheduler, so that stable behavior is more likely. The times for the 6 and 7 thread cases of 'Mild-bad' ranged from the approximately 200ms shown in the table to over a second. Standard deviations are not meaningful for this data because it is skewed and contains a small number of data points. Absolute Discouragement was not tested because it would result in times on the order of 100ms per synchronization given the clock interrupt frequency of 10Hz.

These results demonstrate the benefits of scheduling hints. Running this program with no scheduling hints yields times of half a second to a second per synchronization, demonstrating the potential performance penalties for ignoring this problem. If the threads are at the same priority then context switching is effective, but poor results occur if this is not the case due to priority inversions. Strong Discouragement has predictable performance, but is slower than the best cases of Mild Discouragement due to the costs of the timeouts associated with the priority depressions. These costs also account for the increasing difference between the 'Strong' and 'Mild-ok' as the number of threads increases. Handoff Scheduling produces the best performance and is significantly faster than sending a message because no time is spent formatting and transporting the message, or blocking to wait for it. These results suggest that effective support of system concurrent applications requires Strong Discouragement support, and that Handoff Scheduling is an effective optimization if the information it requires is available. These are worst case results, but do indicate the relative performance of the hints.

The second experiment concerns the performance benefits of using Handoff Scheduling in the kernel. It uses a message passing exerciser to measure the performance impact of Handoff Scheduling in the Mach IPC system. The experiment involves exchanging messages between two threads in a task on both single and multiple processor configurations. The key difference between the uniprocessor and multiprocessor experiments is the availability of an idle processor to run the recipient thread. Hence the uniprocessor results are also applicable to multiprocessors when no idle processors are available. Table 3 shows the results in μsec of elapsed time per exchange as mean $\pm$ standard deviation. The time differences are statistically significant only for the RPC case without idle processors. The RPC case with idle processors benefits from a handoff in the dispatching code discussed in Section 3.1. This handoff was not disabled for these experiments because it is not specific to the IPC system, and because disabling it requires scheduler modifications that impact the critical context switch path. The performance gain from this handoff can be estimated from the performance gain in the 'no idle processors' case of about 65μsec. Individual messages do not gain performance from Handoff Scheduling for two reasons: in its absence a sender can queue multiple messages before context switching to the receiver, and on a multiprocessor the sender and receiver can run in parallel with complete overlap.

Based on these results, Mach is configured to use Handoff Scheduling for RPC when no idle processor

Messages	RPC		1-way	
Idle processor?	No	Yes	No	Yes
No Handoff	1914±11	1630±6	857±5	432±3
Handoff	1848±8	1628±7	861±10	429±4

Table 3: Message Passing Handoff Results

is available. Current Mach kernels can only handoff once per RPC because the send half of the RPC is implemented separately from the receive. This results in the sender handing off to the receiver before the sender is queued for the reply. When the reply comes back,no thread is queued and no handoff takes place. The Mach IPC system is being redesigned to incorporate a bidirectional message primitive that can handoff in both directions. Similar functionality exists in other systems, such as the Topaz operating system developed at DEC SRC for the Firefly[8].

An alternative approach to this area is to combine the scheduling hints into higher level kernel synchronization primitives, such as semaphores or condition variables. The advantages of this approach are that a cleaner interface can be provided by hiding more scheduler details, and higher level primitives can simplify the implementation of a library or language runtime that uses them. The corresponding disadvantages are that languages and libraries must use these primitives to influence the scheduler, and the primitives may be specialized towards some languages or classes of applications. This can impact performance if the primitives are not a good match to the programming language or model. The Topaz system uses this approach, but specialization is not an issue in that environment because most programming is done in Modula-2+[8].

6. Processor Allocation

Gang scheduling is the common name for a scheduling discipline that guarantees to schedule components of an application simultaneously, and is the primary use for processor allocation in a multiprocessor operating system. Gang scheduling is required by fine grain parallel applications whose performance is severely impacted when any part of the application is not running, but is also applicable to other classes of parallel programs. The need for gang scheduling has been widely recognized, and implementations exist on a variety of multiprocessors. This section describes the design and implementation of Mach's processor allocation facility.

6.1. Design

Flexibility is the driving factor in the design of Mach's processor allocation facility since Mach supports a multitude of applications, languages, and programming models on a variety of multiprocessor architectures. This flexibility has several aspects. It should be possible to allocate processors to applications written in different languages with different programming models. Binding threads to individual processors is not sufficient because applications that use system concurrency need to bind a pool of threads to a pool of processors. This improves their performance by allowing any thread to run on a processor vacated by a blocked thread. The allocation facility should be adaptable to the different multiprocessor architectures that can run Mach, in particular, it should be able to support UMA and NUMA architectures without major changes to the kernel interface. The facility should also accommodate the different policies

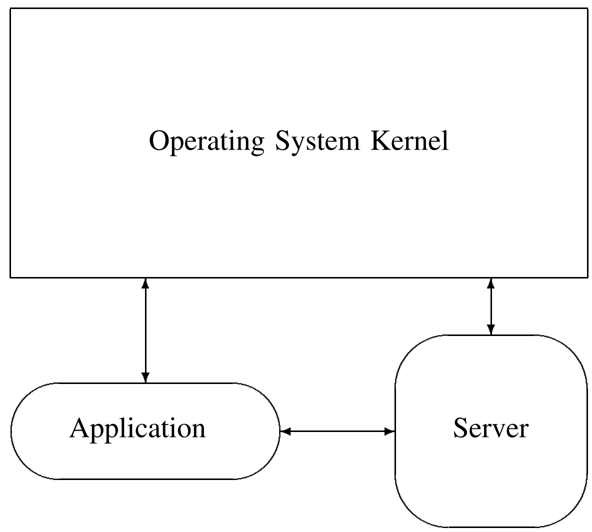

Figure 2: Processor Allocation Components

that are sure to exist at various installations concerning who can allocate how many processors, when, and for how long. Changes to these policies should not require rebuilding the kernel. Finally, it is desirable to offer applications complete control over which threads execute on which processors, but applications that do not want this degree of control should not be forced to implement it.

Mach's processor allocation facility meets these goals by dividing the responsibility for processor allocation among the three components shown in Figure 2:

1. Kernel — Allocation mechanisms only.
2. Server — Implements allocation policy.
3. Application — Requests processors from server and uses them. Can control their use if desired.

The server must be privileged to gain direct control over processors. It is also the primary component affected by changes in usage policies or hardware configuration, and is much easier to replace or reconfigure than the kernel. This design assumes that processors will be dedicated to applications for time periods of seconds or more rather than milliseconds to amortize the overhead of crossing the boundaries among the components in Figure 2 to perform allocation. The application to server interface is not specified because it can be expected to change with both policy and hardware architecture. Some servers may require applications to provide authentication information to establish their right to use certain processors, while other servers may require information describing the location of the requested processors in a NUMA architecture. The kernel interface does not change from machine to machine, but some calls return machine-dependent information.

The allocation facility adds two new objects to the Mach kernel interface, the *processor* and the *processor set*. Processor objects correspond to physical processors, and are used to manipulate them. Processor set objects are independent entities to which threads and processors can be assigned. Processors only execute threads assigned to the same processor set and vice-versa, and every processor and thread is always assigned to a processor set. If a processor set has no assigned processors, then threads assigned to it are suspended. Assignments are initialized by an inheritance mechanism. Each task is also assigned to a processor set, but this assignment is used only to initialize the assignment of threads created in that task. In turn each task's initial assignment is inherited from its parent upon creation, and the first task in the system is initially assigned to the *default processor set*. Thus in the absence of explicit assignments, every thread and task in the system will inherit the first task's assignment to the default processor set. All processors are initially assigned to the default processor set, and at least one processor must always be assigned to it so that internal kernel threads and important daemons can be run. Processors and processor sets are represented by Mach ports. Since access to a port requires possession of a kernel-managed capability for the port, or *port right*, entities other than the appropriate server and/or application cannot interfere with allocations because they do not have the required port right(s). Processor sets also have a name port for identification and status queries, but this port cannot be used to manipulate the processor set.

The allocation and use of dedicated processors involves the application, server and kernel. The division of responsibilities is that the server controls the assignment of processors to processor sets, the application controls the assignment of tasks and threads to processor sets, and the kernel does whatever the application and server ask it to do. The following example describes how an application might allocate six processors for its use:

1. **Application** $\Rightarrow$ **Kernel** Create processor set.
2. **Application** $\Rightarrow$ **Server** Request six processors for processor set.
3. **Application** $\Rightarrow$ **Kernel** Assign threads to processor set.
4. **Server** $\Rightarrow$ **Kernel** Assign processors to processor set.
5. **Application** Use processors.
6. **Application** $\Rightarrow$ **Server** Finished with processors (OPTIONAL).
7. **Server** $\Rightarrow$ **Kernel** Reassign processors.

This example illustrates three important features of the allocation facility. The first is that the application creates the processor set and uses it as the basis of its communication with the server, freeing the server from dependence on the internal structure of the application. The second feature is that only one processor set is used. The scheduling algorithms described earlier function within each processor set, and if the task in this example contains six or fewer threads there will be no context switches to shuffle the threads among the allocated processors. The third feature is that the server does not need the application's cooperation to remove processors from it. The server retains complete control over the processors at all times because it retains the access rights to the processor objects. Removing processors without the application's cooperation should not be necessary for well-behaved applications, but can be useful to remove processors from a runaway application that has exceeded its allocated time.

This design meets its flexibility goals. Different programming models and languages are supported by the ability to assign processors to individual processor sets, or to one or more common sets that match the application's requirements. An assignment of one processor to each of a collection of processor sets provides the application with complete control over which threads run on which processors. Isolating scheduling policy in a server makes it easy to change for different hardware architectures and site-specific usage policies. NUMA machines can use this processor allocation facility to match processor sets to clusters of processors with identical memory access characteristics. This disables the kernel scheduler's load balancing between clusters which is a minimum requirement for scheduling on NUMA machines. The allocation facility does not provide a replacement for the disabled load balancing, but the kernel interface has been designed to make sufficient information available for a user-mode implementation of a NUMA load balancer.

Operation	Time
Create Processor Set	2250±50
Assign Processor	4772±128
Assign thread (self)	1558±25
Assign thread (other)	2624±185

Table 4: Performance of some Allocation Operations

6.2. Implementation

The kernel implementation of processor sets is an extension of the Mach timesharing scheduler described in Section 3. The data structure for each processor set contains a run queue that is used as the global run queue for processors assigned to it. The list of idle processors is also maintained on a per processor set basis because a processor can only be dispatched to threads that are assigned to its current processor set. The processor set data structure is also the head of individual linked lists that are threaded through the data structures of assigned tasks, threads and processors so that these entities can be found and reassigned when the processor set is terminated. In addition, the data structure contains some state information required to run the timesharing scheduling algorithm, the identities of the ports that represent the set, and a mutual exclusion lock to control access to the data structure. The implementation of thread assignment suspends the thread during assignment so that it can remain suspended if assigned to a processor set with no processors. Interprocessor interrupts are used for thread and processor assignment as needed. Table 4 shows the times required by some basic operations in the processor allocation system as mean $\pm$ standard deviation in microseconds. The self and other cases of thread assignment correspond to a thread assigning itself and a thread assigning another thread.

Special techniques are used to manage the processor to processor set assignments. Code in a critical path of the scheduler reads the assignment as part of finding a thread to run on a processor. To optimize this common case against infrequent changes in assignment, each processor is restricted to changing only its own assignment. This avoids the need for a mutual exclusion lock because a processor looking for a new thread cannot be simultaneously changing its assignment. The cost to the assignment operation is that it must temporarily bind a thread to the processor while changing the assignment. An internal kernel thread, the *action thread*, is used for this purpose. Current kernels only use one action thread, but are designed to accomodate more than one. The processor assignment interface allows a server to avoid synchronizing with completion of each assignment so that a server thread can exercise the parallelism available from multiple action threads.

6.3. A Gang Scheduling Server

To demonstrate the utility of this work, a simple processor allocation server for gang scheduling has been implemented. This server is a batch scheduler for processors that schedules requests on a greedy first-come, first-served basis subject to the number of available processors. The server is configured for a maximum allocation of 75% of the machine for at most 15 minutes at a time. The server implementation uses two threads: one to manage processors, the other to communicate with applications. The primary interaction between these threads is by operations on shared data structures describing the requests, but the interaction thread sends a message to the processor thread when an immediate change is needed to the assignment of processors. One such situation is the receipt of an allocation request that can be satisfied immediately. Library routines are available to hide the server interfaces, so an application can make a single call indicating how many processors it wants for how many seconds. This routine contacts the server, arranges the allocation, and returns when the server has begun to assign the requested processors. The total time taken by this routine is about 35ms to allocate one processor plus the processor assignment time of about 5ms per additional processor from Table 4. This overhead is acceptable given the expected allocation durations of tens of seconds to tens of minutes. This server and library interface have been successfully used by researchers and students in an undergraduate parallel programming course at Carnegie Mellon to facilitate performance measurements of parallel programs. The server removed almost all of the administrative difficulties usually involved in obtaining dedicated machine time. In addition, development of the server repeatedly demonstrated the utility of implementing policy in a separate server because server crashes did not crash the operating system.

Many extensions and changes to the policy implemented in the **cpu_server** are possible. Since it is a batch scheduler for processors, techniques originally developed for batch scheduling of memory, such as assigning higher priority to shorter requests, are applicable. In addition, the server could be extended to allow some users higher or absolute priority in allocating processors, or to allow more processors to be allocated during light usage periods. Finally, the server can be replaced in its entirety by a server that implements a different scheduling policy. One promising new policy is to vary the number of processors available to applications based on the overall demand for processors. A server with this policy can notify applications to reconfigure when it changes the number of processors available. Researchers at Stanford are pursuing this approach and have implemented a server for this scheduling policy under Mach with good initial results[9].

7. Related Work

Previous work on *policy mechanism separation* has proposed separating the scheduler into two pieces: mechanisms implemented in the operating system, and policy decisions made by a user mode *policy module*. This work only considered the problem of scheduling within applications, but encountered two problems. The first is the overhead of crossing the boundary between the operating system and an application to access a policy module. This required the operating system to implement short term policy, and made it more efficient to delay long term policy decisions to the detriment of the policy modules. The second is that experience with the resulting systems reveals that most applications did not use the available flexibility. One reason for this is the inherent complexity of the policy modules, as most non-trivial instances require an intricate scheduler implementation[10,11]. Our use of policy mechanism separation avoids these problems because processor allocation decisions are made infrequently enough to effectively amortize the boundary crossing costs, and because the complex policy implementation resides in a server that is implemented once for a system rather than a module that must be customized to each application.

Another body of related work concerns the area of coscheduling. *Coscheduling* is a multiprocessor scheduling policy that attempts to schedule components of an application at the same time, but makes no guarantees about its success in doing so. This policy was originally proposed for medium grain parallel message passing (hundreds to thousands of instructions between interactions) applications that benefit from this coscheduling but can achieve reasonable performance in its absence. The major work on coscheduling was done for the Medusa operating system on Cm*[11], but it is not directly applicable to current multiprocessors because the techniques depend on synchronized clocks and a memory structure that precluded short term load balancing. In contrast, the uniform (UMA) shared memory machines that are our primary interest do need short term load balancing and do not have synchronized clocks.

The Alliant Concentrix[11] scheduler described by Jacobs[12] is an example of an alternative approach to processor allocation and control. This scheduler supports a fixed number of scheduling classes and uses a *scheduling vector* for each processor to indicate which classes should be searched for work in what order. Each processor cycles through a set of scheduling vectors based on time durations associated with each vector, typically fractions of a second. Processes are assigned to scheduling classes by their characteristics or a system call available to privileged users and applications. This scheduler is oriented towards dividing processors among statically defined classes of applications over short periods of time. This contrasts with the Mach orientation of dedicating processors to applications over longer periods of time. Mach's processor sets can be created dynamically as opposed to the fixed number of scheduling classes. Scheduling servers could be implemented by reserving some scheduling classes for their exclusive use, but the static class and vector definitions appear to restrict the flexibility available in forming sets of processors. The Concentrix scheduler also enforces a more restrictive version of gang scheduling in which a blocking operation by any thread blocks the entire gang. This restricts its use to applications that do not use system concurrency and makes parallel handling of blocking operations such as I/O and page faults all but impossible.

8. Conclusion

This article has described problem areas posed by the multiprogrammed use of multiprocessors and techniques used by the Mach operating system to deal with them. In the areas of synchronization and concurrency management, it described the use of Discouragement and Handoff Scheduling techniques to make efficient use of processor time. Handoff Scheduling is a particularly powerful technique, so hardware support such as the atomic 'compare and swap' operation required to use it for shared memory locks should be considered. This paper also described a general approach and architecture for supporting processor allocation and a sample implementation, the **cpu_server**. This work does not solve the problems of multiprocessor scheduling, but rather opens up new research areas by replacing human administrative processor allocation techniques with automatic techniques that can be changed without modifying the operating system kernel.

Acknowledgements

The initial design of the run queue structure is due to Avadis Tevanian, Jr. and is similar to structures used in various versions of Unix. The initial implementation of handoff scheduling for kernel use was done by Richard Rashid. The author would like to thank all of the members of the Mach project for making this research possible. Code to implement the features described in this article is expected to be included in a forthcoming release of Mach. The author would also like to thank the referees and editors for suggestions that greatly improved the structure and presentation of this article.

References

[1] R. F. Rashid, "Threads of a New System," *Unix Review*, vol. 4, pp. 37–49, August 1986.

[11] Concentrix is a trademark of Alliant Computer Systems.

[2] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System*. Reading, MA: Addison-Wesley, 1989.

[3] E. I. Organick, *The Multics System: An Examination of Its Structure*. Cambridge, MA: MIT Press, 1972.

[4] L. J. Kenah and S. F. Bate, *VAX/VMS Internals and Data Structures*. Maynard, MA: Digital Press, 1984.

[5] A. Langerman, J. Boykin, S. LoVerso, and S. Mangalat, "A Highly-Parallelized Mach-based Vnode Filesystem," in *Proceedings of the Winter 1990 USENIX Conference*, January 1990.

[6] J. W. Wendorf, *Operating System/Application Concurrency in Tightly-Coupled Multiple-Processor Systems*. PhD thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, PA, 1987. Available as Technical Report CMU-CS-88-117.

[7] E. C. Cooper and R. P. Draves, "C Threads," Tech. Rep. CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June 1988.

[8] C. P. Thacker, L. C. Stewart, and J. Edwin H. Satterwaithe, "Firefly, A Multiprocessor Workstation," *IEEE Trans. Computers*, vol. 8, pp. 909–920, August 1988.

[9] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[10] W. A. Wulf, R. Levin, and S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System*. New York, NY: McGraw-Hill, 1981.

[11] E. F. Gehringer, D. P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience*. Maynard, MA: Digital Press, 1987.

[12] H. Jacobs, "A User-tunable Multiple Processor Scheduler," in *1986 USENIX Winter Conference Proceedings*, January 1986.