Scheduling Support for Concurrency and Parallelism in the Mach Operating System

David L. Black
Carnegie Mellon University

Significant changes in the use of multiprocessors require new support from operating system schedulers. Originally, multiprocessors increased throughput by running several applications at once, but no individual application ran faster. This use is giving way to parallel programming, which reduces the runtime of individual applications.

Parallel-programming models and languages 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 time-sharing operating system such as Unix. The shared-use model of these systems conflicts with the dedicated-use model of many programs, but the conflict is seldom resolved by restricting multiprocessor use to one application at a time.

Another impact on schedulers comes from the increased use of concurrency. The *application parallelism* for a multiprocessor application is the actual degree of parallel execution achieved, while *application concurrency* is the maximum degree of parallel execution that could be achieved with unlimited processors. For example, an application consisting of 10 independent processes running on a six-processor multiprocessor has an application parallelism of six, based on the six processors, and an application concurrency of 10, because it could use up to 10 processors. Application concurrency beyond hardware parallelism can improve hardware use; if one portion of the application blocks (for example, a disk or network operation), other portions can still proceed. The use of concurrency can also simplify programming of concurrent applications by capturing the state of ongoing interactions in local variables of executing entities instead of in a state table.

Traditional time-sharing schedulers are inadequate for parallel and concurrent programs, which require new techniques such as processor allocation and handoff scheduling.

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 seldom arises for serial applications because they can often be scheduled independently with little impact on overall performance. In contrast, the medium- to fine-grain interactions of parallel and concurrent programs may require scheduling portions of these programs together to achieve acceptable performance. Parallel applications that require a fixed number of processors complicate scheduling across applications. They make division of machine time more difficult and may introduce situations for which a fair-sharing policy is inappropriate. For example, an application configured for a fixed numbers of processors may be unable to cope efficiently with fewer processors.

At Carnegie Mellon University, developing the Mach operating system[1] for uniprocessors and multiprocessors produced some new approaches to scheduling. Mach provides flexible memory manage-

0018-9162/90/0500-0035$01.00 © 1990 IEEE

ment and sharing, multiple *threads* (control locus, program counter, registers) within a single address space or *task* for concurrency and parallelism, and a network-transparent communication subsystem. This communication subsystem is called the IPC (interprocess communication) subsystem for historical reasons; all communication in Mach is actually between tasks. 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 DEC VAX, Sun 3, IBM RP3, and Encore Multimax. Mach is available and supported as a product by a number of hardware vendors, including Next, Encore, and Omron. It is also the base technology for the OSF/1 operating system from the Open Software Foundation.

This article concentrates on the shared use of general-purpose uniprocessors and shared-memory multiprocessors, emphasizing support for common uniform-memory-access architectures that have all memory equidistant from all processors in terms of access time. This work is also applicable to nonuniform-memory-access machines, whose memory access times depend on the physical distance between the processor and the accessed memory, but it does not provide a complete solution to load-balancing problems for this class of machine.

Time-sharing scheduling

A major goal of time-sharing schedulers is allocating resources so that competing applications receive approximately equal portions of processor time. The "approximately equal" notion applies over periods of a few seconds to ensure interactive response in the presence of computation-bound jobs. In practice, this requires tracking processor usage and using the information in scheduling decisions. The simplest use, a decreasing-priority scheduler, continuously decreases a process' priority as it uses processor time and favors higher priority processes. Multics[3] used such a scheduler and discovered its major disadvantage: on a heavily loaded system with many short-lived jobs, the priority of a lengthy job can decrease until little or no further processor time is available to it. The automatic priority depression of 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. The two major elevation methodologies are event-based elevation and processor usage aging. Event-based elevation deliberately favors interactive response over computation-bound jobs. It elevates process priority through such events such as I/O completion. Elevations associated with events of interest must be determined in some manner, such as tuning under expected workloads to produce desired behavior. This methodology, used by VAX/VMS,[4] assumes that jobs are either distinctly computation-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. Such applications may not generate enough priority-raising events to offset the priority decreases caused by processor usage. Also, under this methodology, it may be necessary to retune priority elevations in response to workload or hardware changes.

Under the second priority-elevation methodology, processor usage aging, a scheduler elevates priorities by gradually forgetting about past processor usage, usually in an exponential fashion. For example, usage from one minute ago is half as costly as use within the past minute, usage from two minutes ago is half again as costly, and so on. As a result, the scheduler's measure of usage is an exponentially weighted average over the lifetime of a process. A simple exponential average is not desirable; it has the unexpected side effect of raising priorities when system load 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 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. Aging is slower in the presence of a higher load, which keeps priorities in approximately the same range.[2] An alternative technique uses an overload factor to alter the rate at which usage accumulates. Under this technique, the usage accumulated by the scheduler is the actual usage multiplied by a factor based on the system load.

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 average two-thirds of a processor. This often requires the scheduler to shuffle processes among processors to keep the load balanced. An important trade-off between load balancing and overhead is that optimal load balancing causes high scheduler overhead due to frequent context switches for load balancing. A uniprocessor time-sharing scheduler encounters similar issues in minimizing the number of context switches used to implement fair sharing.

Mach scheduler

The Mach operating system splits the usual process notion into task and thread abstractions, but the Mach time-sharing scheduler only schedules threads. The knowledge that two threads occur in the same task can be used to optimize the context switch between them but is not used to select which threads run. Favoring threads on this basis may not improve performance, depending on the hardware and application involved. It can also be detrimental to usage and load balancing because this favoritism may conflict with decisions needed to accomplish balancing.

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. Mach uses 32 queues, so four priorities from the Unix range of 0 to 127 map to each queue. (More recent Mach kernels use a priority range of 0 to 31 so that queues and priorities correspond.) 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 it may be at a lower priority. This allows the search for the highest priority thread to start from the hint, potentially avoiding the search of a dozen or more queues because the highest possible priority 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 optimizes 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, thereby 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. Significant contention can be expected in the synchronized case; thus, a shared global run queue may not be appropriate.

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

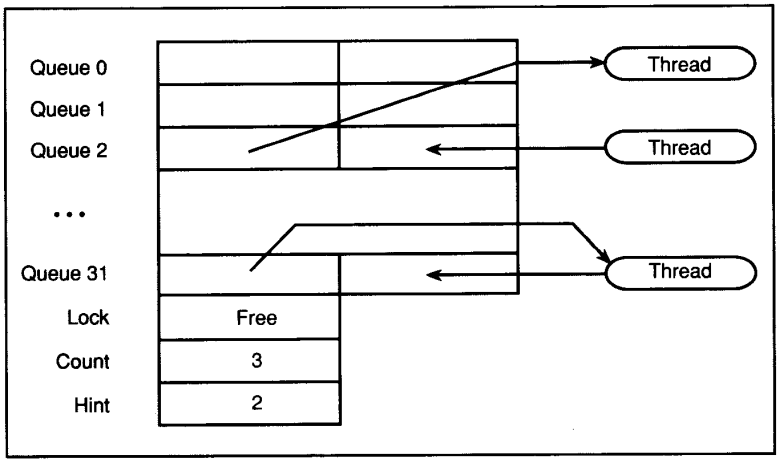

Figure 1. Mach run-queue structure.

When a new thread is needed for execution, each processor consults the appropriate run queues. 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: (1) Although most of the Unix compatibility code in current Mach kernels has been parallelized,[5] threads executing in the unparallelized portion are temporarily bound to a single, designated *master processor*; (2) interrupts that invoke Unix compatibility code are also restricted to this processor. The remaining use of the local run queues is by the processor allocation operation described in the section entitled "Processor allocation."

Mach is self-scheduling in that, instead of having threads assigned by a centralized dispatcher, individual processors consult the run queues when they need a new thread to run. A processor examines the local run queue first to give bound threads absolute preference over unbound threads. This precaution avoids bottlenecks by maximizing throughput of the unparallelized Unix compatibility code[6] and improves processor allocation performance by preempting other operations. If the local queue is empty, the processor examines the global run queue. In either case, it dequeues and runs the highest priority thread. If both queues are empty, the processor becomes idle.

Special mechanisms are used for idle processors. Most uniprocessor systems execute the idle loop by borrowing the kernel stack of the most recently run thread or process. On a multiprocessor this can be disastrous. If the thread resumes execution on another processor, the two processors will corrupt the thread's kernel stack. To avoid stack corruption, Mach uses 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.

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, which perform critical kernel functions such as pageout, to run at higher priorities than user threads. The offset derived from usage is weighted by a load factor to preserve system responsiveness under load. If an adequate hardware source of time stamps exists, such as a 32-bit microsecond counter, the scheduler can be configured to base usage calculations on time stamps from this counter. This configuration eliminates the inaccuracies and distortions caused by statistical usage calculations (see Wendorf[6] for details).

Mach uses the overload factor technique for processor usage aging. Aging overhead is distributed by making each thread responsible for aging 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. This figure was chosen for efficiency (two shifts and an add can implement multiplication by 5/8) and to produce behavior similar to other time-sharing systems. (The number used by 4.3BSD Unix depends on load; it varies from 1/2 to 2/3 for the 0.5 to 1.0 range.) The Mach scheduler uses a load factor of the number of runnable threads divided by the number of processors with a minimum factor of 1. This factor is calculated as an exponential average to smooth the impact of abrupt load changes.

The scheduling algorithm requires all threads to check their copy of the counter on a regular basis. The clock interrupt handler performs the check for running threads, and threads that are not running defer the check until they become runnable. Runnable threads with low priorities may spend long periods 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 prevent them from running to perform the check. To avoid such "starvation," an internal kernel thread scans the run queues every two seconds to age and elevate any threads caught in that situation. The two-second period, selected on the basis of experience with versions that exhibited starvation, produces acceptable behavior.

Managing context switches. The scheduler uses time quanta to minimize the number of context switches it causes. When a processor begins running a thread, that thread is assigned a time quantum. Periodic clock interrupts (every 10 to 100 milliseconds) decrement this quantum, and a rescheduling check occurs when the quantum expires if the thread is still running. This rescheduling check causes a context switch if there is another runnable thread of equal or higher priority; otherwise, the original thread continues running with a new quantum. The context switch mechanism uses an asynchronous system trap (AST) 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. ASTs 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 also preempt lower priority threads when higher priority threads become runnable. On a uniprocessor, where an AST is requested as part of making the higher priority thread runnable, preemption occurs immediately. On a multiprocessor, if another processor must notice the presence of a higher priority thread on a run queue, preemption can take up to one clock-interrupt period. Mach doesn't use interprocessor interrupts for preemption; they haven't been necessary to achieve acceptable time-sharing behavior. Consequently, the scheduler doesn't maintain data structures that permit efficient identification of the lowest priority processor. However, as Mach evolves to support real-time applications, the addition of interprocessor interrupts for preemption is likely.

Programming models

Application programming models can introduce concurrency beyond hardware parallelism at two levels. An operating system can introduce concurrency by multiplexing Unix processes, Mach threads, or other independently schedulable entities (called *virtual processors*) onto hardware processors. A user-level library or language runtime can introduce further concurrency by multiplexing language-level entities onto the VPs. Called *multiroutines*, these entities can be thought of as multiprocessor generalizations of coroutines. Special cases of multiroutines include coroutines (only one virtual processor) and the common programming notion of multiple threads (one virtual processor per multiroutine) found in many systems, including the Mach Cthreads library.[7] Multiroutines and virtual processors can be identified in almost any parallel programming language implementation or environment. One example is an Ada runtime on Unix, which would use Unix processes as its virtual processors and Ada tasks as its multiroutines. The Mach Cthreads library uses Mach threads as its virtual processors and Cthreads as its multiroutines.

Table 1. Programming models for parallel and concurrent programming.

Relationship	Model
MR = VP = PP	Pure parallelism
MR ≥ VP = PP	User concurrency
MR = VP ≥ PP	System concurrency
MR ≥ VP ≥ PP	Dual concurrency

Parallel and concurrent programming models can be classified by the relationships among the multiroutines (MRs), virtual processors (VPs), and physical processors (PPs) they support (see 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 would be programs based on a queue of user-defined tasks dequeued and executed in parallel by virtual processors. A *coroutine* model is a user concurrency model with exactly one physical processor and hence one virtual processor. *System concurrency* models, used by most multithreading packages and many parallel language runtime systems, introduce additional concurrency only at the system level. *Dual concurrency* models, a relatively new class, introduce concurrency at both the system and user levels. The programming model for an application depends on both the programming language or library and how it is used. For example, pure parallelism applications can be written with a system concurrency library or language by creating only as many virtual processors as physical processors.

Fine-grain applications, which execute tens to hundreds of instructions between interactions with other multiroutines, cannot use system concurrency models. They require models with corresponding virtual and physical processors in which every virtual processor is always executing. This assumption is designed into the models' synchronization primitives, such as spin locks, and performance suffers if it is violated. Hence, programs using these models require dedicated physical processors. (Coroutine models, which use only one physical processor, are an exception.) The major disadvantage of these models is inefficient implementation of blocking system operations including page faults. Blocking a virtual processor in the operating system also blocks a physical processor, which wastes time when the physical processor is dedicated to the application.

The remaining model classes support blocking operations efficiently, but their potentially large synchronization overheads make them inappropriate for fine-grain applications. Their blocking operations are efficient because system concurrency can make an additional virtual processor available to use 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 inherent to the model. Applications for which parallel execution is important may need dedicated processors, while others may not. Communication or synchronization with nonexecuting virtual processors can be expensive because the operating system might not understand which processors are involved. Operating system support for such synchronization and communication can improve performance. But, even with this support, synchronization can consume hundreds of instructions, making these models inappropriate for fine-grain parallel applications.

Scheduling concurrency support

Applications using more virtual 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 accepts user hints. These hints consist of local information involving the hint's thread, and possibly one other, so that users can avoid maintaining overall status information on 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 that will complete the operation. For example, in a synchronization based on a message exchange, the initiating thread must block, and the identity of the thread that will reply to the message is often known.

The Mach scheduler accepts and uses two classes of hints: discouragement and handoff. A new primitive, thread_switch, allows simultaneous hints from both classes.

- *Discouragement hints*, which can be mild, strong, or absolute, indicate that the current thread should not run. Mild hints suggest giving up the processor to any other thread if possible. Strong hints go one step further and temporarily depress priority. Absolute hints block for a specified period.
- *Handoff hints* indicate that a specific thread should run instead of the current one.

Discouragement hints are useful for optimizing shared-memory synchronization in applications employing system concurrency. The lock holder's identity is not recorded by the common test-and-set instructions used to implement shared memory locks, making a handoff hint impossible. A mild discouragement hint yields the processor in the hope that the lock's holder will run. This can cause problems if more than one thread is yielding. They may yield to each other, with the result that no useful computation occurs. This situation can occur if the time-sharing scheduler gives the yielding threads higher usage-based priorities than the thread(s) they are waiting for. Absolute discouragement prevents this problem 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. Strong discouragement causes the scheduler to favor threads doing useful work, without the overhead of actually blocking those unable to proceed. A thread that issues a strong discouragement hint may explicitly cancel it when the desired event occurs; otherwise, the hint expires at a timeout supplied with the hint.

The second class of hint, handoff scheduling, directly "hands off" the processor 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 its identity may be available from the application's structure; for example, if a buffer is empty and only one thread fills it, then that thread should be run. One promising use of this technique addresses the "priority inversion" problem, where a low-priority thread holds a lock needed by a high-priority thread. The high-priority thread can detect this situation and hand the processor to the low-priority thread. When used from outside the kernel, handoff scheduling removes the specified thread from its run queue and runs it, avoiding a run queue search. Mach's message-passing subsystem also uses handoff scheduling inside the kernel to immediately run the recipient of a message. Use inside the kernel aids performance by avoiding the run queue entirely.

Performance. Two experiments — performed on an Encore Multimax with NS32332 processors having a speed of approximately 2 MIPS — have shown that scheduling hints benefit performance.

Synchronization hints. The first experiment investigated the use of hints for synchronizing with a thread that is not running. It used a multithreaded test program that synchronized with randomly selected threads. A shared variable contained a thread number. That thread replaced it with another randomly chosen thread number, which then replaced it with another randomly chosen thread number, and so on. This program was restricted to a single processor, so it repeatedly synchronized with a nonexecuting thread. Threads not targeted for synchronization could use a scheduling hint to encourage the operating system to run the target.

Table 2. Synchronization experiment results (in microseconds).

Threads	3	4	5	6	7
Mild-ok	467 ± 3	633 ± 3	817 ± 6	973 ± 12	1,160 ± 19
Mild-bad	578 - 1,224	1,128 - 4,427	3.9 - 7.5 ms	8 - 607 ms	138 - 487 ms
Strong	897 ± 6	1,215 ± 9	1,434 ± 11	1,825 ± 10	2,130 ± 18
Handoff	413 ± 3	418 ± 5	417 ± 3	421 ± 4	428 ± 4

Table 2 shows the elapsed time per synchronization in microseconds for different scheduling hints and numbers of threads as mean ± standard deviation. The mild discouragement hint exhibits two different behaviors. If all threads are at the same usage-based priority, the mild-ok line applies, and synchronization occurs quickly. If some of the threads are at different priorities, the mild-bad line applies, and the time per synchronization not only increases but exhibits pronounced variability. The mild-bad data is shown as ranges because the distributions are skewed toward higher frequencies at lower values. Multiple runs of the same test exhibit both behaviors unpredictably. This is strong evidence that, in many cases, mild discouragement is is not an appropriate scheduling hint. Strong discouragement is an appropriate alternative. Its behavior is far more stable, and its performance is better than most bad cases of mild discouragement. The synshronization times from the five-thread test cases for all hints is slightly better than expected, based on the other results. This is probably because the number of threads is a divisor of the 10-Hertz clock frequency that drives the scheduler, so stable behavior is more likely. Absolute discouragement was not tested because it would result in times on the order of 100 milliseconds per synchronization, given the hardware clock-interrupt frequency of 10 Hertz.

The results show the benefits of scheduling hints. Running the program without scheduling hints yields times of one-half to one second per synchronization, demonstrating the potential performance pen-

Table 3. Message-passing handoff results (in microseconds).

Messages	Remote Procedure Call		One-way	
Idle processor?	No	Yes	No	Yes
No Handoff	1,914 ± 11	1,630 ± 6	857 ± 5	432 ± 3
Handoff	1,848 ± 8	1,628 ± 7	861 ± 10	429 ± 4

alties for ignoring the problem. If the threads are at the same priority, context switching is effective; however, if that is not the case, priority inversions cause poor results. Strong discouragement performs predictably but is slower than the best cases of mild discouragement because of the timeout costs associated with priority depressions. These costs also account for the increasing difference between the strong and mild-ok results as the number of threads increases. Handoff scheduling performs the best and is significantly faster than sending a message, since no time is spent formatting and transporting the message or blocking to wait for it. These results suggest that system-concurrent applications require strong-discouragement support and that handoff scheduling is effective if the required information is available. These are worst-case results, but they do indicate the relative performance of the hints.

IPC handoff scheduling. The second experiment concerned the performance benefits of using handoff scheduling in the kernel. It used a message-passing exerciser to measure the performance impact of handoff scheduling in Mach's network-transparent communication subsystem, the IPC. The experiment involved 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. The experiment was run for both unidirectional and bidirectional (remote procedure call (RPC)) message exchanges.

Table 3 shows the results in microseconds of elapsed time per exchange as mean ± standard deviation. The time differences are statistically significant only for RPCs without idle processors. The RPC case with idle processors benefits from a handoff in the dispatching code (see "Mach Scheduler, Data Structures," above). This handoff was not disabled for these experiments. It is not specific to the IPC system, and disabling it requires scheduler modifications that impact the critical-context switch path. One-way message exchanges do not gain performance from handoff scheduling for two reasons. First, in the absence of handoff scheduling, a sender can queue multiple messages before context switching to the receiver. Second, 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 is available. Current Mach kernels can hand off only once per RPC because the RPC send half is implemented separately. Thus, the sender hands 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 hand off in both directions. Similar functionality exists in other systems, such as the Topaz operating system developed at DEC's Systems Research Center for the Firefly.[8]

Alternatively, scheduling hints could be combined into higher level kernel synchronization primitives, such as semaphores or condition variables. Higher level primitives can provide a cleaner interface, by hiding more scheduler details, and can simplify the implementation of a library or language runtime that uses them. The disadvantages are that languages and libraries must use these primitives to influence the scheduler, and the primitives may be specialized toward some languages or classes of applications. This specialization 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]

Processor allocation

Gang scheduling, which guarantees simultaneous scheduling of an application's components, is one use for processor allocation in multiprocessor operating systems. It is necessary for fine-grain parallel applications whose performance severely degrades when any part of the application is not running, but it is also applicable to other classes of parallel programs. The need for gang scheduling is widely recognized, and implementations exist on a variety of multiprocessors. This section describes the design and implementation of Mach's processor allocation facility.

Design. Because Mach supports a multitude of applications, languages, and programming models on a variety of multiprocessor architectures, flexibility is the driving factor in the design of its processor allocation facility. This flexibility has several aspects.

- The facility should be capable of allocating 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. Such *pool binding* improves performance by allowing any thread to run on a processor vacated by a blocked thread.
- The facility should be adaptable to the different multiprocessor architectures that can run Mach. In particular, it should support uniform-memory access (UMA) and nonuniform-memory-access (NUMA) architectures without major changes to the kernel interface.
- The facility should accommodate different policies — 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, the facility should offer applications complete control over which threads execute on which processors, but it should not force implementation on applications not wanting this degree of control.

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

- kernel, performs allocation mechanisms only;

- server, implements allocation policy;
- 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. As the component most affected by changes in usage policies or hardware configuration, the server is designed to be much easier to replace or reconfigure than the kernel. The design assumes that processors will be dedicated to applications for seconds or longer, rather than milliseconds, to amortize the overhead of crossing boundaries among components. The application-to-server interface is not specified because it will be affected by changes in usage 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 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 and manipulate physical processors. 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 inherits its initial assignment 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 inherits 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. Because access to a port requires a kernel-managed capability for that port, or *port right*, entities other than the appropriate server and/or application lack the required port right(s) and cannot interfere with allocations. Processor sets also have a name port for identification and status queries, but this port cannot be used to manipulate the processor set.

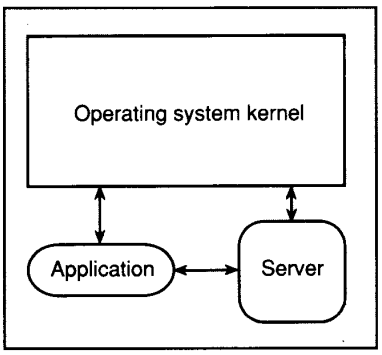

Figure 2. Processor allocation components.

Responsibility for the allocation and use of dedicated processors is divided among the application, server, and kernel. The application controls the assignment of tasks and threads to processor sets. The server controls the assignment of processors to processor sets. The kernel does whatever the application and server ask it to do.

Here's how an application might allocate six processors for its use:

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

This example illustrates three important features of the allocation facility.

- 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.
- 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 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 it can be useful for a runaway application that has exceeded its allocated time.

This design meets its flexibility goals. It supports different programming models and languages; it can assign processors to individual processor sets or to one or more common sets that match application requirements. Assigning one processor to each of several processor sets gives the application complete control over which threads run on which processors.

Isolating scheduling policy in a server simplifies changes for different hardware architectures and site-specific usage policies. NUMA machines can use this 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 facility does not replace the disabled load balancing, but, by design, the kernel interface makes sufficient information available for a user-mode implementation of a NUMA load balancer.

Implementation. Kernel implementation of processor sets extends the Mach time-sharing scheduler. The data structure for each processor set contains a run queue that is used as the global run queue for processors assigned to the set. A list of idle processors is also maintained on a per-processor-set basis because a processor can only run threads assigned to its current set. The processor-set data structure also heads 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 state information required to run the time-sharing scheduling algorithm, the identities of ports that represent the set, and a mutual-exclusion lock to control access to the data structure. Thread assignment suspends the thread involved 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 times required by basic operations in the processor allocation system as mean ± 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 manage processor-to-processor set relationships. Code in a critical scheduler path reads a processor's assignment as part of finding a thread to run on that processor. To optimize this common case against infrequent assignment changes, each processor can change only its own assignment. This restriction avoids the need for a mutual-exclusion lock because a processor cannot look for a new thread and change its assignment simultaneously. 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*, serves this purpose. Current kernels use only one action thread but are designed to accommodate more. The processor assignment interface lets a server avoid synchronizing with each assignment's completion, so a server thread can exercise the parallelism available from multiple action threads.

Gang-scheduling server. To demonstrate the utility of this work, we implemented a simple processor-allocation server for gang scheduling. The 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 percent of the machine for, at most, 15 minutes at a time.

The server implementation uses two threads: one to manage processors and another to communicate with applications. The primary interaction between the threads is via operations on shared data structures describing the requests, but the interaction thread sends a message to the processor thread when an immediate change in processor assignment is needed. 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 assigning the requested processors. The routine takes about 35 milliseconds to allocate one processor plus about 5 milliseconds per additional processor. This overhead is acceptable, given expected allocations of tens of seconds to tens of minutes.

Table 4. Allocation operation performance.

Operation	Time (μs)
Create processor set	2,250 ± 50
Assign processor	4,772 ± 28
Assign thread (self)	1,558 ± 25
Assign thread (other)	2,624 ± 185

At Carnegie Mellon University, researchers and students in an undergraduate parallel programming course have successfully used this server and library interface in measuring the performance of parallel programs. The server removed most of the usual administrative obstacles to obtaining dedicated machine time. In addition, it demonstrated the utility of implementing policy in a separate server; 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, 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 allocation of more processors during light usage periods. Finally, the server could be entirely replaced by a server that implements a different scheduling policy. One promising new policy is to vary the number of processors available to applications according to overall demand. A server with this policy would tell applications to reconfigure when it changes the number of processors available. Researchers at Stanford are pursuing this approach and have implemented such a server under Mach with good initial results.[9]

Related work

Previous work[10,11] on *policy mechanism separation* proposed separating the scheduler into two pieces, with mechanisms implemented in the operating system and policy decisions made by a user-mode *policy module*. This work, which considered only the problem of scheduling within applications, encountered two problems.

The first problem was the overhead of crossing the boundary between the operating system and an application to access a policy module. Crossing the boundary required operating-system implementation of short-term policy, which, to the detriment of the policy modules, made it more efficient to delay long-term policy decisions.

The second difficulty, revealed through experience with the resulting systems, was that most applications did not use the available flexibility. One reason for this lies in the inherent complexity of policy modules; most nontrivial instances require an intricate scheduler implementation.[10,11]

Our use of policy/mechanism separation avoids both problems. Processor allocation decisions are infrequent enough to effectively amortize boundary crossing costs, and the complex policy implementation resides in a server that is implemented once for a system rather than in a module that must be customized to each application.

Another body of related work concerns coscheduling, a multiprocessor scheduling policy that attempts simultaneous scheduling of an application's components but makes no guarantee of success. This policy was originally proposed for medium-grain, parallel message-passing applications with hundreds to thousands of instructions between interactions. Such applications benefit from coscheduling but achieve reasonable performance without it.

The major work on coscheduling was done for the Medusa operating system on Cm*.[11] It is not directly applicable to current multiprocessors because the techniques depend on synchronized clocks and a memory structure that precludes short-term load balancing. In contrast, the uniform shared-memory machines that are our primary interest do need short-term load balancing and do not have synchronized clocks.

The Alliant Concentrix scheduler, described by Jacobs,[12] is 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 toward dividing processors among statically defined classes of applications over short periods of time. This contrasts with Mach's orientation of dedicating processors to applications over longer periods of time. Mach's processor sets can be created dynamically in contrast to Concentrix's 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 processor sets. 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 limits it to applications not using system concurrency and makes parallel handling of blocking operations, such as I/O and page faults, all but impossible.

Many parallel and concurrent applications cannot be scheduled acceptably by traditional time-sharing means. Dedicated processors are required to obtain acceptable performance from some parallel applications. For concurrent applications, communication and synchronization performance can be improved by taking advantage of application-specific scheduling information. Mach's scheduler has been enhanced to meet these challenges.

Mach allows concurrent programs to provide handoff and discouragement hints to influence scheduling decisions. Of these two, handoff hints are more effective; naming the next thread to run bypasses much of the scheduler logic that normally makes this decision. These hints are based on local information that is easy to obtain and provide to the scheduler. Using local information and hints avoids the overhead and complexity drawbacks of previous work based on application-specific policy modules.

Obtaining and scheduling dedicated processors is supported by Mach's processor allocation and control facilities. The Mach kernel implements only allocation mechanisms; policy decisions are made by a privileged server that is much easier to reconfigure or replace than the kernel. This provides a wide degree of flexibility to implement allocation policies and accommodate different multiprocessor architectures. This design also accommodates applications based on different programming models.

Code to implement the Mach features described in this article will be available from several sources. Future Mach releases from Carnegie Mellon University and Mt. Xinu are expected to support these features, but the current 2.5 and 2.6 MSD releases do not. These features are a part of the OSF/1 operating system from the Open Software Foundation and are or will be available in some vendor versions of Mach (for example, Encore's Mach for the Multimax). ■

Acknowledgments

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 members of the Mach project for making this research possible and *Computer*'s referees and editors for suggestions that greatly improved the structure and presentation of this article.

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. However, the views and conclusions contained in this article are those of the author and should not be interpreted as representing official policies, either expressed or implied, of DARPA or the US government.

References

1. R.F. Rashid, "Threads of a New System," *Unix Review*, Vol. 4, Aug. 1986, pp. 37-49.
2. S.J. Leffler et al., *Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley, Reading, Mass., 1989.
3. E.I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass., 1972.
4. L.J. Kenah and S.F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, Maynard, Mass., 1984.
5. A. Langerman et al., "A Highly-Parallelized Mach-based Vnode Filesystem," *Proc. Winter 1990 Usenix Conf.*, Jan. 1990, pp 297-312.
6. J.W. Wendorf, "Operating System/Application Concurrency in Tightly Coupled Multiple-Processor Systems," PhD thesis, Tech. Rep. CMU-CS-88-117, Carnegie Mellon University, Computer Science Dept., Pittsburgh, 1987.
7. E.C. Cooper and R.P. Draves, "C Threads," Tech. Rep. CMU-CS-88-154, Carnegie Mellon University, Computer Science Dept., Pittsburgh, 1988.
8. C.P. Thacker, L.C. Stewart, and J.E.H. Satterwaithe, "Firefly, A Multiprocessor Workstation," *IEEE Trans. Computers*, Vol. 37, No. 8, Aug. 1988, pp. 909-920.
9. A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors," *Proc. 12th ACM Symp. Operating Systems Principles*, Dec. 1989, pp. 159-166.
10. W.A. Wulf, R. Levin, and S.P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.
11. E.F. Gehringer, D.P. Siewiorek, and Z. Segall, *Parallel Processing: The Cm* Experience*, Digital Press, Maynard, Mass., 1987.
12. H. Jacobs, "A User-Tunable Multiple Processor Scheduler," *Proc. Winter 1986 Usenix Conf.*, Jan. 1986, pp. 183-191.

David L. Black is a graduate student and PhD candidate in the School of Computer Science at Carnegie Mellon University, where he has worked on the design and implementation of the Mach operating system since 1986. His research interests are in operating systems, with emphasis on multiprocessor and scheduling issues.

Black received BA and MA degrees in mathematics and the BSE degree in computer science and engineering from the University of Pennsylvania in 1983. He received the MS degree in computer science from Carnegie Mellon University in 1985. He is a member of the ACM, IEEE, the Computer Society, and several honorary societies.

Readers may contact Black at Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213 or, by e-mail, at david.black@cs.cmu.edu.