The following paper was originally published in the
Proceedings of the USENIX 2nd Symposium on
Operating Systems Design and Implementation
Seattle, Washington, October 1996

CPU Inheritance Scheduling

Bryan Ford and Sai Susarla
University of Utah

CPU Inheritance Scheduling

Bryan Ford Sai Susarla

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

flux@cs.utah.edu

http://www.cs.utah.edu/projects/flux/

Abstract

Traditional processor scheduling mechanisms in operating systems are fairly rigid, often supporting only one fixed scheduling policy, or, at most, a few "scheduling classes" whose implementations are closely tied together in the OS kernel. This paper presents *CPU inheritance scheduling*, a novel processor scheduling framework in which arbitrary threads can act as schedulers for other threads. Widely different scheduling policies can be implemented under the framework, and many different policies can coexist in a single system, providing much greater scheduling flexibility. Modular, hierarchical control can be provided over the processor utilization of arbitrary administrative domains, such as processes, jobs, users, and groups, and the CPU resources consumed can be accounted for and attributed accurately. Applications, as well as the OS, can implement customized local scheduling policies; the framework ensures that all the different policies work together logically and predictably. As a side effect, the framework also cleanly addresses priority inversion by providing a generalized form of priority inheritance that automatically works within and among diverse scheduling policies. CPU inheritance scheduling extends naturally to multiprocessors, and supports processor management techniques such as processor affinity[29] and scheduler activations[3]. We show that this flexibility can be provided with acceptable overhead in typical environments, depending on factors such as context switch speed and frequency.

1 Introduction

Traditional operating systems control the sharing of the machine's CPU resources among threads using a fixed scheduling scheme, typically based on priorities. Sometimes a few variants on the basic policy are provided, such as support for fixed-priority threads[17], or several "scheduling classes" to which threads with different purposes can be assigned (e.g., real-time, interactive, or background)[25]. However, even these variants are generally hard-coded into the system implementation and cannot easily be adapted to the needs of individual applications.

In this paper we develop a novel processor scheduling framework based on a generalized notion of priority inheritance. In this framework, known as *CPU inheritance scheduling*, arbitrary threads can act as schedulers for other threads by temporarily *donating* their CPU time to selected threads while waiting on events of interest such as clock/timer interrupts. The receiving threads can further donate their CPU time to other threads, and so on, forming a logical hierarchy of schedulers as illustrated in Figure 1. A scheduler thread can be notified when the thread to which it donated its CPU time no longer needs it (e.g., because the target thread has blocked), so that the CPU can be reassigned to another target. The basic thread dispatching mechanism necessary to implement this framework does not have any notion of thread priority, CPU usage, or clocks and timers; all of these functions, when needed, are implemented by threads acting as schedulers.

Under this framework, arbitrary scheduling policies can be implemented by ordinary threads cooperating with each other through well-defined interfaces that may cross protection boundaries. For example, a fixed-priority multiprocessor scheduling policy can be implemented by maintaining, among a group of scheduler threads (one for each available CPU), a prioritized queue of "client" threads to be scheduled; each scheduler thread successively picks a thread to run and donates its CPU time to the selected target thread while waiting for an interesting event such as quantum expiration (e.g., a clock interrupt). See Figure 2 for an illustration. If the selected thread blocks, its scheduler thread is notified and the CPU is reassigned. On the other hand, if a different event causes the scheduler thread to wake up, the running thread is preempted and the CPU is given back to the scheduler immediately. Other scheduling policies, such as timesharing[23], fixed-priority[12, 17],

This research was supported in part by the Defense Advanced Research Projects Agency, monitored by the Department of the Army, under contract number DABT63–94–C–0058. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies of the U.S. Government.

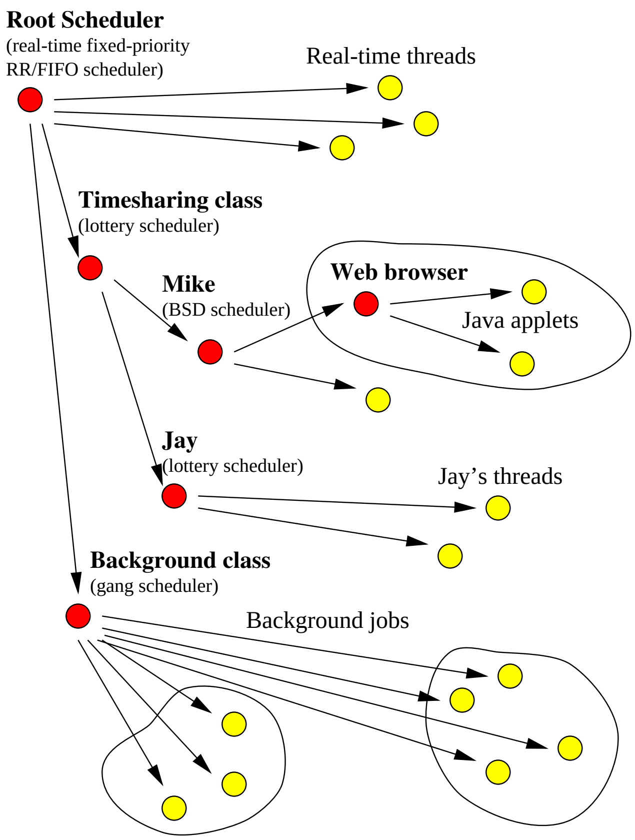

Figure 1: Example scheduling hierarchy. The dark circles represent threads acting as schedulers, while the light circles represent "ordinary" threads.

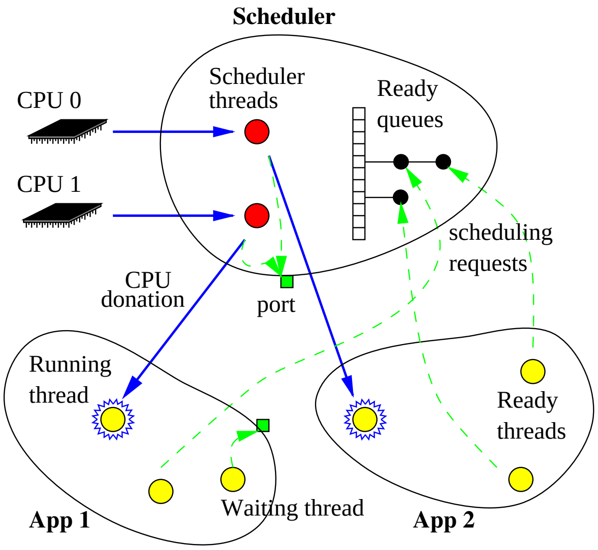

Figure 2: Example fixed-priority scheduler

rate monotonic[22], fair share[9, 16, 19], and lottery/stride scheduling[30, 31, 32], can be implemented in the same way.

This scheduling framework has the following features:

- It supports multiple arbitrary scheduling policies on the same or different processors.
- Since scheduler threads may run either in the OS kernel or in user mode, applications can easily extend or replace the scheduling policies built into the OS.
- It provides hierarchical control over the processor resource usage of different logical or administrative domains in a system, such as users, groups, individual processes, and threads within a process.
- CPU usage accounting can be provided to various degrees of accuracy depending on the resources one is willing to invest.
- Priority inversion is addressed naturally in the presence of resource contention, without the need for explicit priority inheritance/ceiling protocols.
- CPU use is attributed properly even in the presence of priority inheritance.
- The framework naturally extends to multiprocessors.
- Processor affinity scheduling is supported.
- Scheduler activations can be implemented easily.

Based on a prototype implementation in a user-level threads package, we demonstrate that this framework behaves identically to traditional multi-class schedulers in equivalent configurations, and that it provides modular resource control, load insulation, and priority inversion protection automatically across multiple scheduling policies. The additional scheduling overhead caused by this framework is negligible in the test environment, and we show that it should also be acceptable in many kernel environments as well despite greater context switch costs.

The rest of this paper is organized as follows. Section 2 describes motivation and related work, Section 3 presents the CPU inheritance scheduling algorithm in detail, and Section 4 shows how traditional scheduling algorithms can be implemented on this framework. Section 5 demonstrates the behavior and performance properties of our framework through experimental results, and Section 6 concludes with a short reflective summary.

2 Motivation

Traditional operating systems divide a machine's CPU resources among threads using a fixed scheduling scheme, typically based on priorities. However, the requirements imposed on an operating system's scheduler often vary from application to application. For example, for interactive applications, response time is usually the most critical factor—i.e., how quickly the program responds to the user's commands. For batch jobs, throughput is of primary importance but latency is a minor issue. Hard real-time applications require meeting application-specific deadlines, while for soft real-time applications, missing a deadline is unfortunate but not catastrophic. There is no single scheduling scheme that works well for all applications.

Over the years, the importance of providing a variety of scheduling policies on a single machine has waxed and waned, following hardware and application trends. In the early years of computing, use of the entire machine was limited to a single user thread; this evolved into multiprogrammed machines in which a single scheduling policy managed batch jobs effectively. The advent of timesharing on machines still used for batch jobs caused a need for two scheduling policies. As timesharing gradually gave way to single-user workstations and PCs, a single scheduling policy was again adequate for a while.

Supporting multiple scheduling policies is again becoming important, not because of multiple *users* on a system, but because of the variety of concurrent *uses* to which modern systems are put. Multimedia drives the need for soft real-time scheduling policies on general purpose workstations. Untrusted executable content (e.g., Java applets) requires secure control of resource usage while also providing soft real-time guarantees. The hard real-time domain is also making inroads onto general purpose machines as processors and instruments supporting embedded applications become networked, and some customers (e.g., the military) need the ability to shift processing power dynamically to the problem of the moment. All of these additional policies must still work alongside traditional interactive and batch scheduling: though multimedia and video conferencing may be the current rage, this does not mean that users no longer care about getting good interactive response from the word processors, spreadsheets, and other applications that they still rely on daily. Therefore, as the diversity of applications increases, operating systems need to support multiple coexisting processor scheduling policies, in order to meet individual applications' needs as well as to utilize the system's processor resources more efficiently.

2.1 Related Work

One simple approach to providing real-time support in systems with traditional timesharing schedulers, which has been adopted by many commonly-used systems such as Unix, Mach[1, 4], and Windows NT[25], and has even become part of the POSIX standard[17], is support for fixed-priority threads. Although these systems generally still use conventional priority-based timesharing schedulers, they allow real-time applications to disable the normal dynamic priority adjustments on threads specifically designated as "real-time threads," so that those threads always run at a programmer-defined priority. By carefully assigning priorities to the real-time threads in the system and ensuring that all non-real-time threads execute at lower priority, it is possible to obtain the correct behavior for some applications. However, this approach has serious, well-known limitations; in many cases, entirely different non-priority-based scheduling policies are needed[18].

Even in normal interactive and batch-mode computing, traditional priority-based scheduling algorithms are showing their age. For example, these algorithms do not provide a clean way to encapsulate a set of processes/threads as a single unit to isolate and control their processor usage relative to the rest of the system. This shortcoming opens the system to various denial-of-service attacks, the most well-known being the creation of a large number of threads which overwhelm processor resources and crowd out other activity. These vulnerabilities generally don't cause serious problems for machines only used by one person, or when the users of the system fall into one administrative domain and can "complain to the boss" if someone is abusing the system. However, as distributed and mobile computing becomes more prevalent and administrative boundaries become increasingly blurred, this form of system security is becoming more important. This is especially true when completely unknown, untrusted code is downloaded and run in a "secure" environment such as that provided by Java[13] or OmniWare[2]. Schedulers have been designed that address this problem by providing flexible hierarchical control over CPU usage at different administrative boundaries[5, 14, 15, 30, 31, 32]. However, it is not yet clear how these algorithms will address other needs, such as those of hard real-time applications; certainly it seems unlikely that a single "holy grail" of scheduling policies will be found.

With the growing diversity of application needs and scheduling policies, it is increasingly desirable for an operating system to support multiple independent policies. On multiprocessor systems, one simple but limited way of doing this is to run a different scheduling policy on each processor. A more general approach is to allow multiple "scheduling classes" to run on a single processor, with a specific scheduling policy associated with each class. The classes have a strictly ordered priority relationship to each other, so the highest-priority class gets all the CPU time it wants, the next class gets any CPU time left unused by the first class, etc. Although this approach shows promise, one drawback is that since the schedulers for the different classes generally don't communicate or cooperate with each other closely, only the highest-priority scheduling class on a given processor can make any assumptions about how much CPU time it will have to dispense to the threads under its control.

Additionally, most existing multi-policy scheduling mechanisms still require every scheduling policy to be implemented in the kernel and to be closely tied with other kernel mechanisms such as threads, context switching, clocks, and timers. The only existing system we know of that allows different scheduling policies to be implemented in separate, unprivileged protection domains is the Aegis Exokernel[8]. However, the Aegis scheduling mechanism was not described at length and does not address important issues such as timing, CPU usage accounting, and multiprocessor scheduling. Both Aegis's scheduling mechanism and our framework are based on the use of a "directed yield" primitive which allows application-level threads to schedule each other; this core concept originally comes from coroutines[6], in which directed yield is the *only* way thread switching is done. We believe our scheduling framework could be implemented in an Exokernel environment through the use of suitable kernel primitives, application-level support code, and standardized inter-process scheduling protocols.

Finally, most existing systems still suffer from various priority inversion problems. Priority inversion occurs when a high-priority thread requesting a service has to wait arbitrarily long for a low-priority thread to finish being serviced. This problem can be addressed in priority-based scheduling algorithms by supporting priority inheritance[7, 26], wherein the thread holding up the service inherits the priority of the highest priority thread waiting for service. In some cases this approach can be adapted to other scheduling policies, such as with ticket transfer in lottery scheduling[31]. However, the problem of resolving priority inversion between threads of different scheduling classes using policies with different and incomparable notions of "priority" has not been addressed so far.

3 CPU Inheritance Scheduling

In our scheduling model, as in traditional systems, a *thread* is a virtual CPU whose purpose is to execute instructions. A thread may or may not have a real CPU assigned to it at any given instant; a running thread may be preempted and its CPU reassigned to another thread at any time. (For the purposes of this framework, it is not important whether these threads are kernel-managed or user-managed threads, or whether they run in supervisor or user mode.)

In traditional systems, threads are generally scheduled by some lower-level entity, such as a scheduler in the OS kernel or a user-level threads package. In contrast, the basic idea of CPU inheritance scheduling is that threads are scheduled *by other threads*. Any thread that has a real CPU available to it at a given instant can *donate* its CPU temporarily to another specific thread instead of using the CPU itself. This operation is similar to priority inheritance in conventional systems, except that it is done explicitly by the donating thread, and no notion of "priority" is involved, only a direct transfer of the CPU from one thread to another; hence the name "CPU inheritance."

A *scheduler thread* is a thread that spends most of its time donating its CPU resources to *client threads*; the client threads thus *inherit* some portion of the scheduler thread's CPU resources, and treat that portion as *their* virtual CPU. Client threads can in turn act as scheduler threads, distributing their CPU time among their own clients, and so on, forming a scheduling hierarchy.

The only threads in the system that inherently have actual CPU time available to them are the set of *root scheduler threads*; other threads can only run if CPU time is donated to them. There is one root scheduler thread for each real CPU in the system; each CPU is permanently dedicated to supplying CPU time to its associated root scheduler thread. The actions of the root scheduler thread on a given CPU determine the base-level scheduling policy for that CPU.

3.1 The Dispatcher

Even though all high-level scheduling decisions are performed by threads, a small low-level mechanism is still needed to implement primitive thread management functions. We call this low-level mechanism the *dispatcher* to distinguish it clearly from high-level schedulers.

The role of the dispatcher is to to implement thread blocking, unblocking, and CPU donation. The dispatcher fields events and directs them to threads waiting on those events, without actually making any real scheduling decisions. Events can be synchronous, such as an explicit wake-up of a sleeping thread by a running thread, or asynchronous, such an external interrupt (e.g., I/O or timer). The dispatcher itself is not a thread; it merely runs in the context of whatever thread owns the CPU at the time an event of interest occurs.

The dispatcher inherently contains no notion of thread priorities, CPU usage, or even clocks and timers. In a kernel supporting CPU inheritance scheduling, the dispatcher is the only scheduling component that *must* be in the kernel; all other scheduling code could in theory run in user-mode threads outside of the kernel (although this "purist" approach may be impractical for performance reasons).

3.2 Requesting CPU Time

Because no thread (except a root scheduler thread) can ever run unless some other thread donates CPU time to it, a newly-created or newly-woken thread must request CPU time from some scheduler before it can run. Each thread has an associated scheduler which has primary responsibility for providing CPU time to the thread. When the thread becomes ready, the dispatcher notifies the thread's scheduler that the thread needs CPU time. The exact form such a notification takes is not important; in our implementation, notifications are simply IPC messages sent by the dispatcher to Mach-like message ports.

When a thread wakes up, the notification it produces may in turn wake up a scheduler thread waiting to receive such messages on its port. Waking up that scheduler thread will cause another notification to be sent to *its* scheduler, which may wake up still another thread, and so on. Thus, waking up an arbitrary thread can cause a chain of wakeups to propagate back through the scheduler hierarchy. Eventually, this propagation may wake up a scheduler thread that is currently being supplied with CPU time but is donating it to some other thread. In that case, the thread currently running on that CPU is preempted and control is given back to the woken scheduler thread immediately; the scheduler can then make a decision to re-run the preempted client, switch to the newly-woken client, or even run some other thread. Alternatively, the propagation of wake-up events may terminate at some point, for example because a notified scheduler is already awake (not waiting for messages) but has been preempted. In that case, the dispatcher knows that the wake-up event is irrelevant for scheduling purposes at the moment, so the currently running thread is resumed immediately.

3.3 Relinquishing the CPU

At any time, a running thread may block to wait for one or more events to occur, such as I/O completion or timer expiration. When a thread blocks, the dispatcher returns control of the CPU to the scheduler thread that provided it to the running thread. That scheduler may then choose another thread to run, or it may relinquish the CPU to *its* scheduler, and so on up the line until some scheduler finds work to do.

3.4 Voluntary Donation

Instead of simply blocking, a running thread can voluntarily donate its CPU to another thread while waiting on an event of interest; this is done in situations where priority inheritance would traditionally be used. For example, when a thread attempts to obtain a lock that is already held, it may donate its CPU time to the thread holding the lock; similarly, when a thread makes an RPC to a server thread, the client thread may donate its CPU time to the server for the duration of the request. When the event of interest occurs, the donation ends and the CPU is given back to the original thread. In our implementation of this framework, the basic synchronization and IPC primitives automatically invoke the dispatcher to perform voluntary donation appropriately when the thread blocks; however, voluntary donation could also be done optionally or through explicit dispatcher calls.

It is possible for a single thread to inherit CPU time from more than one source at a given time: for example, a thread holding a lock may inherit CPU time from several threads waiting on that lock in addition to its own scheduler. In this case, the effect is that the thread has the opportunity to run at any time *any* of its donor threads would have been able to run. A thread only "uses" one CPU source at a time; however, if its current CPU source runs out (e.g., due to quantum expiration), the dispatcher will automatically send scheduling request notifications on behalf of all the threads depending on (donating to) to the preempted thread, effectively switching the thread automatically to another available CPU source. One potential worry is that a thread consuming CPU time from many sources will cause an "avalanche effect" every time it is preempted or woken as the dispatcher fires off several scheduling requests, each of which may cause more scheduling requests as intermediate-level schedulers are woken up. We believe that in practice it should be uncommon for a thread to inherit from more than one or two other threads at once, so this should not be a major problem; however, we have not yet examined this issue in detail.

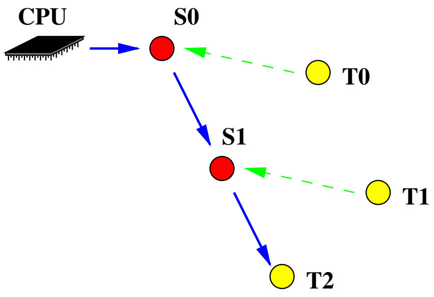

Figure 3: CPU donation chain

3.5 The `schedule` operation

The call a scheduler thread makes to donate CPU time to a client thread is simply a special form of voluntary CPU donation, in which the thread to donate to and the event to wait for can be specified explicitly. In our implementation, the `schedule` operation takes as parameters a thread to donate to, a port on which to wait for messages from other client threads, and a *wakeup sensitivity* parameter indicating in what situations the scheduler should be woken. The operation donates the CPU to the specified target thread and puts the scheduler thread to sleep on the specified port; if a message arrives on that port, such as a notification that another client thread has been woken or a message indicating that a timer has expired, then the `schedule` operation terminates and control is returned to the scheduler thread.

In addition, the `schedule` operation may be interrupted before a message arrives, depending on the behavior of the thread to which the CPU is being donated and the value of the wakeup sensitivity parameter. The wakeup sensitivity level acts as a hint to the dispatcher allowing it to avoid waking up the scheduler thread except when necessary; it is only an optimization and is not in theory required for the system to work. Our system supports the following three sensitivity levels:

- `WAKEUP_ON_BLOCK`: If the target of the `schedule` operation blocks without further donating the CPU, then the `schedule` operation terminates and control is returned to the scheduler immediately. For example, in Figure 3, if scheduler thread S_1 has donated the CPU to thread T_2 using this wakeup sensitivity setting, but T_2 blocks and can no longer use the CPU, then S_1 will receive control again. `WAKEUP_ON_BLOCK` is the "most sensitive" setting, and is typically used when the scheduler has other client threads waiting to run.

- `WAKEUP_ON_SWITCH`: If the client thread using the CPU (T_2) blocks, control is *not* immediately returned to its scheduler (S_1): the dispatcher behaves instead as if S_1 itself blocked, and passes control on back to *its* scheduler, S_0. If T_2 is subsequently woken up, then when S_0 again provides the CPU to S_1, the dispatcher passes control directly back to T_2 without actually running S_1. However, if a *different* client of S_1, such as T_1, wakes up and sends a notification to S_1's message port, then S_1's `schedule` operation *will* be interrupted. This sensitivity level is typically used when a scheduler has only one thread to run at the moment and doesn't care when that thread blocks or unblocks, but it still wants to switch between different client threads manually: for example, the scheduler may need to start and stop timers when switching between client threads.
- `WAKEUP_ON_CONFLICT`: The scheduler is only awakened if a *second* client thread wakes up while the scheduler is already donating CPU to a client (e.g., if T_1 wakes up while T_2 is running). If T_2 blocks, the scheduler blocks too; then, if *any* single client of scheduler S_1 is subsequently woken, such as T_1, the dispatcher passes control directly to the woken client thread without waking up the scheduler. At this weakest sensitivity level, the dispatcher is allowed to switch among client threads freely; the scheduler only acts as a "conflict resolver," making decisions when two client threads are runnable at the same time.

4 Implementing High-level Schedulers

This section describes how the basic CPU inheritance mechanism can be used to implement high-level scheduling policies and related features such as CPU usage accounting, processor affinity, and scheduler activations.

4.1 Single-CPU Schedulers

Figure 4 shows a simplified code fragment from a basic non-prioritized FIFO scheduler in our system. The scheduler keeps a queue of client threads waiting for CPU time, and successively runs each one using the `schedule` operation while waiting for messages to arrive on its port (e.g., notifications from newly-woken client threads). When there are no client threads waiting to be run, the scheduler uses the ordinary non-donating `msg_rcv` operation instead of the `schedule` operation in order to relinquish the CPU while waiting for messages. If there is only one client thread in the scheduler's queue, the scheduler uses the weaker `WAKEUP_ON_CONFLICT` sensitivity level when running it to indicate that the dispatcher may switch among client threads arbitrarily as long as only one client thread attempts to use the CPU at a time.

```
void fifo_scheduling_loop()
{
  cur_thread = NULL;
  more_msgs = 1;
  for (;;) {
    if (more_msgs) {
      more_msgs = msg_rcv(&fifo_pset, &msg);
    } else {

      /* Select the thread to run next. */
      if ((cur_thread == 0) &&
          !queue_empty(&fifo_runq))
        cur_thread = q_remove(&fifo_runq);

      /* Select wakeup sensitivity level. */
      cond = q_is_empty(&fifo_runq) ?
        WAKEUP_ON_CONFLICT : WAKEUP_ON_BLOCK;

      /* Schedule and wait for messages. */
      if (cur_thread != NULL)
        more_msgs = schedule(fifo_pset, &msg,
                             cur_thread, cond);
      else
        more_msgs = msg_rcv(fifo_pset, &msg);
    }

    /* Process the received message. */
    switch (msg.request_code) {
      case MSG_SCHED_REQUEST:
        /* A client thread wants to run. */
        q_enter(&fifo_runq, msg.thread_id);
        break;
      case MSG_SCHED_BLOCKED:
        /* Last thread gave up the CPU. */
        cur_thread = 0;
        break;
    }
  }
}
```

Figure 4: Example single-processor FIFO scheduler.

4.2 Timekeeping and Preemption

The simple FIFO scheduler above can be converted to a round-robin scheduler by introducing some form of clock or timer. For example, if the scheduler is the root scheduler on a CPU, then the scheduler might be directly responsible for servicing clock interrupts. Alternatively, the scheduler may rely on a separate "timer thread" to notify it when a periodic timer expires. In any case, a timer expiration or clock interrupt is indicated to the scheduler by a message being sent to the scheduler's port. This message causes the scheduler to break out of its `schedule` operation and preempt the CPU from whatever client thread was using it. The scheduler can then move that client to the tail of the ready queue for its priority and give control to the next client thread at the same priority.

4.3 Multiprocessor Support

Since the example scheduler above only contains a single scheduler thread, it can only schedule a single client thread at once. Therefore, although it can be run on a multiprocessor system, it cannot take advantage of multiple processors simultaneously. For example, a separate instance of the FIFO scheduler could be run as the root scheduler on each processor; a client thread assigned to a given scheduler is effectively bound to its scheduler's CPU. Although in some situations this arrangement can be useful, e.g., when each processor is to be dedicated to a particular purpose, in most cases it is not what is needed.

In order for a scheduler to provide "real" multiprocessor scheduling to its clients, where different client threads can be dynamically assigned to different processors on demand, the scheduler itself must be multi-threaded. Assume for now that the scheduler knows how many processors are available, and can bind threads to processors. (This is trivial if the scheduler is run as the root scheduler on some or all processors; we will show later how this requirement can be met for non-root schedulers.) The scheduler creates a separate thread bound to each processor; each of these scheduler threads then selects and runs client threads on that processor. The scheduler threads cooperate with each other using shared variables, e.g., shared run queues in the case of a multiprocessor FIFO scheduler.

Since a scheduler's client threads are supposed to be unaware that they are being scheduled on multiple processors, the scheduler exports only a single port representing the scheduler as a whole to all of its clients. When a client thread wakes up and sends a notification to the scheduler port, the dispatcher arbitrarily wakes up one of the scheduler threads waiting on that port. A good policy is for the dispatcher to wake up the scheduler thread associated with the CPU on which the wakeup is being done; this allows the scheduler to be invoked on the local processor without interfering with other processors unnecessarily. If the woken scheduler thread discovers that the newly woken client should be run on a different processor (e.g., because it is already running a high-priority client but another scheduler thread is running a low-priority client), it can interrupt the other scheduler thread's `schedule` operation by sending it a message or "signal"; this corresponds to sending interprocessor interrupts in traditional systems.

4.3.1 Processor Affinity

Scheduling policies that take processor affinity into consideration[27, 28, 29], can be implemented by treating each scheduler thread as a processor and attempting to schedule a client thread from the same scheduler thread that previously donated CPU time to that client thread. Of course, this will only work if the scheduler threads themselves are consistently run on the same processor. Any processor affinity support in one scheduling layer will only work well if all the layers between it and the root scheduler also consider processor affinity; a mechanism to ensure this is described in the next section.

4.3.2 Scheduler Activations

In the common case, client threads "communicate" with their schedulers implicitly through notifications sent by the dispatcher on behalf of the client threads. However, there is nothing to prevent client threads from *explicitly* communicating with their schedulers through some additional interface. One particularly useful explicit interface is *scheduler activations*[3], which allows clients to determine initially and later track the number of actual processors available to them. Clients can then create or destroy threads as appropriate in order to make use of all available processors without creating superfluous threads that compete with each other uselessly on a single processor.

Since scheduler threads are notified by the dispatcher when a client thread blocks and temporarily cannot use the CPU (e.g., because the thread is waiting for an I/O request or a page fault to be serviced), the scheduler can notify the client in such a situation and give the client an opportunity to create a new thread to make use of the CPU while the original thread is blocked. For example, a client can create a pool of "dormant" threads, or "activations," which the scheduler knows about but normally never runs. If a CPU becomes available, e.g., because of another client thread blocking, the scheduler "activates" one of these dormant threads on the CPU vacated by the blocked client thread. Later, when the blocked thread eventually unblocks and requests CPU time again, the scheduler preempts one of the currently running client threads and notifies the client that it should make one of the active threads dormant again.

Scheduler activations were originally devised to provide better support for application-specific thread packages running in a single user mode process. In an OS kernel that implements CPU inheritance scheduling, extending a scheduler to provide this support should be quite straightforward. However, in a multiprocessor system based on CPU inheritance scheduling, scheduler activations are also useful in stacking of first-class schedulers. As mentioned previously, multiprocessor schedulers need to know the number of processors available in order to use the processors efficiently. As long as a base-level scheduler (e.g., the root scheduler on a set of CPUs) supports scheduler activations, a higher-level multiprocessor scheduler running as a client of the base-level scheduler can use the scheduler activations interface to track the number of processors available and schedule *its* clients effectively. (Simple single-threaded schedulers that only make use of one CPU at a time don't need scheduler activations and can be stacked on top of any scheduler.)

4.4 Timing

Most scheduling algorithms require a *measurable* notion of time in order to implement preemptive scheduling. For most schedulers, a periodic interrupt is sufficient, although some real-time schedulers may need finer-grained timers whose periods can be changed at each quantum. With CPU inheritance scheduling, the precise nature of the timing mechanism is not important to the general framework; all that is needed is some way for a scheduler thread to be woken up after some amount of time has elapsed. In our implementation, schedulers can register *timeouts* with a central clock interrupt handler; when a timeout occurs, a message is sent to the appropriate scheduler's port, waking up the scheduler. The dispatcher automatically preempts the running thread if necessary and passes control to the scheduler so that it can account for the elapsed time and potentially switch to a different client thread.

4.4.1 CPU Usage Accounting

Besides simply deciding which thread to run next, schedulers often must account for CPU resources consumed. CPU accounting information is used for a variety of purposes, such as reporting usage statistics to the user on demand, modifying scheduling policy based on CPU usage (e.g., dynamically adjusting thread priority), or billing a customer for CPU time consumed for a particular job. As with scheduling policies, there are many possible CPU accounting mechanisms, each with different cost/benefit tradeoffs. The CPU inheritance scheduling framework allows a variety of accounting policies to be implemented by scheduler threads.

There are two well-known approaches to CPU usage accounting: *statistical* and *time stamp-based*[4]. With statistical accounting, the scheduler wakes up on every clock tick, checks the currently running thread, and charges the entire time quantum to that thread. This method is quite efficient, since the scheduler generally wakes up on every clock tick anyway; however, it provides limited accuracy. A variation that provides better accuracy at slightly higher cost is to sample the current thread at random points *between* clock ticks[21]. Alternatively, with time stamp-based accounting, the scheduler reads the current time at every context switch and charges the appropriate thread for the time since the last context switch. This method provides extremely high accuracy, but also imposes a high cost due to lengthened context switch times, especially on systems on which reading the current time is expensive.

In the root scheduler on a processor, these methods can be applied directly. To implement statistical accounting, the scheduler simply checks what thread it ran last upon being woken up by the arrival of a timeout message. To implement time stamp-based accounting, the scheduler reads the current time each time it schedules a different client thread. The scheduler must use the `WAKEUP_ON_BLOCK` sensitivity level in order to ensure that it can check the time at each thread switch and to ensure that idle time is not charged to any thread.

For schedulers stacked on top of other schedulers, CPU accounting becomes a little more complicated because the CPU time supplied to such a scheduler is already "virtual" and cannot be measured accurately by a wall-clock timer. For example, in Figure 3, if scheduler S_1 measures T_2's CPU usage using a wall-clock timer, then it may mistakenly charge against T_2 time actually used by the high-priority thread T_0, which S_1 has no knowledge of because it is scheduled by the root scheduler S_0. In many cases, this inaccuracy caused by stacked schedulers may be ignored in practice on the assumption that high-priority threads and schedulers will consume relatively little CPU time; this assumption is similar to the one made in many existing kernels that hardware interrupt handlers consume little enough CPU time that they may be ignored for accounting purposes. In situations in which this assumption is not valid and accurate CPU accounting is needed for stacked schedulers, virtual CPU time information provided by base-level schedulers can be used instead of wall-clock time, at the cost of additional communication between schedulers. For example, in Figure 3, at each clock tick (for statistical accounting) or each context switch (for time stamp-based accounting), scheduler S_1 could request its own virtual CPU time usage from S_0 instead of checking the current wall-clock time. It then uses this virtual time information to maintain usage statistics for *its* clients, T_1 and T_2.

4.4.2 Effects of CPU Donation on Timing

As mentioned earlier, CPU donation can occur implicitly as well as explicitly, e.g., to avoid priority inversion when a high-priority thread attempts to lock a resource already held by a low-priority thread. For example, in Figure 5, scheduler S_0 has donated the CPU to high-priority thread T_0 in preference over low-priority thread T_1. However, it turns out that T_1 is holding a resource needed by T_0, so T_0 implicitly donates its CPU time to T_1. Since this donation merely extends the scheduling chain, S_0 is unaware that the switch occurred, and it continues to charge CPU time used to T_0 instead of T_1 which is the thread that is *actually* using the CPU.

While it may seem non-intuitive at first, in practice this is often precisely the desired behavior; it stems from the basic rule that with privilege comes responsibility. While T_0 is donating CPU to T_1, T_1 is effectively doing work *on behalf of* T_0, even if T_1 is unaware that it is receiving

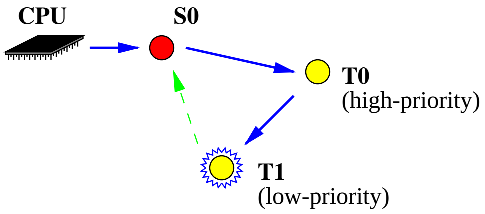

Figure 5: Implicit CPU donation from high-priority thread T_0 to low-priority thread T_1 to avoid priority inversion during a resource conflict.

CPU time from T_0. Since T_0's share of the CPU is being used to perform this work, the CPU time consumed must also be charged to T_0, even if the work is actually being done by another thread. Demonstrated another way, charging T_1 rather than T_0 would allow the accounting system to be subverted. For example, if high-priority CPU time is "expensive" and low-priority CPU time is "cheap," then T_0 could collude with T_1 to use high-priority CPU time while being charged the low-priority "rate" simply by arranging for T_1 to do all the actual work while T_0 blocks on a lock perpetually held by T_1. This ability to charge the "proper" thread for CPU usage even in the presence of priority inheritance is unnatural and difficult to implement in traditional systems, and therefore is generally *not* implemented by them; on the other hand, this feature falls out of the CPU inheritance framework automatically.

4.5 Threads with Multiple Scheduling Policies

Sometimes it is desirable for a single thread to be associated with two or more scheduling policies at once. For example, a thread may normally run in a real-time rate-monotonic scheduling class; however, if the thread's quantum expires before its work is done, it may be desirable for the thread to drop down to the normal timesharing class instead of simply stopping dead in its tracks.

Support for multiple scheduling policies per thread can be achieved in our framework even under a dispatcher that supports only a single scheduler association per thread, by creating one or more additional "dummy" threads which perpetually donate their CPU time to the "primary" thread. Each of these threads can have a different scheduler association, and the dispatcher automatically ensures that the primary thread always uses the highest priority scheduler available, as described in Section 3.4. In situations in which this solution is not acceptable due to performance or memory overhead, the dispatcher could easily be extended to allow multiple schedulers to be associated with a single thread, so that when such a thread becomes runnable the dispatcher automatically notifies all of the appropriate schedulers.

Although it may at first seem inefficient to notify two or more schedulers when a single thread awakes, in practice many of these notifications never actually need to be *delivered*. For example, if a real-time/timesharing thread wakes up, finishes all of its work and goes back to sleep again before its real-time scheduling quantum is expired (presumably the common case), then the notification posted to the low-priority timesharing scheduler at wakeup time will be canceled (removed from the queue) when the thread goes to sleep again, so the timesharing scheduler effectively never sees it.

5 Analysis and Experimental Results

We have created a prototype implementation of this scheduling framework and devised a number of tests to evaluate its flexibility, performance, and practicality.

5.1 Test Environment

In order to provide a clean, easily controllable environment, we implemented an initial prototype of this framework in a user-level threads package. The threads package supports common abstractions such as mutexes, condition variables, and message ports for inter-thread communication and synchronization. The package implements separate thread stacks with `setjmp`/`longjmp`, and the virtual CPU timer alarm signal (`SIGVTALRM`) is used to provide preemption and simulate clock interrupts. We used the virtual CPU timer instead of the wall-clock timer in order to minimize distortion of the results due to other activity in the host Unix system; in a "real" user-level threads package based on this scheduling framework, the normal wall-clock timer would probably be used instead. Our prototype allows threads to wait on only one event at a time; however, there is nothing about the framework that makes it incompatible with thread models in which threads can wait on multiple events at once[25].

Although implemented in user space, our prototype is designed to reflect the structure and execution environment of an actual OS kernel running in privileged mode. For example, the dispatcher itself is passive, nonpreemptible code executed in the context of the currently running thread, an environment similar to that of BSD and other traditional kernels. The dispatcher is cleanly isolated from the rest of the system, and supports scheduling hierarchies of unlimited depth and complexity. Our prototype schedulers are also isolated from each other and from their clients; the various components communicate with each other through message-based protocols that could easily be adapted to operate across protection domains using IPC.

Except where otherwise noted, all measurements were taken on a 100MHz Pentium PC with 32 megabytes of RAM, running FreeBSD 2.1.5.

5.2 Scheduler Configuration

The following experiments are based on scheduling hierarchy shown in Figure 6, which is designed to reflect

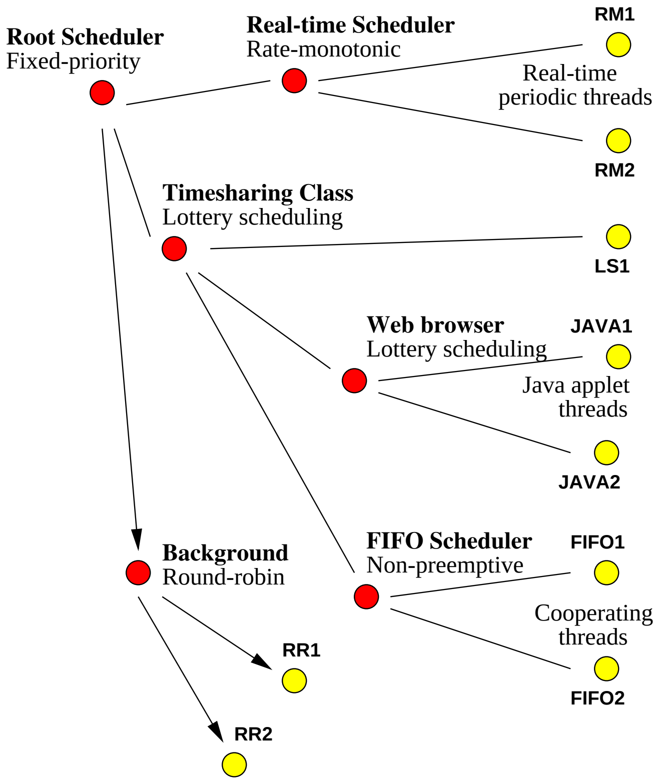

Figure 6: Multilevel scheduling hierarchy used for tests.

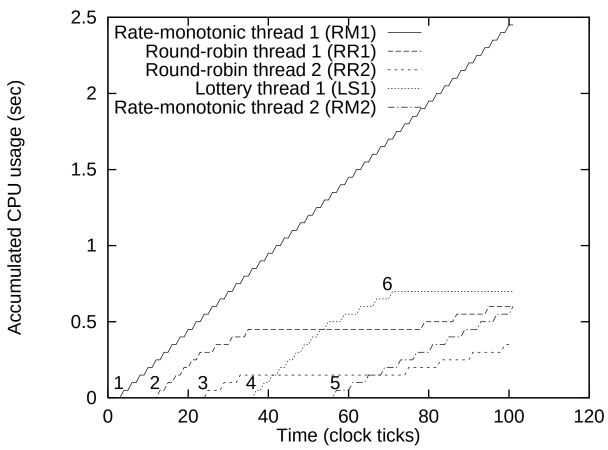

Figure 7: Behavior of a multilevel scheduling hierarchy. Events are numbered according to the description in Section 5.3.

the activity present in a general-purpose environment. In this environment, the root scheduler is a nonpreemptive fixed-priority scheduler with a first-come-first-served policy among threads of same priority. This scheduler arbitrates between three scheduling classes: a real-time rate-monotonic scheduler at the highest priority, a lottery scheduler providing a timesharing class, and a simple round-robin scheduler for background jobs. On top of the lottery scheduler managing the timesharing class, an application-specific third-level scheduler manages two threads using lottery scheduling (e.g., a Web browser managing Java applet threads). Finally, a second application-specific scheduler in the timesharing class schedules two cooperating threads using nonpreemptive FIFO scheduling.

5.3 Scheduling Behavior

The purpose of our first experiment is simply to verify that our framework works as expected, producing the same scheduling behavior as traditional single- and multi-class schedulers do in equivalent configurations. Figure 7 shows the scheduling behavior of the threads simulated in our scheduling hierarchy over a time period covering the following sequence of events:

1. A rate-monotonic thread, `RM1`, becomes runnable and starts consuming CPU time periodically using a fixed reservation of 50%.
2. A round-robin background thread, `RR1`, begins a long computation in which it consumes all the CPU time it can get. The rate-monotonic thread, `RM1`, is unaffected because the fixed-priority root scheduler always schedules the real-time scheduler in preference to the background scheduler.
3. A second round-robin background thread, `RR2`, becomes runnable at the same priority as `RR1`.
4. An application thread in the timesharing class becomes runnable, stealing all available CPU time from the background jobs for a short burst of time (e.g., a spreadsheet recalculation).
5. A second rate-monotonic thread, `RM2`, becomes runnable, consuming 25% of the total CPU time. The time available to the timesharing class is reduced accordingly, but `RM1` remains unaffected.
6. The timesharing thread, `LS1`, finishes its burst of activity, allowing the background jobs to continue once again.

5.4 Modular Control

In order to demonstrate the modular control provided by the framework, we now consider the two third-level schedulers in our example configuration, which represent application-specific schedulers. The first, implementing lottery scheduling, simulates a web browser arbitrating between two downloaded Java applets. The browser can vary the ticket ratio allocated to each applet according to some policy, e.g., giving more CPU time to applets whose output windows are currently on-screen. The second application-level scheduler, a simple non-preemptive FIFO scheduler, represents an application that uses threads merely for programmatic purposes, and does not need to maintain any notion of priority or timeliness between its threads. Using a non-preemptive scheduler in such an application eliminates unnecessary contention and context switches between application threads without giving up the benefits of preemptive scheduling in other parts of the system.

In this example, the two Java applet threads perpetually consume as much CPU time as possible, and each of the cooperating FIFO threads alternately run for some time and then yield control to the other. Initially, the ticket ratio between the web browser and the cooperative application is 4:1, the ratio between the applet threads is 1:4, and each of the FIFO threads compute for approximately the same time before yielding to the other. Figure 8 shows the behavior of the system across four events:

1. At time 2000, the web browser changes the relative ticket ratio between the Java applet threads to be 1:1, e.g., because the first has come on-screen. The amount of CPU time allocated to the cooperative application remains unchanged, however, demonstrating *load insulation*. Since the timesharing-class scheduler and the web browser's scheduler are both lottery schedulers, this example is equivalent to the use of two *currencies* in a single lottery scheduler[31].

2. The cooperative thread `FIFO1` changes its computation so that it now consumes four times the amount of CPU time before yielding to `FIFO2`. The effective distribution of CPU time changes to 4:1, reflecting the fact that the FIFO scheduling policy makes no attempt at fairness. However, since the timesharing-class scheduler uses a proportional-share policy, the applications are insulated from each other and the web browser is unaffected. This example demonstrates load insulation between *different* scheduling policies.

3. The user changes the ticket allocation between the two applications to 1:1. Both sub-schedulers automatically adjust to the new allocation according to their individual scheduling policies.

4. Finally, the priority of `RR1` is raised above that of `RR2`, causing it to receive all of the CPU time allocated to the round-robin scheduler while leaving the other scheduling domains unaffected.

5.5 Avoidance of Priority Inversion

In the next experiment we demonstrate how priority inversion can be avoided in our framework even among threads running under different scheduling policies. For this experiment, we use rate-monotonic thread `RM1`, lottery scheduled thread `LS1`, and background thread `RR1`. The rate-monotonic thread and the background thread together simulate a data acquisition application in which a high-priority real-time thread periodically receives data and stores it in a memory buffer, and a low-priority computational thread inspects that data in the background using spare CPU cycles. In our simulated environment, the high-priority thread wakes up every five clock ticks, acquires a shared mutex lock, and then releases it one clock tick later. The low-priority thread alternately acquires the lock and computes for up to three clock ticks, and then releases it for a varying amount of time. The lottery scheduled thread, `LS1`, is a medium-priority thread representing other priority inversion-producing activity in the system; it alternately runs and sleeps for random amounts of time up to 20 ticks, but otherwise does not interact with the other threads.

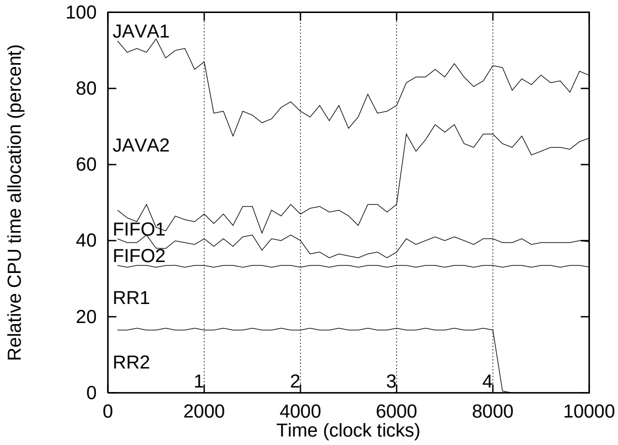

Figure 8: Demonstration of modular control of CPU allocation. Space between the lines represents the percentage of CPU time used by a particular thread. Events occur at 2000-tick intervals, and correspond to the description in Section 5.4.

Figure 9 shows a plot of the distribution of latencies experienced by the high-priority thread in attempting to acquire the shared mutex, with and without the voluntary donation protocol described in Section 3.4. When the inheritance protocol is in effect, maximum latency is bounded by the maximum amount of time that the low-priority thread runs with the lock held (three clock ticks), and does *not* reflect the much larger delays that would be imposed if `LS1` could prevent `RR1` from running while `RM1` is waiting for it to release the shared lock. Thus, the framework provides correct real-time behavior even though all three threads run under different scheduling policies: it is not *priority inheritance*, but the more general *CPU inheritance* protocol, that makes this work.

5.6 Multiprocessor Scheduling

To test the ability of our framework to support multiprocessor scheduling, we extended the test environment to simulate a multiprocessor. Since a real multiprocessor machine was not available for this test, we emulated one by time slicing the BSD process at a fine granularity among several "virtual processors." Each processor has its own

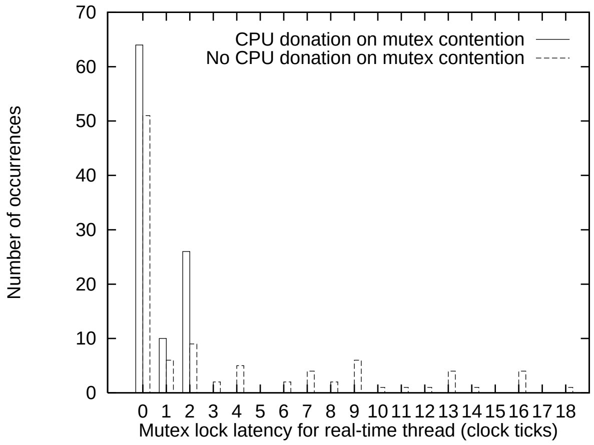

Figure 9: Priority inheritance between schedulers

dispatcher and root scheduler thread, but the root scheduler threads cooperate as described in Section 4.3 to implement a single fixed-priority multiprocessor scheduler with a shared ready queue for each priority level. We tested the multiprocessor scheduler by implementing a simple parallel database lookup application, in which multiple threads repeatedly search for a randomly chosen item in a shared binary tree, locking and unlocking the nodes of the tree as they descend. We ran this application on 1, 2, 4, and 8-processor configurations, with varying numbers of worker threads; in each case, the behavior and performance was exactly as would be expected from a comparable traditional scheduler. Although this experiment demonstrates that the framework extends to multiprocessors, we have not yet explored the effects of stacking multiprocessor scheduling policies, nor have we implemented scheduler activations, which we suspect will be critical to making CPU inheritance scheduling work well in many practical multiprocessor situations.

5.7 Scheduling Overhead

While the preceding experiments demonstrate the basic properties and potential usefulness of our scheduling framework, it remains to be seen whether it is efficient enough in practice. In comparison to traditional multi-class scheduling mechanisms, our framework introduces two additional sources of overhead: first, the overhead caused by the dispatcher itself while computing the thread to switch to after a given event, and second, the overhead resulting from additional context switches to and from scheduler threads. The computation overhead caused by a given scheduling algorithm is not an issue, since the same algorithms can be used in each case.

5.7.1 Dispatcher Costs

To address the first issue, Table 1 shows the basic costs of the dispatcher's computation. These costs are dependent on the depth of the scheduling hierarchy since the dispatcher must often iterate through trees and linked lists whose length is proportional to the depth of the hierarchy.

Scheduling Hierarchy Depth	Dispatch Time (μs)
Root scheduler only	8.0
2-level scheduling	11.2
3-level scheduling	14.0
4-level scheduling	16.2
8-level scheduling	24.4

Table 1: Basic dispatching costs.

One concern raised by the dependence of dispatch time on scheduling hierarchy depth is that, since in theory the framework imposes no depth limit and the dispatcher itself is always effectively the "highest priority activity in the system," this creates a source of unbounded priority inversion that would be unacceptable in hard real-time systems. However, in practice there is no reason a particular dispatcher must support unlimited depth; the dispatcher in a hard real-time system could limit the depth to four or eight levels, which should be sufficient for all practical purposes and still impose little computational overhead.

5.7.2 Context Switch Costs

The second type of overhead in our framework is the cost of additional context switches to and from scheduler threads. Unfortunately, the cost of a context switch varies widely between different environments, from user-level threads packages in which context switches are almost free, to monolithic kernels in which context switches are several orders of magnitude more expensive. An obvious inference is that our framework is likely to be practical in some environments but not in others. Therefore, in order to gain a meaningful idea of how expensive our framework is likely to be in a given environment, we first measure the *number* of additional context switches produced, which varies with different applications and system loads but is not dependent on thread and protection boundary-crossing costs.

Table 2 shows the context switch statistics observed for each of several example applications: "Client/Server" is an application in which a number of client threads repeatedly invoke services on a smaller set of server threads; "Parallel Database" is the multiprocessor binary search application described in Section 5.6; "Real-time" is the data acquisition application from Section 5.5; finally, "General" is the test in Section 5.3 representing general-purpose computing activity. The chart shows the number of times the dispatcher switched to each thread over the course of the test; user threads are shown separately from scheduler threads. The last row in the table shows the percentage of total context switches that were invocations of scheduler threads. This percentage is consistently near 50% regardless of the application; this indicates that, on average,

	Client/ Server	Parallel Database	Real-time	General
RM1	57		322	101
RM2	19			26
RM3	19			
LS1	25		622	17
JAVA1	46			
FIFO1	9			
RR1	114	238	249	7
RR2	3	242		14
RR3		234		
RR4		243		
User invocations	492	957	1193	165
Root scheduler	262	956	1237	142
Rate monotonic	43		1	65
Lottery scheduler	30		57	3
Java thread scheduler	2			
FIFO scheduler	1			
Round-robin scheduler	8		8	8
Scheduler invocations	346	956	1303	218
Total context switches	838	1913	2496	383
Scheduler invoc. rate	41%	50%	52%	56%

Table 2: Switch Costs for Simple Applications.

approximately one additional scheduler thread invocation can be expected in our framework for each ordinary context switch. However, note that this chart is overly pessimistic because the "scheduler invocations" figures include *all* context switches to scheduler threads, not only those directly related to scheduling: in particular, it includes context switches caused by explicit RPCs, which greatly inflate the counts for the root scheduler in our system because our root scheduler thread also provides message-based timers and other facilities to the rest of the system.

5.7.3 Overhead in Kernel Environments

Although we have not yet implemented our framework in a kernel environment, we can get some idea of how it would perform in such an environment based on the results shown above and a knowledge of how real-world applications exercise the scheduler. Table 3 shows system-wide statistics we gathered for several familiar applications running on FreeBSD 2.1.5, measured using BSD's `vmstat` command. `gzip` is a compute-intensive application compressing an 8MB file; `gcc` is a build of a 20,000-line program using the GNU C compiler; `tar` represents an I/O-intensive application copying 8MB of source files; and `configure` is an extremely I/O and `fork`-intensive 3000-line Unix shell script. All tests were performed on a networked machine running in multi-user mode with a full complement of daemons but no other user activity, representative of a typical single-user workstation.

For three of the applications in Table 3, `gzip` (best-case), `gcc` (average), and `configure` (worst-case), Figure 10 shows the overall application slowdown that would result if each normal process-to-process context switch was n microseconds more expensive, where n varies from 1

	`gzip`	`gcc`	`tar`	`configure`
Run time (sec)	26.4	35.3	9.6	26.0
Context switches/sec	11	32	81	202
Traps/sec	10	562	22	3470
System calls/sec	23	651	517	1807
Device interrupts/sec	427	509	3337	1055

Table 3: Scheduling-related statistics for test applications measured under FreeBSD-2.1.5 on a 100MHz Pentium PC.

to 1000 (1 ms). This graph effectively shows the "tolerance" of a system to scheduling overhead in different situations: for example, supposing that typical users would tolerate no more than about 2% slowdown for typical applications such as `gcc`, our framework (or any other scheduling mechanism) would have to add no more than about $300 \mu s$ to the process-to-process context switch time if implemented in FreeBSD. Suppose FreeBSD was changed so that *all* scheduling was done in user mode, adding approximately one additional context switch due to a scheduler invocation for each existing process-to-process context switch. Based on context switch times we measured using the `lmbench` benchmark suite[24], which are approximately 39μs on this machine, plus an additional 11μs to reflect the dispatcher's cost (Section 5.7.1), the overall overhead should still be negligible simply because FreeBSD does not context switch all that often (see arrow A on the graph). Furthermore, given BSD's monolithic design, we would in practice expect at least the root scheduler to be implemented in the kernel and only application-specific schedulers to be in user mode; this would reduce the cost even further.

Of course, microkernels have much less tolerance for scheduling overhead simply because they perform more context switches. For example, in Mach, traps and system calls can be expected to produce about two additional context switches each; for an extremely "purist" microkernel such as L4, in which even device drivers are in user mode, device interrupts would also add an additional two context switches each. Figure 10 also shows a plot of scheduling overhead tolerance for a hypothetical L4-like microkernel, based on the numbers in Table 3 except with traps, system calls, and interrupts each counting as two additional context switches. For example, to keep the overhead under 2% for `gcc` in this system, the additional per-context switch cost must be no more than about 6μs (see arrow B), which would be difficult even with L4's phenomenal 3.2μs round-trip RPC time[20]. On the other hand, these "back of the envelope" calculations are pessimistic in several ways: first, an additional scheduler invocation should not be needed for every normal context switch, as explained in Section 5.7.2; second, in a microkernel that dispatches hardware interrupts to threads, we would expect at least a minimal fixed-priority root scheduler to be implemented in the kernel so that device driver threads can be scheduled without the help of user-level schedulers; and

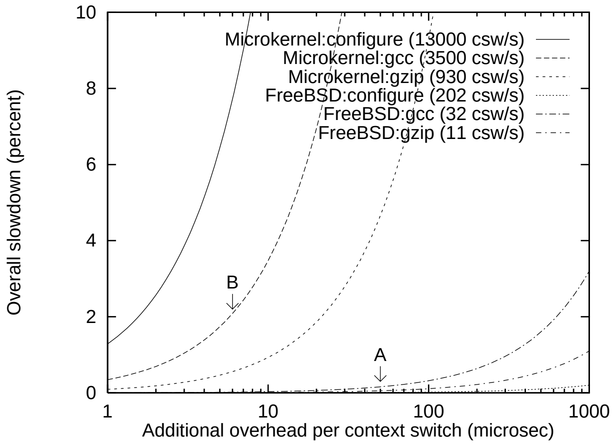

Figure 10: Overall application slowdown as a function of additional overhead per process-to-process context switch. In BSD, only switches between Unix processes count; in Mach, traps and system calls count as RPCs (two context switches each); in L4, device interrupts also count as RPCs.

third, our prototype dispatcher is extremely unoptimized in terms of both its algorithm (it still causes many avoidable context switches) and its implementation (it takes much longer than necessary to compute the next thread to run). There should certainly be many points in the design space at which CPU inheritance scheduling is practical even for microkernels; we are currently working on a second prototype of the framework in our Fluke microkernel[10, 11] in order to evaluate the framework further in this light.

5.8 Code Complexity

As a final useful metric of the practicality of our framework, we measured our prototype's code size and complexity in terms of both raw line count and lines containing semicolons. The entire dispatcher is contained in a single well-commented file of 550 (raw) or 158 (semicolons) lines. Each of our example schedulers is around 200/100 lines long; of course, production-quality schedulers that handle processor affinity, scheduler activations, dynamic priority adjustments, and so on, would probably be significantly bigger.

6 Conclusion

In this paper we have presented a novel processor scheduling framework in which threads are scheduled by other threads. Widely different scheduling policies can coexist in a single system, providing modular, hierarchical control over CPU usage. Applications as well as the OS can implement customized local scheduling policies, and CPU resources consumed are accounted for and attributed accurately. The framework also cleanly addresses priority inversion by providing a generalized form of priority inheritance that automatically works within and among multiple diverse scheduling policies. CPU inheritance scheduling extends naturally to multiprocessors, and supports processor management techniques such as processor affinity and scheduler activations. We have shown that this flexibility can be provided with negligible overhead in environments in which context switches are fast, and that the framework should be practical even in some kernel environments in which context switches are more expensive.

Acknowledgements

For their many thoughtful and detailed comments on earlier drafts we thank the anonymous reviewers and Kevin Jeffay, our shepherd, as well as the members of the Flux project. We are especially grateful to Jay Lepreau for his support and advice, to Kevin Van Maren for considerable help on the results and bibliography, and to Mike Hibler for spending an entire day reviving Bryan's machine after a complete disk failure on Friday the 13th just before the final paper deadline.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of the Summer 1986 USENIX Conf.*, pages 93–112, June 1986.

[2] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and Language-Independent Mobile Programs. In *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation*, pages 127–136, May 1996.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, Feb. 1992.

[4] D. L. Black. *Scheduling and Resource Management Techniques for Multiprocessors*. PhD thesis, Carnegie Mellon University, July 1990.

[5] A. C. Bomberger and N. Hardy. The KeyKOS Nanokernel Architecture. In *Proc. of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Seattle, WA, Apr. 1992.

[6] O.-J. Dahl. Hierarchical Program Structures. *Structured Programming*, pages 175–220, 1972.

[7] S. Davari and L. Sha. Sources of Unbounded Priority Inversions in Real-time Systems and a Comparative Study of Possible Solutions. *ACM Operating Systems Review*, 23(2):110–120, Apr. 1992.

[8] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, Dec. 1995.

[9] R. B. Essick. An Event-based Fair Share Scheduler. In *Proc. of the Winter 1990 USENIX Conf.*, pages 147–161, Washington, D.C., Jan. 1990.

[10] B. Ford, M. Hibler, and Flux Project Members. Fluke: Flexible μ-kernel Environment (draft documents). University of Utah. Postscript and HTML available under http://www.cs.utah.edu/projects/-flux/fluke/html/, 1996.

[11] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.

[12] D. B. Golub. Adding Real-Time Scheduling to the Mach Kernel. Master's thesis, University of Pittsburg, 1993.

[13] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, 1996. Available as http://java.sun.com/doc/language_environment/.

[14] P. Goyal, X. Guo, and H. M. Vin. A Hierarchical CPU Scheduler For Multimedia Operations. In *Proc. of the Second Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996. USENIX Assoc.

[15] N. Hardy. The KeyKos Architecture. *Operating Systems Review*, Sept. 1985.

[16] G. J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8), Oct. 1984.

[17] Institute of Electrical and Electronics, Inc. *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Programming Interface (API) — Amendment 1: Realtime Extension [C Language]*, 1994. Std 1003.1b-1993.

[18] E. D. Jensen. A Benefit Accrual Model of Real-Time. In *Proc. of the 10th IFAC Workshop on Distributed Computer Control Systems*, Sept. 1991.

[19] J. Kay and P. Lauder. A Fair Share Scheduler. *Communications of the ACM*, 31(1), Jan. 1988.

[20] J. Liedtke. On Micro-Kernel Construction. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, Dec. 1995.

[21] J. Liedtke. A Short Note on Cheap Fine-grained Time Measurement. *ACM Operating Systems Review*, 30(2):92–94, Apr. 1996.

[22] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[23] J. M. McKinney. A survey of analytical time-sharing models. *Computing Surveys*, page 105"116, jun 1969.

[24] L. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of 1996 USENIX Conf.*, Jan. 1996.

[25] Microsoft Corporation. *Win32 Programmer's Reference*, 1993. 999 pp.

[26] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[27] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[28] J. Torrellas, A. Tucker, and A. Gupta. Evaluating the Performance of Cache-Affinity Scheduling in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 24:139–151, 1995.

[29] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. In *Proc. of the 13th ACM Symp. on Operating Systems Principles*, pages 26–40, Oct. 1991.

[30] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Sept. 1995.

[31] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proc. of the First Symp. on Operating Systems Design and Implementation*, pages 1–11, Monterey, CA, Nov. 1994. USENIX Assoc.

[32] C. A. Waldspurger and W. E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Report MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995.