D R A F T — UUCS-96-005

Flexible Multi-Policy Scheduling based on CPU Inheritance

Bryan A. Ford   Sai R. Susarla

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

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

May 10, 1996

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 multiple diverse scheduling policies. CPU inheritance scheduling extends naturally to multiprocessors, and supports processor management techniques such as processor affinity [7] and scheduler activations [1]. Experimental results and simulations indicate that this framework can be provided with negligible overhead in typical situations, and fairly small (5-10%) performance degradation even in scheduling-intensive situations.

1 Introduction

Traditional operating systems control the sharing of the machine's CPU[1] 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 (non-degrading) threads [**?**, **?**], or several "scheduling classes" to which threads with different purposes can be assigned (e.g. real-time, interactive, background). [**?**]. However, even these variants are generally hard-coded into the system implementation and cannot easily be adapted to the specific 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 other 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. Scheduler threads can be notified when the thread to which they donated their CPU time no longer needs it (e.g., because the target thread has blocked), so that they can reassign their CPU to other target threads. 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

[1] We use the terms CPU and processor synonymously.

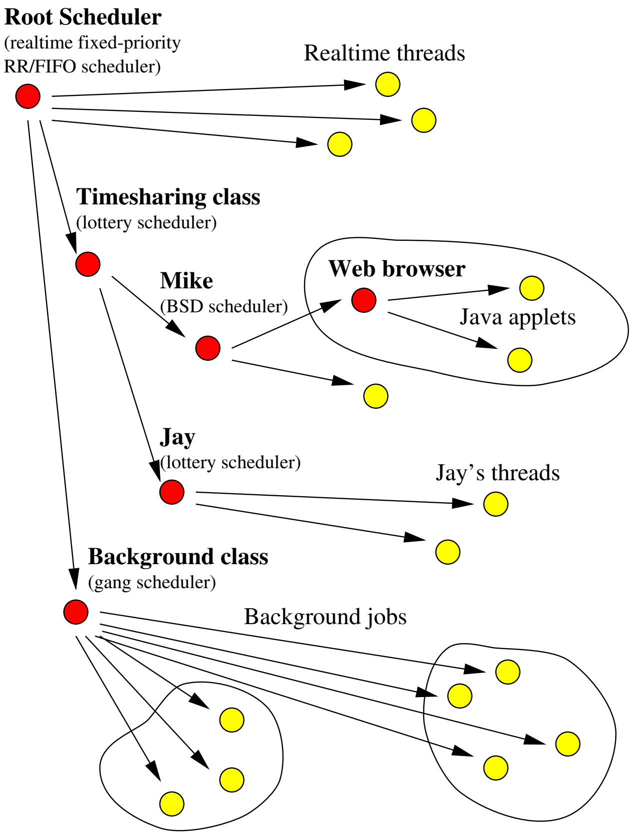

Figure 1: Example scheduling hierarchy. The red (solid) circles represent threads acting as schedulers, while the yellow (open) circles represent "ordinary" threads.

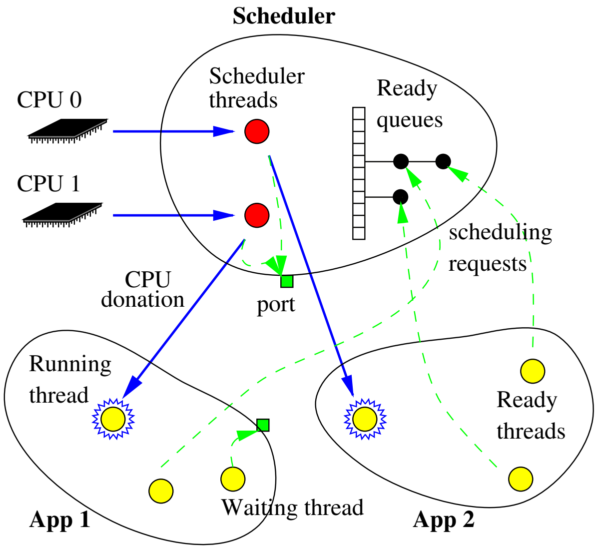

Figure 2: Example fixed-priority scheduler

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 so that it can reassign the CPU; on the other hand, if an 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, rate monotonic, and lottery/stride scheduling, can be implemented in the same way.

We believe this scheduling framework has the following benefits:

- Provides coherent support for 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.
- 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.
- Usage of CPU resources can be accounted for to various degrees of accuracy depending on performance tradeoffs.
- Addresses priority inversion naturally, without the need for explicit priority inheritance/ceiling protocols, in the presence of contention for resources.
- CPU use is attributed properly even in the presence of priority inheritance.
- Naturally extends to multiprocessors.
- Supports processor affinity scheduling.
- Allows scheduler activations to be implemented easily.

The rest of this paper is organized as follows...

2 Motivation

Traditional operating systems control the sharing of the 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. For hard real-time applications, meeting application-specific deadlines is a requirement, 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; that evolved to multiprogrammed machines with batch job loads, which could still be handled by a single scheduling policy. 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 usually adequate.

Today, we are entering what appears will be a long period of needing multiple scheduling policies. Multimedia content drives the need for an additional scheduling policy on general purpose workstations: soft real-time. Untrusted executable content (e.g., Java applets) will require policies which limit resource use while also providing soft real-time guarantees. Concomitantly, the hard real-time domain is also making inroads onto general purpose machines, due to two factors: processors and instruments supporting embedded applications are becoming networked, and some customers, e.g., the military, want the ability to flexibly shift processing power to the problem of the moment.

Hence, as the diversity of applications increases, operating systems need to support multiple coexisting processor scheduling policies, in order to meet individual application's needs as well as to utilize the system's processor resources more efficiently.

2.1 Related Work

One relatively 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 systems, Mach [?], and Windows NT [?], and has even become part of the POSIX standard [?], 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 adjustment mechanisms on threads that are 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-realtime threads execute at lower priority levels, it is possible to obtain real-time processor scheduling behavior sufficient for some real-time applications. However, it is well-known that this approach has serious shortcomings: in many cases, entirely different non-priority-based scheduling policies are needed, such as rate monotonic, earliest-deadline-first, and benefit-accrual scheduling [?].

Furthermore, even in normal interactive or batch-mode computing, traditional priority-based scheduling algorithms are showing their age. For example, these algorithms do not provide a clean way to encapsulate sets of processes/threads as a single unit and isolate and control their processor usage relative to the rest of the system. This lack opens the system to various denial-of-service attacks, the most well-known being the creation of a large number of threads which conspire to consume processor resources and crowd out other activity. These vulnerabilities generally didn't cause serious problems in the past for machines only used by one person, or when the users of the system fall in one administrative domain and can "complain to the boss" if someone is misusing the system. However, as distributed 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 to be downloaded and run in a supposedly secure environment such as that provided by Java [?] or OmniWare [?]. Schedulers have been designed that promise to solve this problem by providing flexible hierarchical control over CPU usage at different administrative boundaries [2, 8]. However, it is not yet clear how these algorithms will address other needs, such as those of various types of real-time applications: certainly it seems unlikely that a single "holy grail" of scheduling algorithms will be found that suits everyone's needs.

With the growing diversity of application needs and scheduling policies, it becomes increasingly desirable for an operating system to be able to support multiple completely different policies. On multiprocessor systems, one simple but limited way of doing this is to allow a different scheduling policy to be selected for each processor [?]. A more general but more controversial 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.

An additional problem with existing multi-policy scheduling mechanisms is that all of them, as far as we know, still require every scheduling policy to be implemented in the kernel and to be fairly closely tied in with other kernel mechanisms such as threads, context switching, clocks, and timers.

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. With traditional priority-based scheduling algorithms, this problem can be addressed with priority inheritance [3, 4], wherein the thread holding up the service is made to inherit 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 [8]. However, the problem of resolving priority inversion between threads of different scheduling classes using policies with completely different and incomparable notions of "priority," such as between a rate-monotonic realtime thread and a timeshared lottery scheduling thread, has not been addressed so far.

3 CPU Inheritance Scheduling

This section describes the CPU inheritance scheduling framework in detail.

3.1 Basic Concepts

In our scheduling model, as in traditional systems, a *thread* is a virtual CPU whose purpose is to execute arbitrary 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, depending on the scheduling policies involved. (For the purposes of this framework, it is not important whether these threads are kernel-level or user-level threads, or whether they run in supervisor or user mode.)

The basic idea of CPU inheritance scheduling is that unlike in traditional systems where threads are scheduled by some lower-level entity (e.g., a scheduler in the OS kernel or a user-level threads package), threads are instead 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 thread of its choosing, instead of using the CPU itself to execute instructions. 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 directly 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 whatever CPU resources it may have to other threads: it essentially distributes its own virtual CPU resources among some number of *client threads* to satisfy their CPU requirements. The client threads thus *inherit* some portion of the scheduler thread's CPU resources, and treat that portion as *their* virtual CPU for use in any way they please. These client threads can in turn act as scheduler threads, distributing their virtual CPU time among their own client threads, and so on, forming a scheduling hierarchy.

The only threads in the system that *inherently* have real CPU time available to them are the set of *root scheduler threads*; all other threads can only ever run if CPU time is donated to them. There is one root scheduler thread for each real CPU in the system; each real 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 scheduling policy* for that CPU.

3.2 The Dispatcher

In order to implement CPU inheritance scheduling, 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 handle threads blocking and unblocking, donating CPU to each other, relinquishing the CPU, etc., without actually making any real scheduling decisions. The dispatcher fields events and directs them to threads waiting on those events. 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 inherently contains no notion of thread priorities, CPU usage, or even measured time (e.g., clock ticks, timers, or CPU cycle counters). In an OS 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 is likely to be impractical for performance reasons).

3.3 Requesting CPU time

Since no thread (except a root scheduler thread) can ever run unless some other thread donates CPU time to it, the first job of a newly-created or newly-woken thread is to request CPU time from some scheduler. Each thread has an associated scheduler that has primary responsibility for providing CPU time to the thread. When the thread becomes ready, the dispatcher makes the thread "spontaneously" notify its scheduler that it needs to be given 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 and sends a notification to its scheduler port, that notification may in turn wake up a server (scheduler) thread waiting to receive messages on that 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 thread can then make a decision to re-run the preempted client thread, switch to the newly-woken client thread, or even some run other client 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.

3.4 Relinquishing the CPU

At any time, a running thread may block to wait for some event to occur, such as I/O completion.[2] 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.

[2] In our prototype implementation, a thread can only wait on one event at a time; however, there is nothing in the CPU inheritance scheduling framework that makes it incompatible with thread models such as that of Windows NT [?], in which threads can wait on multiple events at once.

3.5 Voluntary donation

Instead of simply blocking, a running thread can instead 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 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.

It is possible for a single thread to inherit CPU time in this way 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), it will automatically be switched to another if possible.

3.6 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, this `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 from a clock device driver 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 in some cases, 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 required in theory for the system to work. The following three sensitivity levels seem to be useful in practice:

- `WAKEUP_ON_BLOCK`: If the thread receiving the CPU blocks without further donating it, then the

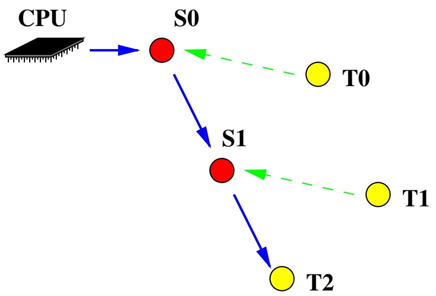

Figure 3: CPU donation chain

`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 (e.g., lower-priority) client threads waiting to run.

- `WAKEUP_ON_SWITCH`: If the client thread using the CPU (e.g., 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 client threads manually: for example, the scheduler may need to start and stop timers when switching between client threads.

- `WAKEUP_ON_CONFLICT`: As above, if T_2 blocks, the scheduler blocks too. However, in this case, if *any* client of scheduler S_1 is subsequently woken, such as T_1, the dispatcher passes control directly through to the woken client thread without waking up the scheduler thread. The scheduler is only awakened if a *second* client thread wakes up while the scheduler is already donating CPU to the first client (e.g., if both T_1 and T_2 become runnable at the same time). At this weakest sensitivity level, the dispatcher is allowed to switch among client threads freely; the scheduler only acts as a "conflict resolver," making a decision when two client threads become runnable at once.

4 Implementing High-level Schedulers

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

4.1 Single-CPU Schedulers

Figure **??** shows example pseudocode for a simple fixed-priority FIFO scheduler. The scheduler basically keeps a prioritized 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" wait-for-message operation instead of the `schedule` operation, to relinquish the CPU while waiting for messages. If there is only one client thread in the scheduler's queue, the scheduler uses the `WAKEUP_ON_CONFLICT` sensitivity level when running it to indicate to the dispatcher that it may switch among client threads arbitrarily as long as only one client thread attempts to use the CPU at a time.

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; then, client threads assigned to a given scheduler will effectively be bound to the CPU the scheduler is associated with. 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 must itself be multi-threaded. Assume for now that the scheduler knows how many processors are available, and can bind threads to processors. (This is clearly 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 the 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 general 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 inter-processor interrupts in traditional systems.

4.3.1 Processor Affinity

Scheduling policies that take processor affinity into consideration [5–7], 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 are indeed consistently run on the same processor. Any processor affinity support in one scheduling layer will only work well if all the layers below it (between it and the root scheduler) also pay attention to 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 agreed-upon interface. One particularly useful explicit client/scheduler interface is a *scheduler activations* interface [1], which allows clients to determine initially and later track the number of actual processors available to them, and create or destroy threads as appropriate in order to make use of all available processors without creating "extra" threads that compete with each other uselessly on a single processor.

Furthermore, since scheduler threads are notified by the dispatcher when a client thread blocks and temporarily cannot use the CPU available to it (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 highly useful to allow 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) exports a scheduler activations to its clients, 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 between each time quantum. In CPU inheritance scheduling, the precise nature of the timing mechanism available to schedulers 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 back to the scheduler so that it can account for the elapsed time and possibly 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, 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* [**?**]. 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 inexpensive in terms of overhead, since the scheduler generally wakes up on every clock tick anyway; however, it provides limited accuracy. A variation on this method that provides better accuracy at slightly higher cost is to sample the current thread at random points *between* clock ticks [**?**]. Alternatively, with time stamp-based accounting, the scheduler reads the current time at every context switch and charges the difference between the current time and the time of the last context switch to the thread that was running during that period. 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 an expensive operation.

In the root scheduler in a CPU inheritance hierarchy, both of the above 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 usage 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. (If this weren't the case, then the low-priority scheduler probably would not be able to run at all!) This assumption corresponds 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 some additional cost due to 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 4, 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 S_0 instead of S_1 which is the thread that is *actually* using the

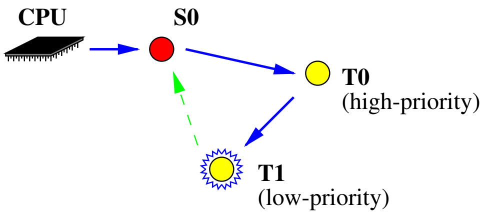

Figure 4: Implicit CPU donation from high-priority thread T_0 to low-priority thread T_1 to avoid priority inversion during a resource conflict.

CPU.

While it may seem somewhat nonintuitive at first, in practice this is precisely the desired behavior; it stems from the basic rule that with privilege comes responsibility. If T_0 is donating CPU to T_1, then T_1 is effectively doing work *on behalf of* T_0: i.e., finishing its job and unlocking the resource as quickly as possible so that T_0 can get on with its other activities. Since this work is being done (at this point) primarily for the benefit of T_0, the CPU time consumed must be charged to T_0. Demonstrated another way, charging T_1 rather than T_0 would be incorrect because it would allow the 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 generally 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 provided in the CPU inheritance scheduling framework in two ways. First, the effect can be achieved even in an implementation such as ours that only directly supports a single permanent scheduler association per thread, although in a somewhat ad-hoc and possibly inefficient way. First, the thread of interest is created and associated with its "primary" (presumably highest-priority) scheduler. Then, one additional thread is created for each additional scheduling policy desired; these threads then block forever on a lock held by the first thread so that they perpetually donate their CPU time to it. The dispatcher will automatically ensure that the primary thread always uses the highest priority scheduler available, because whenever the primary thread becomes runnable and requests CPU time from its scheduler, the secondary threads will also request CPU time from *their* schedulers, and the scheduling algorithms will ensure that the highest-priority request always "wins."

In situations in which this solution is not acceptable for reasons of performance or memory overhead, the dispatcher could fairly 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 and performance. The basic questions to be answered are:

- Is the framework *practical*? Can it perform the same functions as existing schedulers without unacceptable performance cost?
- Is the framework *useful*? Does it provide sufficient additional flexibility or functionality to justify its use in practice?

5.1 Test Environment

In order to provide a clean, easily controllable environment, as our initial prototype we implemented a simple user-level threads package incorporating CPU inheritance scheduling as its mechanism for scheduling the user-level threads it provides. 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

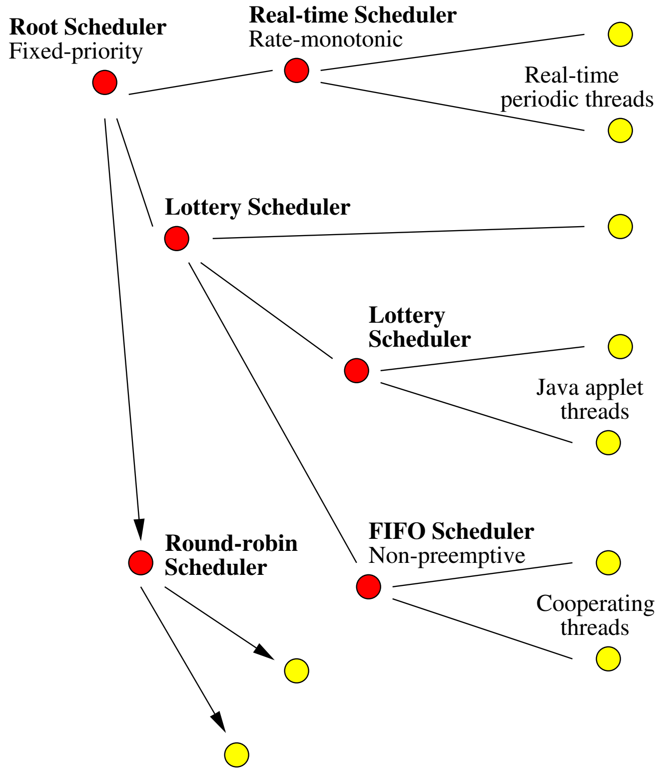

Figure 5: Multilevel scheduling hierarchy used for tests

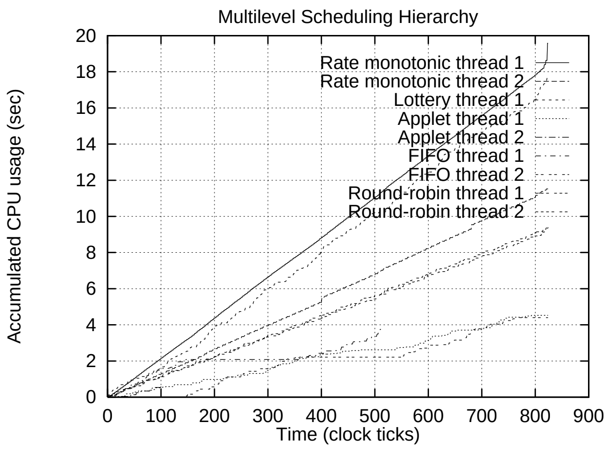

Figure 6: Behavior of a multilevel scheduling hierarchy

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, intended for practical use, the normal wall-clock timer would probably be used instead.)

Although our prototype is implemented in user space, the 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.

5.2 Scheduling behavior

Our first test demonstrates multiple scheduling policies stacked on top of each other. For this test we use the scheduling hierarchy shown in Figure 5, which is designed to reflect the activity that might be 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 is used to arbitrate 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, a second-level scheduler, also implementing lottery scheduling, manages two threads of a Java applet. (It is actually possible to collapse adjacent levels of lottery scheduling while achieving the same effect by using *currencies*; however, we show two separate schedulers here for generality.) Finally, an application-specific FIFO scheduler schedules two cooperating threads in a single application under the global timesharing scheduler.

Figure 6 shows the scheduling behavior of the threads simulated in this hierarchy.

5.3 Priority inheritance and priority-driven resource arbitration

To study how our scheduling mechanism tackles the priority-inversion problem, we implemented a simple application in which clients execute two services - one to look up the IP address of a machine given its name, which in turn contacts the second service that models a network access service. We implemented both servers as critical sections protected by "prioritized" mutex locks. These locks avoid

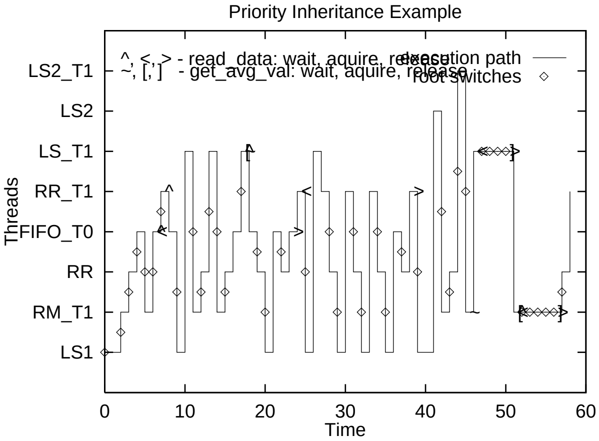

Figure 7: Priority inheritance between schedulers

unbounded priority inversion by donating the client's CPU to the lock holder while the lock is held. They also grant the lock in "priority order" by logically waking up all clients queued for the lock. Whichever client gets CPU first will succeed in acquiring the lock, while others continue donating CPU to the lock holder.

Figure 7 illustrates an observed execution sequence. Horizontal lines denote thread execution and vertical lines denote context switches. The smallest segment of execution denotes one clock period, at which time the root scheduler wakes up and reschedules. The sequence of events are as follows. The scheduling hierarchy used is shown in Figure 5.

1. First, the thread FIFO_T1 enters the `read_data` service's critical section by locking its mutex. starts executing.

2. Next the Roundrobin thread RR_T1 tries to acquire the above mutex and fails. It donates its CPU to the lock's current holder, FIFO_T1.

3. Next the lottery scheduled thread LS_T1 enters the first service which in turn tries to enter S2, but blocks, again donating CPU to FIFO_T0. (we call it `get_avg_val()`, or S1).

4. A little after 20 clock ticks, root scheduler runs the round robin scheduler, which in turn selects to run RR_T1. However, since RR_T1 has donated its CPU to the S2 lock holder, FIFO_T0. Hence FIFO_T0 gets CPU directly.

5. When FIFO_T0 eventually leaves S2 by unlocking its mutex, it logically wakes up and dispatches both of the contenders namely RR_T1 and LS_T1. The decision to choose RR_T1 is made by their common ancestor scheduler, which is the root. Hence RR_T1 enters S2 next. Note that its interleaved execution with LS1 causes its to immediately get back the CPU thru' LS_T1.

6. When the real-time thread RM_T1 eventually tries to enter S1, it finds that the holder of S1 is LS_T1 and donates it the CPU. This CPU is utilized by LS_T1 to rapidly finish its work in both S2 as well as S1, as RM_T1 is the highest priority thread in the system.

7. One important event not shown in the figure is the fact that LS2_T1 tries to enter S1 before RM_T1, and hence gets queued up before RM_T1 in the lock's wait queue. When LS_T1 unlocks S2's mutex, it wakes up both RM_T1 as well as LS2_T1. But due to its high priority, RM_T1 jumps the queue and enters S1 before LS2_T1. This is an example of prioritized granting of lock requests, between a lottery scheduled thread and a real-time thread with totally different notions of priority.

Thread	Level 0	Level 1	Level 2	Level 3
Root	**0.216**			
RM		**0.051**		
RM_T1			25.465	
RM_T2			14.978	
LS1		**0.082**		
LS_T1			23.080	
LS2			**0.020**	
LS2_T1				5.882
FIFO			**0.001**	
FIFO_T1				5.722
RR		**0.056**		
RR_T1			12.227	
RR_T2			12.219	
Total	0.216	0.189	87.990	11.604

Percent of time used by schedulers: 0.426
Percent of time not used by schedulers: 99.573

Table 1: CPU consumption by scheduler and worker threads

5.4 Scheduling overhead

Table 1 shows scheduling overhead in the test above. The total amount of time spent in each scheduler is shown in boldface, with the times for the corresponding threads to the right of their schedulers. It can be seen in the table that scheduling overhead is quite small compared to the time spent doing actual work: all of the scheduler threads combined consume only about 1% of the total CPU time.

6 Conclusion

In this paper we have presented 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. 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. 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 [7] and scheduler activations [1]. Experimental results and simulations indicate that this framework can be provided with negligible overhead in typical situations, and fairly small (5-10%) performance degradation even in scheduling-intensive situations.

References

[1] 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.

[2] 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.

[3] 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, April 1992.

[4] 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.

[5] 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.

[6] 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.

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

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