Slot Scheduling: General-Purpose Multiprocessor Scheduling for Heterogeneous Workloads

by

Brandon Hall, B.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

December 2005

Slot Scheduling: General-Purpose Multiprocessor Scheduling for Heterogeneous Workloads

Approved by
Supervising Committee:

____

____

Acknowledgments

This thesis has been the product of collaboration with several people. Above all, I would like to acknowledge the guidance I received from my advisor, Mike Dahlin. Mike provided significant direction on a number of key points in the evolution of this work, but most importantly, under his tutelage I have come to recognize what good systems research looks like and hopefully a little about how to communicate it and apply it. I feel truly privileged for the opportunity to work with him during my career as a graduate student.

More generally, I would like to thank the other faculty, students and staff of the LASR group for their advice and support during my studies at UT. I particularly had several helpful discussions with Ravi Kokku that shaped the development of the experimental analysis.

Anu Vaidyanathan and I worked closely together during various stages of the design and the K42 implementation, and I am grateful for both her considerable contributions and her persistence.

Finally, I would like to acknowledge the contributions of members of the K42 team, including Orran Krieger, Bryan Rosenburg and Bob Wisniewski. Via phone and e-mail discussions, they offered substantial insights on the design, as well as necessary guidance for the K42 implementation.

Brandon Hall

The University of Texas at Austin
December 2005

Slot Scheduling: General-Purpose Multiprocessor Scheduling for Heterogeneous Workloads

Brandon Hall, M.A.
The University of Texas at Austin, 2005

Supervisor: Michael D. Dahlin

This thesis presents slot scheduling, a approach to general-purpose CPU scheduling for multiprocessor systems. The chief virtues of slot scheduling are versatility (the ability to support a broad range of classes of schedulers) and intelligibility (allowing system designers to easily reason about interactions among different kinds of schedulers). In particular, slot scheduling is well-suited for meeting the needs of both real-time and gang-scheduled applications. These characteristics distinguish slot scheduling from existing scheduler proposals which tend to suffer from overspecialization and complexity. Slot scheduling achieves its goals by employing an explicit "bulletin board" representation of CPU allocation that decentralizes the task of scheduling from the kernel and permits scheduling logic to be carried out by applications themselves. We show that this approach is both feasible and efficient.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis presents *slot scheduling*, an approach to task scheduling for multiprocessor systems. Slot scheduling provides a simple but powerful framework for building a wide range of schedulers, including gang schedulers [12] and various types of real-time schedulers. In addition, it allows system designers to easily reason about how these different classes of schedulers will interact when they coexist in the same system. Both of these qualities—versatility and intelligibility—distinguish slot scheduling from existing scheduler design approaches, which tend to suffer from overspecialization and complexity. Thus, we speculate that slot scheduling should be applicable to a much larger set of computing workloads and platforms than previous approaches have been.

The ability to support a diverse mix of scheduling needs has become more and more important over the past decade. Individual systems are increasingly being asked to run applications with varying scheduling requirements, and run these applications at the same time. A typical example is the modern desktop computer, which may be used to play a video stream over the Internet while simultaneously burning a CD and compiling a program. This scenario requires balancing the needs of real-time, I/O and CPU-intensive processes. More sophisticated examples include the multiprocessor embedded systems found in medical, avionic, networking and other industrial environments. These systems may be asked to perform parallel scientific computations and provide timeliness guarantees to certain tasks, while factoring in the effects of temporary power-saving measures such as voltage or frequency scaling on particular processors. The challenges of combining such diverse application services within a single system's framework are immense, and not surprisingly, today's operating systems provide very limited scheduling support for such combinations of applications.

Lacking a general solution to all of these diverse scheduling needs, the systems industry has responded the demand with a proliferation of OS variants specialized for particular types of workloads. Notable examples are "real-time operating systems" such as QNX, VxWorks and RTLinux. We believe that this proliferation, while it satisfies the short-term needs of some classes of applications, is ultimately costly. The most obvious cost is the multiplication of development efforts that goes into creating and maintaining the numerous operating systems. However, a more significant cost is that which is borne by users of these systems. The absence of a single design point resulting from the multiplicity of systems limits the ability of software developers and system architects to reuse libraries and applications. At worst, solutions already developed on one platform cannot be translated to other platforms because the effort required is considered infeasible, or perhaps equally bad, the paths of development branch to reflect the zoo of system interfaces facing programmers.

To overcome these problems, we propose a significantly different paradigm—slot scheduling—that departs from the traditional design approaches in a few key respects. Perhaps the most important difference is that we eliminate the usual convention of a global, heavyweight scheduling algorithm that tries to coordinate the needs of all processes in the system. Instead, slot scheduling provides a thin kernel mechanism that allows the bulk of the scheduling logic to be distributed to library-level or application-level code. This mechanism consists of a globally-visible reservation system for CPU resources, in which processor time is discretized into fixed-length timeslices (*slots*). User-level code is given the responsibility of deciding which available slots would satisfy an application's particular scheduling needs and asking the kernel to run the application during those slots. With this framework, constructing a broad range of classes of schedulers becomes quite simple.

The rest of this thesis is devoted to explaining the slot scheduling approach in greater detail and defending its practicality as a solution to the scheduling challenges we have just mentioned. In Chapter 2, we walk through the main features of the slot scheduling architecture, showing how slot scheduling works and providing the rationale behind the various design decisions we made. Chapter 3 reports on our experiences with implementing slot scheduling for an actual operating system, K42. In Chapter 4 we analyze the feasibility and performance of the slot scheduling approach through a number of experiments. Chapter 5 situates our research contributions with respect to other recent work on scheduling in the systems community. We conclude our presentation in Chapter 6 and outline directions for future work and improvements. Our hope is that this thesis will clearly demonstrate slot scheduling's promise for tackling the challenges of the next generation of computing systems and for making these systems more widely accessible and usable than they are today.

Chapter 2

How it works

As we argued in the introduction, the general-purpose demands of emerging multiprocessor systems pose severe (we believe insuperable) challenges for traditional approaches to CPU scheduler design. The two main challenges of these systems are

(1) *versatility*: being able to support a large variety of schedulers (particularly gang and real-time), and

(2) *intelligibility*: enabling system builders to easily reason about interactions among different schedulers, decide how tradeoffs ought to be made (*policy* decisions), accurately predict "who will win" in the midst of competition for CPU time, etc.

Slot scheduling is motivated by the desire to address both of these challenges, and its architecture consequently reflects the design goals of versatility and intelligibility.

In order to achieve versatility, we do two things in slot scheduling. First, we break with the tradition of keeping all scheduling logic under tight wraps inside the kernel. Locking functionality away behind a kernel barrier is a hindrance to the flexibility and adaptability required by the systems we have described. Instead, we follow a design path laid out in recent work on extensible operating systems [2], [4], [6]: responsibility for process scheduling is shared between privileged kernel code and unprivileged user-space code, with the goal of empowering user code as much as possible and keeping kernel mechanism to a minimum. This strategy enables schedulers to be implemented as libraries which applications can pick and choose from at either compile time or runtime, and it avoids the need for patching the kernel in order to introduce novel types of schedulers.

The second thing we do to achieve versatility is create a primitive, "lowest common denominator" abstraction upon which we can build many different kinds of schedulers. The abstraction is this: we explicitly instantiate a schedule of CPU timeslices that precisely specifies when and where (on which processors) programs will be running in the future. Schedulers, operating as user-level code, scan the slot schedule and look for available timeslices that satisfy their scheduling requirements, and the kernel functions as a simple arbiter among the different user-level schedulers which are jockeying for CPU time. This approach differs markedly from those taken in existing scheduler architectures, architectures which rely on higher-order abstractions such as priorities, tickets [15], virtual time [8], [5], trees [7] or scheduling graphs [9]. These latter models have obvious benefits that have led to their repeated usage in system design: they present compact and algorithmically efficient ways of encoding future processor schedules, and they are typically good at achieving certain goals such as fairness. However, each of them in their own ways tends to restrict the range of possible scheduling behaviors and can be a clumsy fit for certain classes of schedulers. By mapping CPU schedules onto the most explicit representation possible—an actual schedule of which timeslices are going to whom—we can attain a greater degree of arbitrariness in how CPU time is distributed among running programs and consequently support a wider variety of schedulers.

The slot schedule representation also helps us achieve our other goal, intelligibility. The slot schedule functions as a globally visible "bulletin board" where applications can make their CPU time requirements known in an interpretable, easy-to-understand fashion. As other application schedulers enter the fray, they can clearly see what constraints they are operating under in terms of the CPU needs of the existing set of applications and can work around those needs as necessary, or if need be, force less privileged or overly greedy[1] applications to adjust their timeslice reservations in order to make room for the new application. These local scheduling algorithms make it easy to reason about what is happening in the system, and they lead to sensible interactions among different classes of schedulers. By contrast,

[1] In Section 2.4 we present some higher-level policy tools that can be used to define the CPU rights of various applications. These mechanisms allow administrators to define more precisely what it might mean for a process to be "more privileged" than another process or to be considered "greedy".

abstractions such as priorities or virtual time constitute relatively poor vehicles for reasoning about the runtime interactions of various schedulers. Precisely analyzing how the introduction of a real-time task would affect other deadline-based tasks in a system based on priorities, for example, is an extremely complicated undertaking, as attested to by the large body of real-time systems literature. In proposing the slot scheduling paradigm, we seek to eliminate this kind of inscrutability.

The remainder of this chapter is devoted to discussing the major features of the slot scheduling architecture and the rationale behind the various aspects of its design. We first describe the characteristics of the slot schedule itself, then explain how scheduling labor is divided between kernel and user-level code. Following that, we explore what happens when diverse schedulers compete with one another for CPU time in the slot scheduling framework and discuss some mechanisms for managing this competition. Next we turn to the issue of supporting best effort and event-driven types of jobs with an auxiliary scheduler, and finally we walk through an example demonstrating how all the pieces fit together.

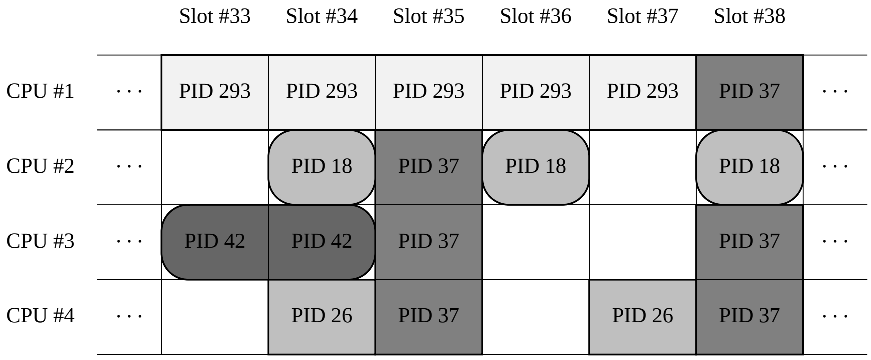
2.1 The slot schedule

At the center of our scheduling architecture is a global data structure which we call the *slot schedule*. The purpose of the slot schedule is to record and publicize decisions made in advance about which processes should be run on which processors at various times. The actual data structure we use to represent the schedule is a two-dimensional array, with rows corresponding to physical processors in the system and columns corresponding to consecutive timeslices.[2] Figure 2.1 illustrates part of a sample schedule for a four-way multiprocessor system.

As shown in Figure 2.2, each slot in the matrix records several pieces of information. The most important of these is the schedulee field, which names the process that should be run during that timeslice on the corresponding processor. The other fields will be discussed throughout the remainder of this chapter.

In our proposed framework, the timeslices represented by slots are all of equal, fixed duration. Choosing this slot length is somewhat of an engineering decision that involves trading off schedule granularity and flexibility against the overhead of frequent context switching. Reasonable values for today's systems would probably

[2] Readers familiar with [12] will recognize this as the transpose of the Ousterhout matrix.



		Slot #33	Slot #34	Slot #35	Slot #36	Slot #37	Slot #38	
CPU #1	···	PID 293	PID 293	PID 293	PID 293	PID 293	PID 37	···
CPU #2	···		PID 18	PID 37	PID 18		PID 18	···
CPU #3	···	PID 42	PID 42	PID 37			PID 37	···
CPU #4	···		PID 26	PID 37		PID 26	PID 37	···

Figure 2.1: Partial view of CPU schedule

```
struct Slot
{
    pid_t    owner;                      // gets charged for this CPU resource
    pid_t    schedulee;                  // designated by owner, gets CPU time
    int      token_priority;             // what owner spent for this slot
    int      expiration_timestamp;       // how long this reservation is valid
    void     (*revocation_callback)(int row, int col);
                                         // notify on preemption?
}
```

Figure 2.2: Data structure representing an individual slot.

range from ones to tens of milliseconds. A conceivable alternative to discretizing processor time in this manner might be to use per-processor free list structures that allow CPU time to be divided up into arbitrary, variable-length blocks, similar to the way that memory allocation is handled in many systems. We have instead chosen the discrete slot approach because it provides a simpler programming interface for library-level schedulers. It also leads to a slot schedule data structure that is of constant and predictable size, which can be helpful for the memory-mapping techniques we suggest for sharing the schedule among multiple processes.

2.2 Kernel scheduler

Actually switching from one task to the next in a system that uses slot scheduling is the responsibility of a low-level kernel scheduler. The algorithm for this low-level scheduler is very simple. On each processor, a timer goes off at every slot boundary and wakes up a per-processor kernel scheduler. This kernel scheduler inspects its row of the schedule and examines the upcoming slot: if the designated new process is the same as the process that was running prior to interruption, the kernel simply resumes it where it left off; otherwise, the kernel saves the state of the interrupted process and loads the context of the new process onto the machine. In either case, the entire procedure only involves a single lookup into the slot schedule and its running time should therefore be bounded by a constant. For most implementations, this code path will be very efficient and nearly optimal in terms of the overhead that any scheduling algorithm would have to incur in order to perform a task switch.

Because system resources are finite and preclude laying out the CPU schedules indefinitely far into the future, we let the kernel scheduler walk through the slot schedule arrays in round-robin fashion, going back to the first slot after the last slot has been scheduled. In addition, the kernel interprets the slot schedule as being *periodic*: slot reservations made by processes persist through repeated cyclings of the schedule. For example, in a CPU array that is 100 timeslices long, a process claiming Slot #42 will be scheduled by the kernel at timeslices 42, 142, 242, etc.

The alternative to periodicity would be to have slot reservations expire immediately after their use, thereby making the slots available again to other processes that want to be considered during the kernel's next pass over the schedule. While this nonperiodic approach would appear to have the advantage of offering greater adaptability to changing or unstable workloads, we have rejected it because it requires ongoing recomputation of the schedule even when the system is in a steady state. Instead, we offer processes the option of tagging their reservations with an expiration timestamp: the kernel scheduler looks at this timestamp in order to decide whether a reservation is currently valid or not. Our slot scheduling design is hence well-suited for workloads consisting of highly periodic tasks (like are found in certain real-time scenarios) and/or aperiodic tasks (such as event-driven types of jobs).

Although here we have described schedulees as processes, the actual unit of scheduling may vary from one scheduling architecture to the next. In particular, many operating systems may wish to single out individual threads or thread schedulers as the entities that are named in individual slots. As we describe in Chapter 3, the slot scheduler we implemented for the K42 operating system specifies schedulees in terms of resource domains, a K42-specific scheduling primitive. A slot scheduler implementation for Linux would probably deal in terms of Linux's LWPs (lightweight processes).

2.3 Library-level schedulers

Up to now we have discussed the slot scheduling framework mainly in terms of the low-level mechanisms required for its functioning: the slot schedule itself and the supporting kernel infrastructure. These mechanisms are fairly basic and by themselves do not provide any interesting scheduling behavior for applications. It is the role of user-level code to deliver higher-level guarantees, such as timeliness, fairness or coscheduling, to applications using these kernel services.

All user-level schedulers will follow the same general algorithm, which proceeds as follows. First, the scheduler has to examine the slot schedule to determine which timeslices are currently available for it to take. Then it must decide which of the available slots it actually wants to take in order to satisfy its particular scheduling requirements. Finally, it must ask the kernel to reserve these slots for the application. If each of these stages completes successfully, the user-level scheduler's work is done and it can idle until the application's requirements change. Otherwise, it may have to revisit the slot schedule to see if there is another way to satisfy the demands, and if not, possibly inform the application of the inability to meet its scheduling requirements.

Because user-level schedulers will need a quick and efficient means of reading the schedule, we recommend that slot scheduling implementations use shared memory techniques to map the schedule into a read-only segment of each process's address space. An implementation for Linux could simply create a new entry in the `/proc` filesystem, for example.

The API for allowing user-level code to modify the state of the schedule is shown in Figure 2.3. It consists of two system calls, one for requesting slots from the kernel (claim_slots) and another for releasing slots that are no longer needed (free_slots). Each individual slot request specifies several things: who is to be scheduled (schedulee), what level of token the caller wishes to spend for this CPU resource (token_priority), whether lifetime of the CPU claim should be limited (expiration_timestamp), and an optional handler for notifying the caller in case the claim is revoked due to external events (revocation_callback).

```
err_t claim_slots(SlotRequest *request_vector, int *len);
```

> Stakes claims on CPU timeslices or changes parameters for existing claims. Caller passes in list of slot requests; kernel returns list of requests that were successful and adjusts length parameter appropriately.

```
void free_slots(SlotRequest *request_vector, int len);
```

> Releases already claimed CPU timeslices.

```
struct SlotRequest
{
    int      row;                  // which processor
    int      column;               // which timeslice
    pid_t    schedulee;
    int      token_priority;
    int      expiration_timestamp;
    void     (*revocation_callback)(int row, int col);
}
```

> Data structure for specifying parameters of claims. All of the fields are significant for the claim_slots() system call; for the free_slots() call, the kernel cares only about the row and column fields.

Figure 2.3: Scheduling system calls

Note that we make a distinction between the process which owns a slot (the owner field) and the process which is designated to receive the CPU time (the schedulee field). The process calling claim_slots() or free_slots() is implicitly identified as the owner and can name any process it likes as a schedulee. The purpose of this distinction is to facilitate collaborative types of scheduling among multiple processes: for instance, a group of processes that wish to gang schedule their execution.

Typically such processes will centralize the logic in a designated coordinating process, who takes responsibility for assigning timeslices to all the participants. Since we do not place any restrictions on who owners may nominate for CPU time, as a necessary security measure, the system charges owners, not schedulees, for the use of slots. If it were not for this, a malicious or buggy process could unfairly deplete the token resources of other processes by designating them as schedulees against their will. (Tokens and resource accounting are discussed more fully in Section 2.4.1.) Another interesting use for the owner / schedulee distinction is in solving problems of priority inversion: a high-priority client waiting on a contended resource can temporarily slot schedule the server in order to hasten processing.

When a process asks for slots via the claim_slots() call, it is possible that some of its requests may be denied, perhaps because of insufficient resource rights or other unknown constraints. In this case, the kernel will follow a best-effort approach and attempt to claim as many of the requested slots as it can for the caller; the kernel then returns an error code and sends the caller a list of only the successful claims, via the same buffer that the caller used to send the initial request list. By doing things in this way, we make it easy for the scheduling client to free slots *en masse*: the client can simply send back the same list in free_slots() that the kernel had returned from the earlier call to claim_slots().

Having spelled out this framework, we can now easily describe library-level implementations for some common classes of schedulers:

Periodic real-time Following the original model of Liu and Layland [11], these are tasks with requirements like "give me m milliseconds of CPU time out of every n milliseconds". The library scheduler for these tasks simply scans the schedule looking for at least m milliseconds worth of slots in each n-millisecond window. Other variants might allow some degree of jitter, job execution requirements permitting.

Event-driven/aperiodic real-time These are tasks that wait for an event to occur (like the arrival of I/O or the emptying of a buffer) and then require a certain amount of computation before a deadline, after which they can rest until the next event occurrence. The library-level scheduler for these tasks looks for the requisite number of slots between the current time and the deadline and tags them with the appropriate expiration timestamp. (We will say more about how event-driven tasks get the opportunity to schedule themselves in Section 2.5.1.)

Gang scheduled These are jobs involving k processes or threads that must run simultaneously in order to make forward progress. Their user-level scheduler simply looks for columns containing k available slots.

Cache-sensitive The scheduler looks for long contiguous blocks of slots on the same processor in order to minimize the number of context switches and maximize efficient cache usage.

Best-effort These tasks do not have any special scheduling requirements; their scheduler tries to obtain whatever slots it can get its hands on, working around the claims made by other, more demanding types of schedulers.

In most cases, processes will probably prefer their slots to all be on the same rows (processors), but it is important to note that the low-level slot scheduling infrastructure does not enforce this in any way. How to trade off concerns of utilization versus CPU affinity is a decision that is completely left up to library-level scheduler implementations. These implementations can elect to use either simple but cheap heuristics or deploy more complex but more expensive analytical algorithms.

One common task profile that we have omitted from this list of schedulers is that of an interactive task, i.e. a task that is I/O-bound, unpredictable and seeks low response time. The most obvious approach for slot scheduling such a task would be to look for slots which are as equidistantly-spaced as possible. However, workloads for desktop and server environments can consist of a large number of interactive tasks, and applying this kind of scheduling algorithm for each of them could be very uneconomical with regard to total slot usage. What we propose instead for interactive tasks is not to use slot scheduling at all but rather to address their scheduling needs with a secondary, fallback scheduler: this topic is covered more fully in Section 2.5.

We expect that production operating systems will generally want to supply a variety of stock library schedulers which provide higher-level scheduling behaviors and interfaces to programs, so that a process can request, say, certain types of real-time guarantees using a real-time API and not even be aware that slot scheduling was “under the hood”. Nevertheless, applications are always free to reach beyond any such APIs and directly carry out their own specialized scheduling algorithms using the slot scheduling system calls, if they so desire.

2.4 Interactions among competing schedulers

Understanding how the library-level scheduling algorithms we have just presented operate in isolation is fairly straightforward. However, the simultaneous and competing interaction of several such schedulers can be much more complex and harder to grasp, especially if the schedulers are of diverse types. In this section, we present a number of improvements to the basic slot scheduling architecture that enable reasoning about these interactions at a higher, easier-to-understand level. The goal of these enhancements is to provide designers and administrators with an effective means of managing the competition among different schedulers; essentially, a comprehensible way of articulating CPU sharing policy.

2.4.1 Tokens

A pure free-for-all, first-come-first-served approach for handing out slots to processes would present obvious problems: individual processes could monopolize the schedule and unfairly starve other processes, could make it impossible for other processes to meet real-time demands, etc. So, in order to restrict processes' access to the CPU and resolve competing claims for slot time, we introduce a *token* system. Tokens are the primary means of metering CPU resource rights for slot-scheduled applications and provide a way to constrain processes' CPU usage without diminishing the flexibility offered by the slot scheduling approach.

Tokens are an integral part of the scheduling procedure for applications and fit into the slot scheduling framework as follows. Each process, in order to obtain slots, must spend tokens that have been granted to it by the system. The system charges processes one token per slot for the privilege of ownership. The kernel acts a banker and keeps track of how many tokens every process currently possesses: whenever a process attempts to claim new slots via the claim_slots() system call, the kernel verifies that the process has sufficient token funds before granting the request and then decrements the process's funds accordingly. A process that has spent all of its tokens is unable to purchase any further CPU time in the slot schedule. Processes get their tokens back either when they release their slot claims voluntarily or when the system forcibly revokes the claims.

Thus, tokens provide one immediate way for administrators to define CPU sharing policy: by granting different amounts of tokens to different applications, based on their relative merit for CPU cycles. A process that has been given twice as many tokens as another process, for example, will be able to purchase twice as much processor time in the slot schedule and consequently get to run twice as much. Similarly, if the system wants to grant real-time privileges to a certain job but does not want it to acquire any CPU time beyond the minimum required to satisfy its timeliness demands, it can provide the job with just enough tokens to meet those needs and no more.

Following are some enhancements and further comments on the basic token scheme we have just described.

Priorities

In general, the system "sells" slots to processes on a first-come, first-served basis: an empty slot can be claimed by any process possessing at least one token, and once claimed will belong to that process until it chooses to give it up or until it is revoked. This simple system by itself, however, is not really adequate for balancing the needs of schedulers with CPU demands that vary in their stringency. A hard real-time job, for example, may desperately need certain slots that it cannot get because other applications with less exacting CPU requirements "got there first".

To fix this problem, we introduce the notion of *priorities* for tokens. Each token in the system has a fixed and unchangeable numeric priority attached to it. Processes can have tokens belonging to several different priority levels, and when a process purchases slots, it must name the priorities of the tokens it is choosing to spend (specified via token_priority). The key idea of priority is this: a process that spends a higher-priority token for a given slot will get to preempt any previous lower-priority claim made on that slot. The problem previously described can now be fixed by giving the hard real-time job higher-priority tokens than the other jobs in the system.

This priority enhancement thus provides us with a simple, high-level way to distinguish among applications with more or less strict scheduling needs. A likely usage scenario for priorities would be to define a few different priority levels and supply them to applications according to the following ranking: system critical, hard real-time, various classes of soft real-time, gang scheduled, best effort and background. Other usages or rankings are possible as well.

Weights

Although proportional sharing or fairness among applications can be achieved by carefully calibrating the number of tokens dispensed to them, such an approach is generally undesirable for a couple of reasons. First, it requires constantly monitoring the token levels of all processes in the system, raising or lowering these levels as processes join or leave. For example, if there are four equal jobs each in possession of 25% of the total available tokens, when a fifth job enters the system, the token funds of the first four jobs must be scaled down to 20% to maintain equity. This sort of constant monitoring of token funds may be burdensome when there are a lot of processes and the system is in a state of flux. Another problem with this approach is that it may unnecessarily constrain slot schedule utilization. In the foregoing example, if only one of the four jobs is currently using the slot schedule, that job will be limited to 25% of the slots and the remaining 75% will be unavailable to it, even though no other process might be using them.

To improve this situation, we allow a proportional-sharing weight to be specified for each token priority level of a process. This weight defines what a "fair share" of the number of slots is for that process, relative to other processes at the same priority level. With this parameter in place, a process P is able to preempt the claim of another process Q with an equal-priority token if Q currently has more than its fair share of slots relative to P. The previous dilemma can now be resolved by issuing a large number of tokens of equal weight to every job—even as many tokens as there are slots in the schedule. The first such job entering the schedule can claim as many slots for itself as it likes, so long as no other jobs have claimed them. When a second job wishes to enter the schedule, it will have the power to preempt up to half of the claims made by the first job and effectively split up CPU bandwidth 50-50 between the two processes. A third job entering the schedule can scale all the jobs down to 33% of the slots, and so forth. Most systems will probably want to apply this kind of token policy to their best-effort and background jobs (give them lots of tokens of equal weight) and supply lesser amounts of higher-priority, equal-weight tokens to the more critical classes of applications that we mentioned earlier (real-time, gang, etc.).[3]

Banking system calls

Figure 2.4 lists the basic system calls for token management. There is a method that enables ordinary processes to find out how many tokens they currently possess (knowledge which would be necessary for most kinds of scheduling algorithms); a privileged, administrative method for setting a process's current token levels; and a token donation method that allows processes to transfer tokens out of their own funds into the funds of other processes. This latter method can be useful for facilitating cooperative types of scheduling arrangements such the previously mentioned example of gang scheduling a group of processes. In that example, each of the participants in the group would donate the requisite number of tokens to the leader process, empowering it to buy slots on behalf of all the participants.

Final comments on tokens

There are two implementation alternatives for systems to consider with regard to token accounting: a system can issue tokens either on a per-process or a per-user basis. In the latter case, all the processes belonging to a given user will draw from a common pool of tokens when they make requests for slots, free slots, etc. While per-process token accounting yields a finer degree of control over CPU usage rights, it may be overly cumbersome for systems that mostly wish to enforce access control at the granularity of users. For these reasons, we opted for per-user token accounting in our K42 implementation described in Chapter 3.

We have made several allusions in this section to the "system supplying processes with tokens" but we have not been very specific about how this actually happens. Most systems will probably want some sort of minimal policy built into the kernel that automatically bestows a certain number of best-effort tokens upon newly-forked processes or newly-logged-in users. Specialized scheduling services,

[3] Hard real-time jobs may not be able to tolerate any slot preemptions at all, lest they miss critical deadlines. As a special case, it may be necessary to set aside a top-level token priority for these tasks that does not permit weighted sharing. Claims made with these tokens would then be nonpreemptable.

like real-time services or gang services, are probably best brokered by a privileged admission control server running in user-space that has permission to invoke the set_tokens operation and provide applications with the appropriate amounts and types of tokens, per whatever policy system architects decide is reasonable.

2.4.2 Revocations

Another dimension to interactions among schedulers in slot scheduling is the notion of *revocations*. Revocations of slot claims are a necessary evil that allow the system to adapt appropriately to changes in the types of CPU demands imposed on it. We have already noted a couple of ways in which a process's slot claims can be revoked: being preempted by a process with higher-priority tokens and exceeding one's fair share of slots for a given priority level. (A third revocation event—being evicted in order to reserve spaces for latency-sensitive tasks—is discussed in Section 2.5.1.)

We give library-level schedulers in slot scheduling a chance to respond to revocations by allowing them to specify callback handlers in the owner's address space (the revocation_callback field in a SlotRequest). These callbacks allow schedulers to repair any damage caused by the activity of their competitors. When a slot claim is revoked, the kernel will invoke this callback asynchronously and notify the owner which of its slots was taken away via the row and col arguments. Precisely how a scheduler chooses to respond to a revocation event (if it chooses to respond at all) will be scheduler-dependent. A gang scheduler, for instance, will want to find another slot in the same column to make up for the slot it lost, and if such a slot cannot be found, cancel the remainder of its slots in that column, possibly trying to reschedule itself in a new column. A real-time scheduler will want to find a makeup slot somewhere else within the same horizontal window of time; if some degree of jitter is allowed, it may also have the option of making up the lost CPU time in a location farther away. Best-effort and background applications might not care to respond immediately to revocation events and might instead opt to reëvaluate their scheduling status at some later juncture.

2.5 Fallback scheduling

Slot scheduling is a powerful tool for reconciling the complex and interlocking demands of different classes of schedulers. However, slot scheduling alone cannot suffice to meet scheduling needs of most real systems. In particular, there are two common situations arising in slot scheduling which the slot scheduler is not able to handle directly. The first situation concerns what scheduling decision is to be made when an empty slot comes up in the schedule, that is, a slot which has not been claimed by any process. The second situation is similar to the first and occurs when a nonempty slot comes up in the schedule but its designated schedulee currently happens to be unrunnable, perhaps because it is blocked on I/O, waiting for an alarm, etc. In both of these situations, since the slot schedule itself does not furnish a valid candidate for CPU time, we propose resorting to an auxiliary *fallback scheduling* algorithm that will select among the existing runnable processes in the system. The presence of the fallback scheduler ensures that CPU does not idle unnecessarily when there are processes waiting for CPU time.

From the slot scheduler's point of view, the concrete choice of which kind of scheduler to use for the fallback scheduler is not too critical; the best-effort or proportional sharing schedulers commonly deployed in existing operating systems would be suitable choices. However, from the point of view of overall system performance, choosing a scheduler that is good at balancing the needs of compute-bound versus I/O-bound processes (as many widely-used proportional sharing schedulers are) can be very beneficial. This is because, as we pointed out in Section 2.3, slot scheduling is not well-suited for dealing with large numbers of interactive tasks. The scheduling needs of these tasks are probably better addressed within the auxiliary framework of a fallback scheduler. In addition, best-effort or batch jobs may wish to avoid using the slot schedule altogether and rely instead on the fallback scheduler for their CPU time needs: this relieves them from the minor burden of locating and reserving slots for themselves, a task which would otherwise eat up their own CPU cycles as part of a library-level scheduling algorithm.

The necessary existence of a supporting fallback scheduler also eases the migration path over to slot scheduling for legacy systems: these systems' legacy schedulers can simply serve as the fallback schedulers in the newer slot scheduling framework. Existing applications need not even be aware of the slot scheduler's existence and can continue to run without changes and without relinking or recompilation. The cost of any desired per-program upgrades to slot scheduling can safely be deferred to a later date when it is reckoned to be economical.

Finally, we note a minor optimization to the kernel scheduling algorithm permitted by the addition of the fallback scheduler. Rather than wake up on every slot boundary, the kernel slot scheduler can set its timing interrupt to go off only at the beginning of the next nonempty slot, effectively sleeping for the intervening period of slot schedule nonactivity—the fallback scheduler will shepherd the existing runnable processes during this time. Implementations of slot schedulers must, of course, take care to reset the kernel timers appropriately if new jobs are inserted into the schedule during these empty blocks. With this enhancement in place, the slot scheduler can lay completely dormant when it is unneeded and thus becomes a cost-free addition to existing operating systems, with zero performance impact on non-slot-scheduled processes.

2.5.1 Reservations for latency-sensitive tasks

In systems based on precomputed allocation of processor time, as slot scheduling is, event-driven real-time jobs (and event-driven jobs more generally) face a predicament. These are tasks that require short-term QoS guarantees when certain events occur (arrival of I/O, buffer ready to fill, etc.). The problem for these jobs is that typically the event occurrences are unpredictable and preclude laying out a CPU schedule in advance. These jobs therefore must depend on the fallback scheduler to run them when the events occur; once they get a hold of the CPU, they can execute their library-level schedulers and temporarily book the upcoming slots they need. However, this strategy is prone to a serious risk: if the event occurs during a window of time in which the slot schedule is solidly booked by other processes, the newly-ready-to-run delay-sensitive job may not get the chance to carry out its scheduling logic until it is too late and subsequently fail to meet a computational deadline.

One solution to this problem would be for these latency-sensitive jobs to claim slots for themselves at small, regular intervals in the schedule. This kind of safety net would ensure that these jobs never have to wait too long before getting the opportunity to run their schedulers, i.e. provide a guaranteed lower bound on their latency. Unfortunately, this approach is undesirable because it rapidly depletes useful space in the slot schedule. Each such event-driven task would require its own exclusive scattering of slots throughout the schedule, slots which would likely go unused most of the time, causing poor utilization or schedulability.

What we advocate instead is a variation on this approach. Rather than each latency-sensitive task claiming its own separate set of slots, we propose that the system set aside a fraction of empty slots at regular intervals. Process scheduling during these slots will then be handled by the fallback scheduler. Assuming the fallback scheduler has a way of privileging or distinguishing certain processes, we can assign the latency-sensitive jobs high priority within the legacy scheduling framework and they thus will get to run first during the set-aside slots. This should provide these jobs with sufficient time to begin running their real-time schedulers, or at the very least run a quick-and-dirty algorithm that buys them some initial CPU time which they can use to execute a more complex scheduler. In effect, what we are doing with this solution is to make all the delay-sensitive jobs share the slots that they need in order to run their schedulers. While this solution may not be perfect for all situations, it should be a good compromise for many workloads consisting of delay-sensitive tasks, so long as the distribution of events over time is relatively smooth (not too many events piling up at once that would be competing for a handful of empty slots) and the schedule is not completely saturated.

Figure 2.5 shows the proposed administrative system call for defining the fraction of slots to be set-aside, on a per-processor basis. This method would likely be called by an admission controller or system monitoring daemon. In Section 4.3 we report on some experimental results which show the effectiveness of this system call as a tool for finely tuning the latency experienced by event-driven tasks.

2.6 Putting it all together

Having presented the full architecture of the slot scheduler, we now consider a brief, hypothetical computing scenario that demonstrates how the various pieces might fit together in an actual system. The system we will consider is a 4-way multiprocessor machine. Snapshots of the machine's schedule at various stages of execution are shown in Figure 2.6. The slots in the slot schedule are 10ms in duration, and the entire schedule is 120ms long. Operating behind the scenes is an admission control agent that decides what CPU privileges various processes are entitled to and provides them with tokens commensurate with those privileges.

Following is a history of the system workload corresponding to the different stages:

(i) Initially, the only job using the schedule is a gang-scheduled compute job A with three threads, running on CPUs #2, #3 and #4. A latency-sensitive-task reservation of one out of six slots is in effect on all processors, limiting process A to ten of the twelve columns.

(ii) A periodic real-time task B enters the system. This task requires 20 out of every 60 milliseconds of CPU time on Processor #3. The admission controller provides it with four high-priority tokens, which it uses to preempt some of A's claims.

(iii) Process A's gang scheduler is notified of the revocations on Processor #3. It reexamines the schedule and decides to migrate the thread running on CPU #3 over to CPU #1.

(iv) A latency-sensitive job C on CPU #1 that had been lying dormant is awakened by the arrival of an interrupt. It has to wait until an open slot comes up in the schedule before it has the opportunity to execute its scheduling logic. This opportunity occurs in the sixth column, at which time C temporarily reserves the next 40ms of processor time for itself, preempting process A's lower-priority claims.

(v) After job C has completed its processing, its slots expire and become available again to other processes. In particular, process A's revocation callbacks would have been triggered when it lost the slots earlier; a slightly sophisticated gang-scheduler implementation would probably have noticed that job C's claims were temporary and would have set a timer for itself to reclaim the slots after process C was done with them.

Though this example is admittedly a simple one, it shows how slot scheduling makes the problem of scheduling diverse tasks in a multiprocessor system quite tractable. Such ease of intuition is nearly impossible to come by in the other more conventional approaches to scheduling that we are aware of.

```
const int NUM_TOKEN_PRIORITIES;

struct TokenInfo
{
    // per-priority token parameters
    int     count;
    int     weight;
}

err_t get_tokens(pid_t process, TokenInfo tokens[]);
```

Looks up available token funds at the various priority levels for the named process.

```
err_t set_tokens(pid_t process, int priority, int count,
                 int weight, bool absolute);
```

Allows privileged user to tweak funds for an individual process. Flag absolute specifies whether the passed in count value is absolute or a delta change to the current count.

```
err_t donate_tokens(pid_t recipient, int priority, int count);
```

Transfers tokens of the specified priority from caller to recipient. If the caller has insufficient funds, all its tokens at that priority are transferred and an error is returned to the caller.

Figure 2.4: Banking system calls

```
err_t reserve_slots(int numerator, int denominator, int processor,
                    bool immediate);
```

Set aside numerator slots out of each consecutive window containing denominator slots on the named processor. If immediate is true, the reservation is enforced immediately, revoking existing claims as necessary; if immediate is false, the reservation is enforced lazily, waiting for current owners to free any excess slot claims of their own accord.

Figure 2.5: Reservation system call for delay-sensitive tasks

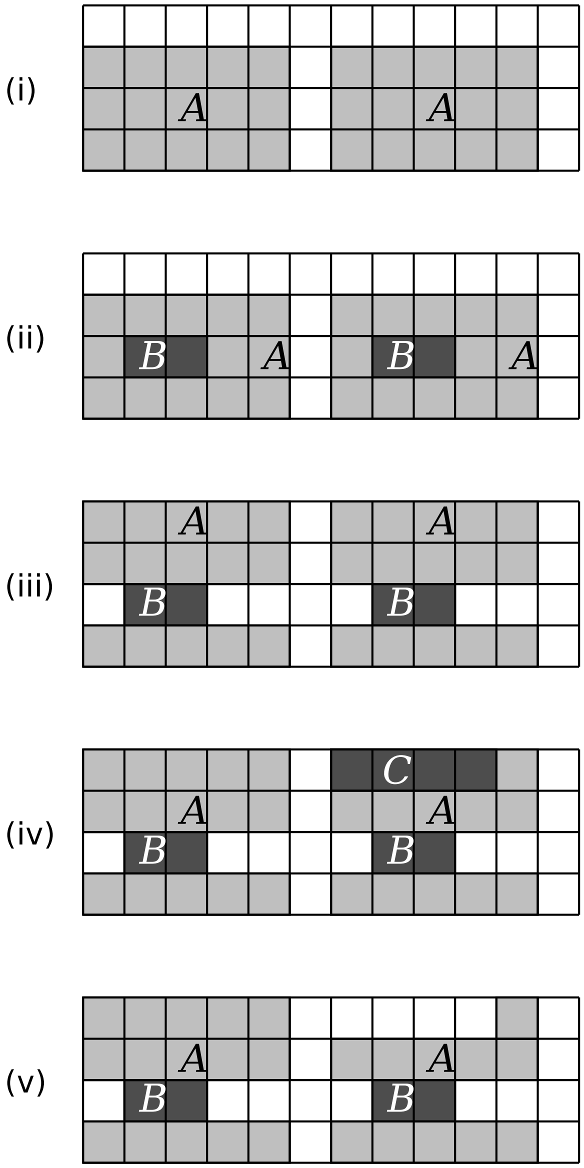


Figure 2.6: Various stages of a hypothetical slot scheduling scenario.

Chapter 3

Case study: K42

To validate our ideas, we implemented slot scheduling for an existing operating system, K42 [13]. K42 is a next-generation general-purpose OS for multiprocessor systems with the design goals of customizability and scalability, making it a good fit for our work. In this chapter, we give an overview of the basic K42 scheduling architecture [14] and describe how we modified it to support slot scheduling.

3.1 Scheduling in K42

K42 supports the common abstractions of processes and user-level threads common to most general-purpose operating systems: threads within a process run on different stacks but share the same address space, file descriptors, etc. For each process in K42, the kernel allocates a fixed number of kernel threads on each CPU that the process is using. Each of these kernel threads is associated with its own user-level thread scheduler that is responsible for some subset of the process's threads. In K42 jargon, these kernel threads, along with the supporting data structures for communicating with user-level schedulers, are called *dispatchers*.

Dispatchers in K42 serve a number of purposes. For each user-level thread scheduler, the system defines a number of entry points, and dispatchers vector various events to their thread schedulers via these entry points. For example, there are entry points for the dispatcher to jump to when the process is scheduled for execution, when a timer that the process has requested goes off, or when an asychronous IPC message arrives. Dispatchers can also reflect events like page faults or CPU exceptions back to the thread scheduler, rather than blocking in the kernel or killing the entire process. The state of any such trapping thread is passed back to the user-level scheduler (à la Scheduler Activations [1]), allowing it to take whatever action it deems appropriate, such as choosing a different thread to run until page fault I/O completes, etc. Process context switches are also handled cooperatively (similar to Exokernel [6]) by calling a special entry point that asks the scheduler to backup running thread states and voluntarily yield.[1] Figure 3.1 presents a slightly simplified illustration of the relationship between processes, dispatchers and thread schedulers. (The actual interactions between dispatchers and user-level schedulers are somewhat more complex than described here and are discussed in [14]).

Choosing which kernel threads (dispatchers) to run at context switch decision points is the responsibility of the K42 kernel scheduler. The kernel scheduling algorithm executes independently on each CPU and is hierarchical in nature. K42 groups dispatchers into containers called *resource domains*, and kernel scheduling policy is defined in terms of these resource domains. Each resource domain possesses a weight and quantum length, and the kernel uses a proportional sharing algorithm to select a resource domain to run. Once a resource domain has been selected, the kernel finally selects among the resource domain's dispatchers in round-robin fashion. Resource domains may contain dispatchers belonging to different processes but they may not contain dispatchers belonging to different users: this allows resource domain weights to operate as a means of enforcing fair sharing of the CPU among users.

Each resource domain additionally is assigned to one of several system-defined priority bands. Resource domains in higher priority bands always get chosen to run over resource domains in lower priority bands, as long as they are runnable. The K42 designers suggest the following semantic associations for the various priority bands:

- Level 0: system critical and hard real-time
- Level 1: gang-scheduled
- Level 2: soft real-time
- Level 3: general purpose
- Level 4: background

[1] Misbehaving schedulers that do not follow protocol are forcibly preempted by the kernel and can eventually be terminated.

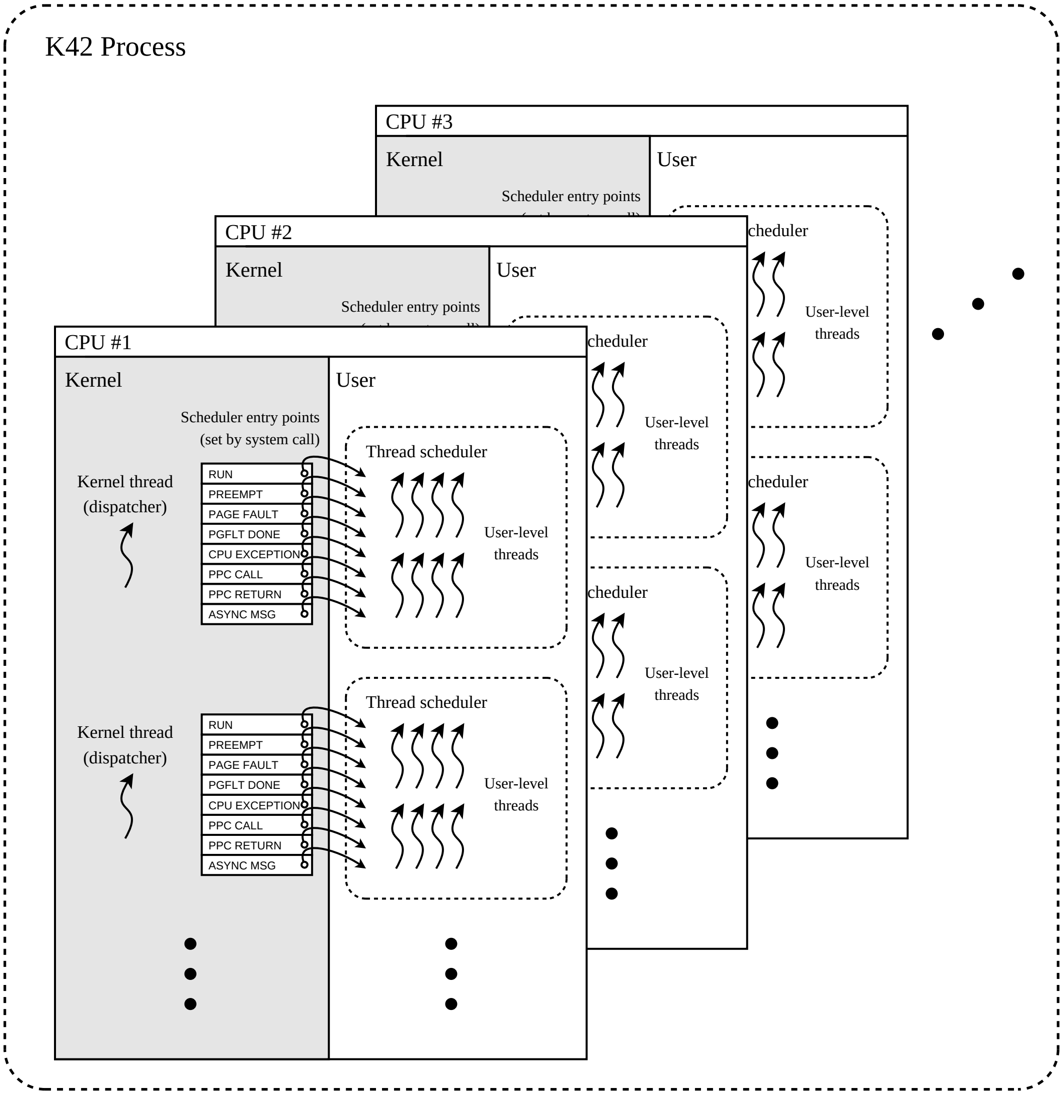


Figure 3.1: Diagram of a single process in K42.

K42 employs a microkernel approach to farm some privileged functionality out to user-space servers. One of these servers, the *resource manager*, is tasked with providing some of the higher-order scheduling behaviors and APIs for the system. Among its responsibilities are deciding when best-effort jobs should be migrated across CPUs to improve utilization, keeping track of the scheduling parameters for resource domains (weight/band/quantum length), and providing an API and services that allow administrators to control these parameters.

To summarize the K42 scheduler:

(1) At each scheduling decision point, the kernel chooses a resource domain to run, according to a proportional-sharing algorithm.

(2) The resource domain then selects one of its dispatchers to run, according to a round-robin scheme.

(3) The selected dispatcher transfers control to a user-level thread scheduler inside the process's address space.

(4) The user-level thread scheduler finally selects a thread for execution and restores its state to the machine.

3.2 Adding slot scheduling

Our strategy for incorporating slot scheduling into K42 was to leave the existing low-level scheduling machinery untouched and instead build the slot scheduler on top of the mechanisms already present in the kernel. While this design decision introduces some overhead as compared to a lower-level implementation, it significantly reduced the complexity of our code and consequently eased testing and debugging the slot scheduler. Experimental analysis in Section 4.1 showed that the amount of overhead was noticeable but still acceptable.

In our slot scheduling implementation, job scheduling is accomplished by the resource manager. We leverage K42's priority band structure and insert a new priority band (Level 1) just below the top-level system priority band (Level 0), as shown in 3.2. When a job's slot comes up in the schedule, the resource manager escalates it into to the Level 1 priority band, where it gets to run to the exclusion of all non-slot-scheduled jobs below it on that processor. At the end of the slot, the resource manager reawakens and restores the job back to its original priority band, repeating the procedure if necessary for any new job in the next upcoming slot.

Because our implementation makes use of the underlying K42 kernel scheduling machinery, slot schedulees are specified as resource domains rather than as process id's, since processes are not a fundamental unit of scheduling for K42. Applications request for slots to be assigned to particular resource domains and are responsible for ensuring that the named resource domains contain the dispatchers that they wish to have scheduled.[2] Unless an application truly wants to multiplex a slot among different processes, it has to make sure that the resource domain it has assigned to the slot contains only one dispatcher: the one it intends to receive CPU time.

The resource manager also maintains all the slot schedule data structures and keeps track of the token funds of all processes. It further provides the slot scheduling API for applications, using K42's *protected procedure call* (PPC) mechanism[3]: all of the slot scheduling system calls defined above are exported as PPC methods by the resource manager. Revocation callbacks for application-level schedulers are implemented as special C++ scheduler object methods, decorated with an `__async` keyword that instructs the K42 stub compiler to set up the underlying IPC machinery and hooks for the program. An application scheduler that relies on these callbacks must (as part of its initialization code) explicitly grant permission to the resource manager process to invoke these methods, since such invocations are a protected operation that require firing up a new thread in the application's address space.

Relying on K42's built-in scheduler to do most of the work, as our implementation does, is beneficial in a number of ways. The K42 priority band architecture guarantees that a slot-scheduled job will be running while it is runnable, since no other processes are assigned to the Level 1 band. Also, if a slot-scheduled job is running and needs to block temporarily in the middle of its slot, it can do so: the K42 scheduler will fall back to selecting among the other resource domains in the system. Should the blocked job become runnable again before the end of the slot, it will immediately be given back the CPU by virtue of its continued residence in the Level 1 priority band. Another benefit to our design is that it provides some

[2] K42 provides applications a means of specifying how their dispatchers should be distributed among the resource domains available to them.

[3] PPCs are a form of synchronous interprocess communication that have RPC semantics.

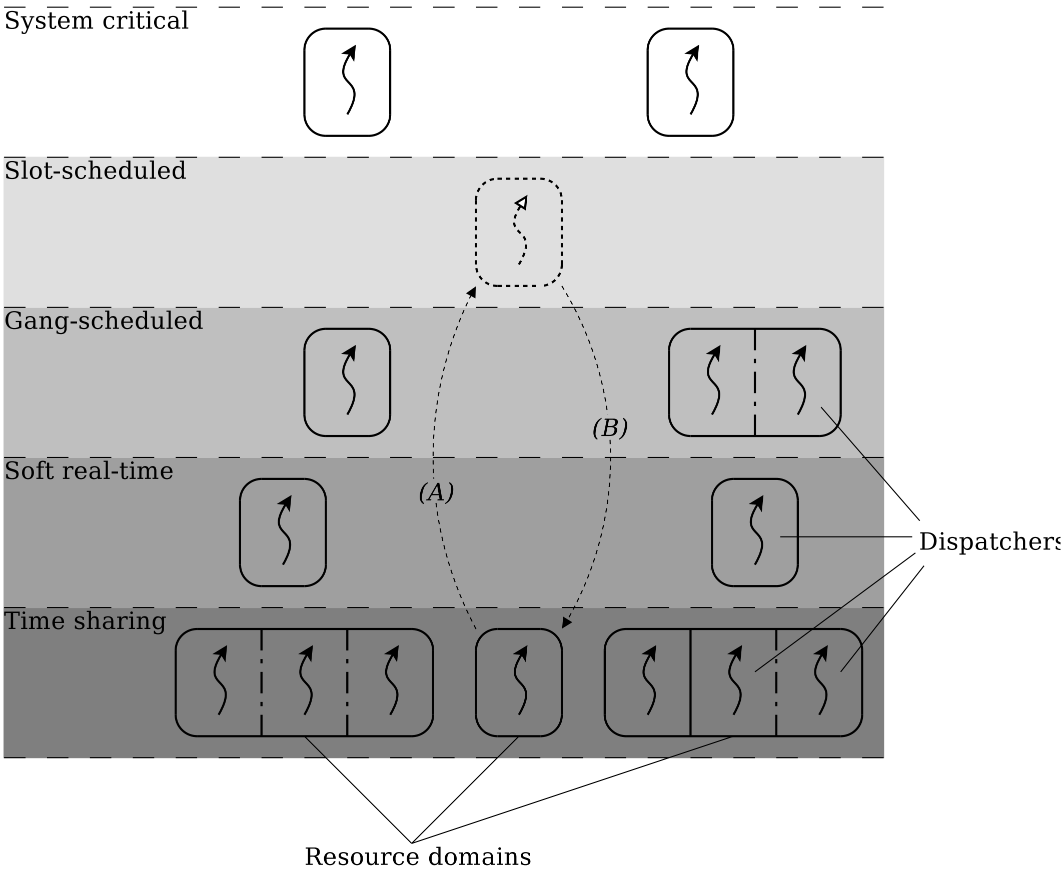


Figure 3.2: Illustration of how slot scheduling fits into the K42 architecture. When a resource domain's slot comes up in the schedule, the resource manager elevates it to the slot-scheduling priority band (A), where it will get to run to the exclusion of all the jobs below it. As soon as its slot is finished, the resource manager restores it back to its original priority band (B).

fairness for non-slot-scheduled, best-effort kinds of jobs in a natural sort of way: jobs that receive CPU time via the slot schedule are effectively penalized by the K42 proportional sharing algorithm and will become less likely than best-effort jobs to be scheduled during empty slots, giving those other jobs their due when the system has an opportunity to run them.

An important goal for any system scheduler is efficiency. In our K42 implementation, the slot scheduler only performs read-only operations on slots: the information dictating which resource domain is to be scheduled is confined to a 64-bit field that can be read atomically on the hardware we are using (PowerPC). This frees the resource manager from having to acquire locks in order to carry out the precomputed decisions contained in the slot schedule. Locks are still necessary, of course, for non-scheduling code paths that involve updating multiple fields in a slot structure, such as the methods for claiming or freeing slots. To achieve maximum concurrency in these situations, we use fine-grained, per-slot locks, and we also adjust users' token funds with atomic arithmetic instructions (FetchAndAdd) provided by the hardware, avoiding the need for software synchronization during token operations.

Chapter 4

Experimental evaluation

We evaluated the feasibility and performance of slot scheduling with a number of experiments described below. Most of these were online experiments and relied upon our K42 slot scheduling implementation. The K42 operating system itself was hosted on a PowerPC hardware simulator, Mambo [3]. Our final experiment was an offline analysis of the slot scheduling algorithms and did not use our K42 implementation.

4.1 Scheduling overhead

To measure the efficiency of the low-level slot scheduling operation, we calculated the time it took to switch from one slot-scheduled job to the next and compared this with the time required by the default K42 scheduler to perform a context switch. Table 4.1 shows the results, averaged over 100 switches.

The slot scheduling operation takes about three times as long as an ordinary task switch. The reason for the increase in overhead is primarily due to our design approach, which builds on top of the preëxisting low-level scheduling machinery. Switching from one slot-scheduled job to the next actually involves two K42 task switches: once to switch from the old job to the resource manager process, and a second time to switch from the resource manager to the new job. The resource

K42 scheduler	9μs
Slot scheduler	29μs

Table 4.1: Scheduling overhead

manager additionally must invoke some services from a master kernel process in order to raise and lower the priorities of the resource domains. A non-microkernel implementation of slot scheduling, more closely integrated with the kernel scheduling machinery, would no doubt reduce the gap in overhead significantly, at the cost of programming complexity.

However, we note that the current overhead is still quite small in absolute terms. When the slots are as small as 1ms, an application will only sacrifice 2% of its usable cycles to the slot scheduler, as compared to what it would ordinarily lose for a context switch, and this percentage drops even further as the slot length increases. By paying this 2% cost, the application gains a great deal: the ability to receive a wide variety of scheduling guarantees (real-time, gang, etc.). Hacking a conventional scheduler to provide all these services would almost certainly drive up the scheduling overhead as well. Furthermore, only slot-scheduled applications suffer the extra overhead; ordinary processes are still scheduled by the default K42 scheduler.

4.2 Proportional sharing

To verify that the K42 slot scheduler implementation was working correctly, i.e. provided applications with a predictable amount of CPU time, we ran a uniprocessor experiment involving two processes A and B. Both processes looped continuously incrementing a counter and were granted various quantities of slots (10ms duration) over a fifteen-second period. Figure 4.1 shows plots of the actual counter values and their rates of change over the fifteen seconds. At time $t = 0$, A and B were each assigned 40% of the total number of slots in the system. At time $t = 5$, each process was scaled down to 10% of the total slots. At time $t = 10$, process A was scaled back up to 40% of the total slots while process B's allocation was left at 10%. A third process consumed all of the remaining slots to prevent A and B from receiving additional CPU time via the fallback scheduler. The graphs show that the system behaved as expected.

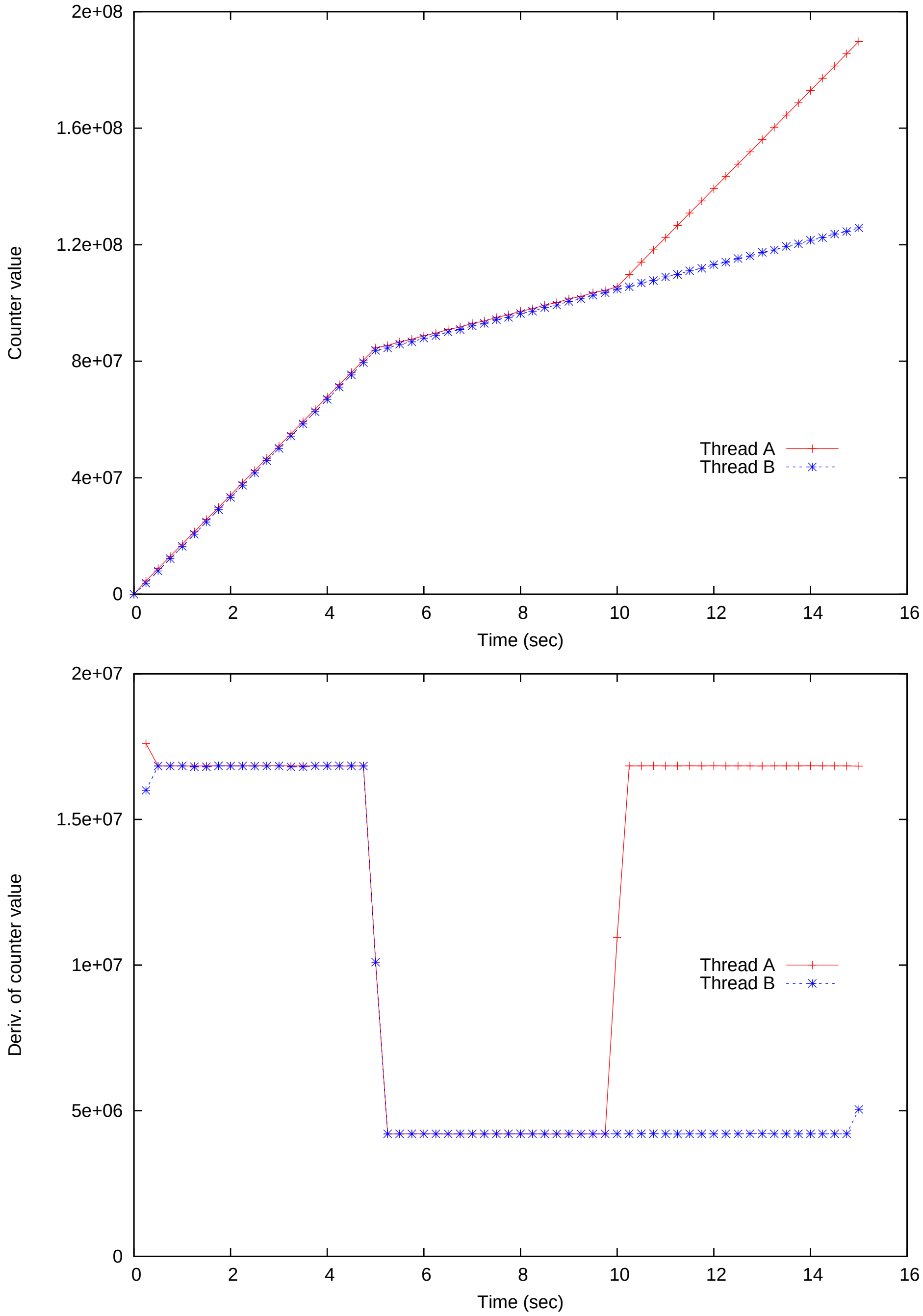


Figure 4.1: Proportional sharing demonstration. The first graph plots the per-thread counter values over time, and the second graph shows their rates of change.

4.3 Reservations for latency-sensitive tasks

In Section 2.5.1, we proposed the reserve_slots method as a way to improve the responsiveness of latency-sensitive tasks without substantially sacrificing utilization of the slot schedule. In order to validate this claim, we conducted a uniprocessor experiment involving a mix of CPU-bound and I/O-bound jobs. The workload on the K42 host had the following characteristics:

- The slot schedule was saturated with CPU-bound tasks, i.e. the only unfilled slots were those explicitly set aside by the reserve_slots parameters. These compute-bound tasks never blocked or relinquished the processor during their slots.

- A number N of single-threaded I/O-bound tasks were also running. These tasks listened for network messages from clients and replied back with message checksums. The network clients (off-host) kept the I/O servers flooded with a constant stream of requests.

- The I/O servers were not explicitly granted any slots in the schedule. However, they were assigned to a higher K42 priority band than the CPU tasks, enabling them run immediately whenever the fallback scheduler was invoked, i.e. during the slots set aside by reserve_slots().

The timeslice array was 120 slots in length, and each slot was 10 ms in duration. We measured the round-trip time of the client requests for varying values of the CPU reservation parameter, averaged over 50 messages. The reservation values were selected in increments of 5% and then translated into the corresponding integer ratios; that is, we set the slot reservation parameter to 1/20, 1/10, 3/20, . . . , 19/20, 1/1.

The results for $N = 1$ and $N = 2$ are displayed in Figure 4.2. To obtain a somewhat realistic scale for the y-axis, we calculated the plot points using timestamps that were generated by the servers running on the PowerPC simulator, rather than the actual "real-world" timestamps generated by our off-host clients. The latter were larger by a couple orders of magnitude but did not make a noticeable difference in the shapes of the resulting curves.

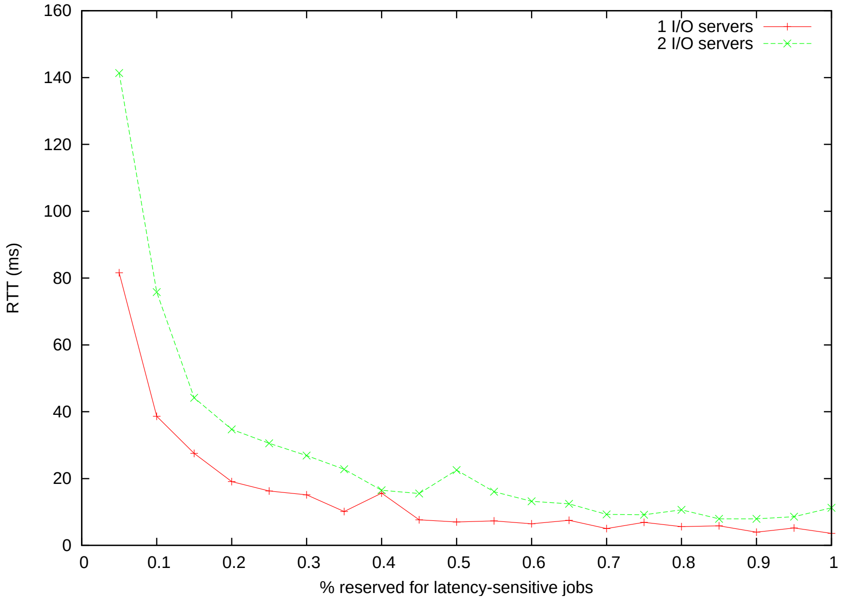


Figure 4.2: Plot showing the effects of reserve_slots() on round-trip time for I/O services.

Both plots approximately follow a reciprocal ($y = \frac{1}{x}$) shape: as the number of slots set aside for the fallback scheduler is doubled or tripled, the response time correspondingly falls by a half or two-thirds. This is behavior is good: it shows that reserve_slots() is functioning effectively as a tunable knob for managing latency. Of course, there is an inherent latency involved in message processing that cannot be eliminated by setting aside slots. Judging from the rightmost plot points (when the reservation is 100% and the CPU tasks are not scheduled at all), this irreducible latency appears to be about 5ms for one task and 10ms for dual tasks. Thus, we see the curves flatten out more quickly as the average inter-slot gap for the I/O tasks approaches this value, indicating that the presence of the CPU-bound tasks is becoming less of a factor in the round-trip delays.

4.4 Local vs. optimal scheduling algorithms

In contrast to traditional approaches to scheduling, slot scheduling decentralizes scheduling logic and distributes it among applications. These per-application scheduling algorithms operate with only local knowledge of their own scheduling requirements. It is possible that, for certain sets of tasks, such a localized approach will fail to completely schedule all tasks even though a globally feasible solution exists. To the extent that this were to be the case, slot scheduling would become a less attractive alternative for online computing environments where high schedulability or CPU utilization is demanded. However, our intuition is that incompletely schedulable task sets will be rare in practice and that the decentralized slot scheduling algorithms will in fact do quite well for the vast majority of cases.

To validate this hypothesis, we conducted an offline simulation of slot scheduling performance on synthetically-generated workloads. Our experiment involved creating random sets of periodic real-time tasks with varying periods and QoS requirements and subsequently executing first-fit slot scheduling algorithms for the tasks in some arbitrary order. (First-fit algorithms are the types of local algorithms we expect real-time scheduling libraries would use in a slot scheduling environment.) If a task was unable to be scheduled first-fit during this one-by-one procedure of placing tasks in the schedule, it was omitted completely and the next task in turn was considered. For a particular set of tasks, carrying out this procedure yields two values: the optimal utilization that would have been achieved if we had been able to schedule all the tasks in the set, and the actual utilization that the first-fit strategy was able achieve in practice.

More precisely, we define the *load* of a set of periodic real-time tasks as

$$\mathsf{load} = \frac{1}{\mathsf{\#\ of\ processors}} \sum_i \frac{\mathsf{worst\text{-}case\ execution\ time\ for\ task}\ i}{\mathsf{period\ of\ task}\ i}$$

For example, if there are two tasks assigned to a uniprocessor system, one of which needs to run 10ms out of every 40ms and the other 2ms out of every 5ms, the load of the set is $\frac{10}{40} + \frac{2}{5} = 0.65$. Obviously, a necessary (but not sufficient) condition for the complete schedulability of a set of tasks is that the load of the set does not exceed 1.0. By comparing the target loads of the task sets we generate in our experiment against the loads of the subsequently scheduled sets, we can assess how well or how poorly slot scheduling handles the schedulability challenge outlined above.[1]

[1] We also attempted to find optimal solutions for the task sets using an integer programming solver, to find out if the task sets we were presenting to the first-fit algorithms were indeed feasible

We ran experiments for simulated slot schedules with varying numbers of processors (powers of two up to 64). The per-processor slot arrays were chosen to be 120 slots in length in all cases. For each number of processors, we generated 10,000 sets of uniprocessor tasks according to a uniform distribution over the possible target loads. The tasks were subject to the following constraints:

1. Each task's period had to evenly divide the length of the schedule.
2. The amount of processor time that a task required was limited to be between 5% and 50% of a single CPU.
3. Each task had to be scheduled on a single processor and could not be migrated across different processors during execution.

The first two constraints were enforced during the generation phase of the experiment and the last constraint was placed on the scheduling algorithms themselves.

A final dimension of the experiments involved selecting a search heuristic for the multiprocessor cases, that is, how a task scheduler should decide which processor to schedule its task on when there are several processors available to it. We tested three simple strategies: *most-utilized* (the task scheduler chooses the CPU that has the highest utilization and still has room for the task), *least-utilized* (opposite of the previous strategy) and *random* (pick any available CPU).

Figures 4.3-4.5 show the average test results for the three different heuristics, measured in terms of the percentage of the target load that the first-fit algorithm was able to achieve. The graph plot points were generated by subdividing the x-axis into 50 intervals and averaging the scatterplotted y-values within those intervals (approximately 200 points per interval).

From the graphs two trends are immediately apparent. First, the first-fit algorithms are able to perfectly schedule all tasks at low loads, with performance dropping off gradually as the target load approaches 1.0. This is not surprising: as the load increases, the free space in the schedule diminishes, and inserting periodic

ones. For $P = 1, 2, 4$, the solver was able perfectly schedule all tasks presented to it. For $P > 4$, the solver could not terminate computation within a reasonable amount of time, but based on the lower-valued cases we strongly suspect that there exist feasible solutions for nearly all the task sets we generated in this fashion. This would imply that the target loads in our experiments are in fact a fairly tight upper bound on the optimal achievable loads.

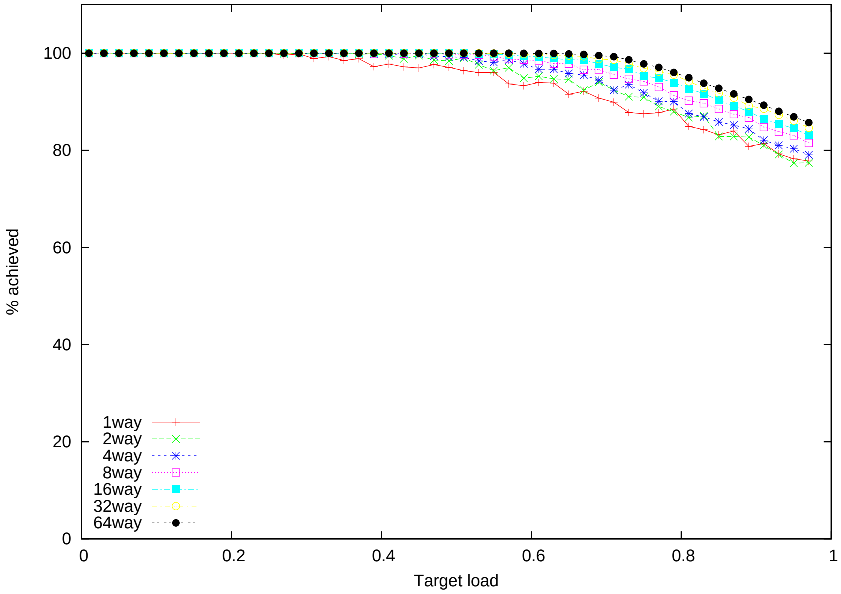


Figure 4.3: Performance of first-fit algorithms using the least-utilized CPU heuristic.

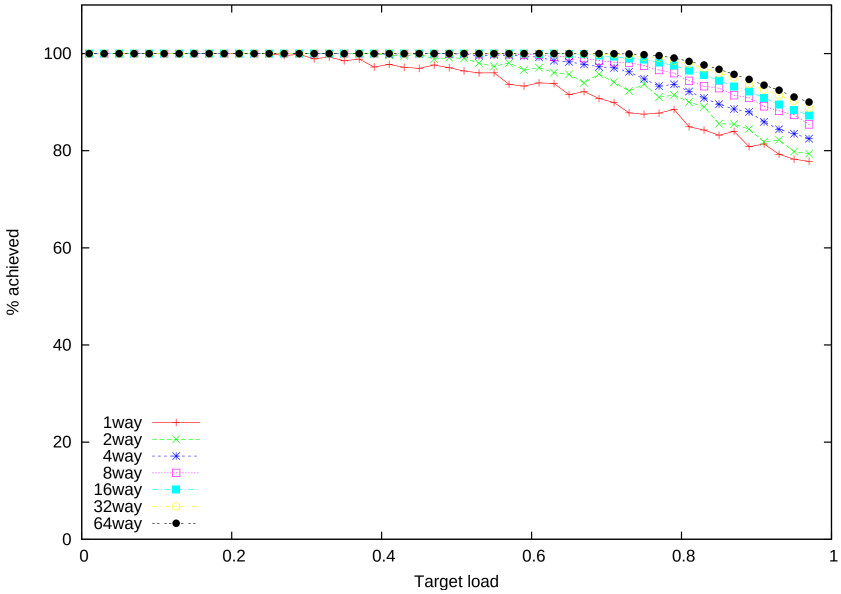


Figure 4.4: Performance of first-fit algorithms using the random CPU heuristic.

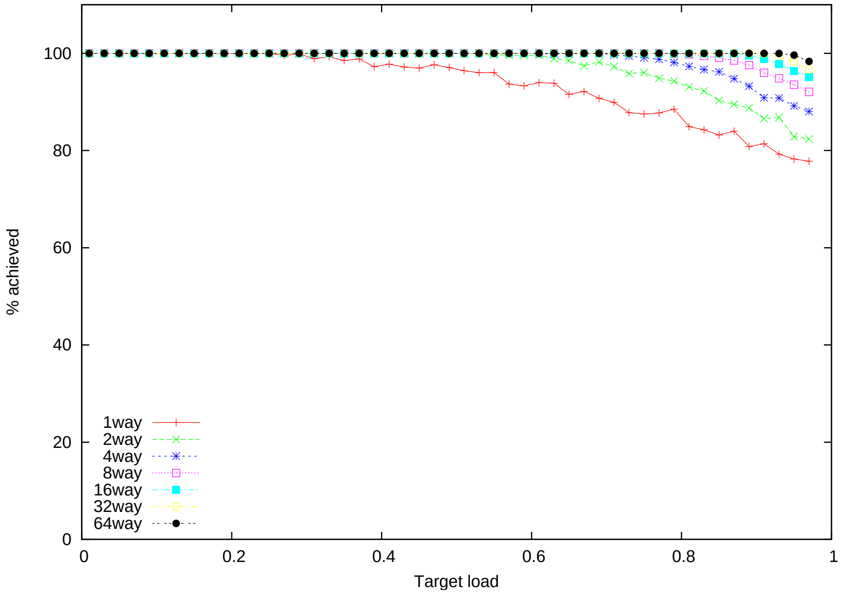


Figure 4.5: Performance of first-fit algorithms using the most-utilized CPU heuristic.

tasks becomes an increasingly tricky enterprise. Suboptimal decisions made early on by the first-fit algorithms begin to show their effects when the schedule is closer to saturation.

The second trend is perhaps less predictable: the performance of the first-fit algorithms improves as the number of processors increases, dramatically so for the most-utilized CPU heuristic. One might expect that increasing the complexity of the system, as adding processors ostensibly does, would demand the use of increasingly sophisticated algorithms in order to maintain utilization; however, it turns out that this is not the case. In fact, the simple first-fit algorithms excelled in our multiprocessor experiment environments: the most-utilized CPU heuristic achieves almost-perfect schedulability for a 64-way machine. Intuitively, the reason for this is that adding more processors enriches the variety of scheduling options available to the algorithms. For systems with large numbers of processors, even at relatively high loads, the probability that every processor will be overconstrained and unable to fit a newly-presented periodic task becomes low.

Among the three search strategies we tested, the best performing was the most-utilized CPU heuristic. This strategy derives its success from its economical approach to fitting tasks into the schedule. As the utilization of an individual CPU goes up, the probability of being able to insert a random periodic task into that processor's schedule falls. By choosing the most utilized CPU it can at each decision point, the most-utilized CPU heuristic essentially focuses on solving the most difficult scheduling problems first and reserves the "easier", less-constrained processors for later-arriving tasks that might need them.

In summary, the simple first-fit real-time scheduling algorithms perform better than might be expected and are demonstrably suitable for general-purpose computing environments. Of course, system designers can still choose to deploy more sophisticated coordinated schedulers, such as EDF, on top of a slot scheduling framework, but our experiment shows that such choices are justifiable only when the demands on system schedulability are relatively severe.

Chapter 5

Related work

Slot scheduling bears similarities to several other scheduler proposals in the literature. We have particularly exploited insights from previous work regarding coscheduling, extensible operating systems and precomputed CPU schedules. Here we present a number of comparisons with those works in order to situate our contributions.

In 1982 Ousterhout introduced the notion of gang scheduling [12] as a means of supporting the efficient execution of closely-cooperating concurrent programs on multiprocessor machines. Ousterhout proposed a matrix representation of timeslices and processors to facilitate running jobs simultaneously. Obviously, slot scheduling is strongly indebted to this way of framing the problem. In one sense, the contribution of slot scheduling lies in recognizing that the Ousterhout matrix has relevance beyond just support for coscheduling and in applying to it principles developed in more recent work on extensible operating systems.

The surge of interest in extensible OS design during the mid-90's has certainly been influential on our work. Exokernel [6], SPIN [2] and Vassal [4] are notable examples of systems that allowed modifying system schedulers at runtime to support various policies or types of workloads. Slot scheduling has more in common with the Exokernel paradigm than it does with the other approaches: the latter systems realize extensibility via the dynamic creation or importation of custom code into the kernel, whereas the focus in slot scheduling and Exokernel is on designing abstractions that permit scheduling decisions to be performed by unprivileged code executing in user space.

However, slot scheduling still differs significantly from the minimalistic approach adopted by the exokernel Aegis described in [6]. Aegis provided a simple, low-level yield primitive that could be used to construct application-level schedulers. Implementation of specific scheduling algorithms with Aegis (apparently) required relying on participating processes to "do the right thing" and cooperatively yield the remainders of their timeslices to the appropriate neighbors when they were supposed to. This approach, barring further enhancements, has obvious shortcomings for environments where processes might not be trustworthy. Slot scheduling provides the same degree of flexibility as Aegis in letting applications manage their own scheduling, but the token system enables administrators to more easily meter rights to CPU resources and not depend so heavily on the correct behavior of application-level schedulers for protecting non-faulty processes. This makes slot scheduling a more realistic alternative for general system deployment.

The real-time scheduler for the Rialto operating system, based on CPU reservations [9], bears a great deal of similarity to slot scheduling. Rialto CPU Reservations and slot scheduling both share the fundamental approach of precomputing a global, periodic CPU schedule, which lends both systems the property that CPU scheduling overhead is bounded by a constant and does not depend on the number of tasks. Both designs also rely on legacy fallback schedulers to "take up the slack" for best-effort types of jobs. However, Rialto's representation of the schedule as a directed graph is somewhat less flexible than the matrix representation employed by slot scheduling. The graph representation is suited for two fairly specific models of real-time scheduling and requires moderately sophisticated algorithms to handle the arrival or departure of new tasks. Slot scheduling, on the other hand, is not limited to any specific paradigms for real-time guarantees and can "get by" with much simpler scheduling algorithms (although it can certainly make use of more sophisticated, coordinated algorithms if desired). Strictly speaking, slot scheduling is more general than Rialto's CPU reservations: Rialto in theory could be mimicked by a privileged admission control server that brokers all real-time scheduling requests, executes the graph construction algorithm and maps the resulting graph into a series of slot reservations.[1]

However, the Rialto scheduler retains at least one advantage over slot schedul-

[1] Somewhat trivially, going in the other direction, one could map slot scheduling on to the graph framework by restricting the scheduling graphs to be linked lists; however, this would essentially *be* slot scheduling and would no longer directly support use of the Rialto algorithms for inserting or removing tasks.

ing: it allows the period of the global schedule to be any arbitrary length, in contrast to the architecture we have described for slot scheduling which requires the length of the schedule to be fixed. This limitation can be worked around to some degree in slot scheduling by doing any of the following: (1) rounding the requirements of periodic tasks down or up appropriately, so that their periods are harmonic with the global schedule length; (2) increasing the initial granularity of the schedule to accommodate a sufficiently large variety of period lengths; and (3) overallocating slots for tasks, above and beyond their QoS requirements, within the "short" intervals that result from the uneven division of the schedule by the tasks' period lengths.

Along somewhat different lines, the work on Hierarchical CPU Scheduling [7] addressed the challenge of providing scheduling services for systems running applications with an assortment of scheduling needs, akin to the challenge taken up by slot scheduling. Hierarchical CPU scheduling tackles the problem by defining a tree framework for combining different classes of schedulers, where each internal node of the tree corresponds to a scheduler and each leaf node corresponds to one of the fundamental schedulable entities in the system (either individual processes or threads). The system makes scheduling decisions recursively by starting at the root of the tree and traversing a path to a leaf node. This scheme is a natural fit for computing workloads consisting of applications that can be cleanly aggregated into scheduling classes which require CPU bandwidth to be divvied up proportionally. However, it can be difficult to reason in this framework about how to meet the deadlines of applications with more precise timing requirements (like hard real-time applications), and there does not appear to be any simple way to extend a hierarchical CPU scheduler to support coscheduling on a multiprocessor system. Slot scheduling solves both of these problems and would provide an excellent complement to a hierarchical CPU scheduler, with the latter functioning as the fallback scheduler in the slot scheduling framework.

Borrowed Virtual Time (BVT) [5] offers another alternative for supporting real-time applications in a general purpose computing environment. Based on a simple modification to the well-known "virtual time" abstraction for fair scheduling, BVT eschews strict deadline-based scheduling in favor of a "warp" mechanism that can temporarily elevate the priority of latency-sensitive jobs. In spite of not providing strong guarantees about timeliness, this mechanism in practice appears to satisfy the needs of a large number of softer real-time applications. It also has the advantage over other real-time schedulers (slot scheduling included) of not requiring any changes to existing applications and not demanding that applications be able to specify their timeliness requirements in any precise fashion. However, its suitability for dealing with some of the challenges of multiprocessor scheduling environments is less certain. Like Hierarchical CPU Scheduling, we view BVT as a solution that can coexist with slot scheduling, in the form of an enhancement to the fallback scheduler.

Chapter 6

Conclusions and future work

In this thesis we have presented slot scheduling, a framework for designing multiprocessor system schedulers. As we have shown, the chief virtues of slot scheduling are versatility (support for a broad range of scheduling algorithms) and intelligibility (ease of reasoning about interactions among disparate types of schedulers). These two properties are largely absent in existing schedulers, and their absence severely hampers the usability of emerging large-scale multiprocessor systems. In particular, slot scheduling is well-suited for the meeting the needs of both real-time and parallel applications, an advantage we believe to be unprecedented in the history of scheduler design.

In addition to its chief advantages, slot scheduling has a number of other beneficial characteristics. The runtime overhead of switching between slot-scheduled tasks is bounded by a constant and is not influenced by the number of processes in the system or the sophistication of the scheduling services being provided. Slot scheduling easily supports both online and offline scheduling, and its library-OS character allows new schedulers to be implemented and tested without having to patch the kernel or reboot the system. In spite of slot scheduling's "LibOS" character, legacy systems can still incorporate slot scheduling into their existing scheduling architectures without requiring any changes of existing programs or imposing any extra runtime scheduling overhead, because of the necessary existence of a fallback scheduler. Slot scheduling thus presents a smooth upgrade path to users of older platforms and applications.

System builders need a high-level, intuitive way to reason about and manage interactions among competing schedulers. To this end, we have introduced the abstractions of tokens, priorities and weights. These abstractions enable system architects to distinguish among various classes of applications and to define policies for fair CPU sharing. The phenomenon of slot revocations gives systems the adaptability needed for responding to changes in workload and for coping with the complexity of heterogeneous CPU demands. The result of all these enhancements is sensible behavior of the multiple schedulers that may be active in a system.

One potential challenge for slot scheduling is supporting jobs with unpredictable execution requirements, such as interactive or other event-driven types of tasks. Because of these jobs' unpredictability, their application-level schedulers cannot precompute a long-term schedule meeting their needs, unlike schedulers for other types of tasks. Our response to this dilemma is to rely on the assistance of the fallback scheduler and to introduce a tunable schedule parameter (reserve_slots()) that can guarantee a desired degree of responsiveness for event-driven applications. The benchmarks described in Section 4.3 demonstrated the efficacy of this solution.

Another potential disadvantage of slot scheduling is low CPU utilization resulting from the suboptimality of the local/greedy algorithms we expect application schedulers would typically employ. We investigated the seriousness of this drawback by generating synthetic workloads of real-time tasks and measuring how well the localized algorithms fared at scheduling these synthetic task sets. The results, presented in Section 4.4, showed that the local algorithms were remarkably adept at fitting tasks into the schedule: bad performance was so rare as to be ignorable for general scheduling purposes. Admittedly, the synthetic character of our experiment limits the conclusions that can be drawn from it, but we believe the results are sufficient to dispel any serious apprehensions about the practicality of slot scheduling.

Our work leaves open several interesting avenues for future research. We did not have the opportunity to deploy slot scheduling on actual hardware or with real-world, multiprocessor workloads; since these are the computing environments for which slot scheduling is intended, it would be good to evaluate its performance in those environments. Also, we have left as future work the exploration of policies for token distribution: how tokens should be distributed to different classes of applications, what priorities and weights should be assigned to the tokens, etc. Such issues would have to be taken up when building admission control servers.

Another aspect of slot scheduling that might merit further attention is our model for notifying applications of changes to the schedule. In the framework we have described, applications receive notifications of slot revocations—when slots are taken away from them. However, applications might also like to be informed about the opposite event: when slots that were previously claimed by other jobs become available. For example, a job that loses a portion of its slots due to the weighted sharing mechanism might want to claim those slots back if the preempting jobs later on exit the system. An obvious solution—polling the schedule at regular intervals—is less than ideal: applications must figure out how frequently to poll and must devote a portion of their CPU cycles to a cause that may not yield any benefits for them. A better solution might be to allow applications to register their desire for certain slots, with the kernel firing off asynchronous notifications when those slots become available. A slightly more sophisticated solution would be to introduce something like the wakeup predicates described for Xok in [10].

Bibliography

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *SOSP '91: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 95–109, New York, NY, USA, 1991. ACM Press.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, USA, 1995. ACM Press.

[3] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo – a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4), March 2004.

[4] George Candea and Michael B. Jones. Vassal: Loadable scheduler support for multi-policy scheduling. In *Second USENIX Windows NT Symposium*, pages 157–166, Seattle, WA, August 1998. USENIX.

[5] Kenneth J. Duda and David R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP '99: Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 261–276. ACM Press, 1999.

[6] D. R. Engler, M. F. Kaashoek, and Jr. J. O'Toole. Exokernel: an operating

system architecture for application-level resource management. In *SOSP '95: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, New York, NY, USA, 1995. ACM Press.

[7] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[8] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *SIGCOMM '96: Conference proceedings on applications, technologies, architectures, and protocols for computer communications*, pages 157–168, New York, NY, USA, 1996. ACM Press.

[9] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 198–211. ACM Press, 1997.

[10] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 52–65, New York, NY, USA, 1997. ACM Press.

[11] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[12] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, October 1982.

[13] K42 Team. K42 Overview, August 2002.

[14] K42 Team. Scheduling in K42, August 2002.

[15] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, 1995.

Vita

Brandon Richard Hall was born in Knoxville, Tennessee on December 8, 1975 to Cheryl Ann Harrison and adopted shortly after by Vicki Anne Slate and Brent Richard Hall. He graduated from Greeneville High School, Greeneville, Tennessee in 1994 and matriculated a few months later to Duke University, Durham, North Carolina. In May 1998 he received the Bachelor of Science degree in Computer Science with a Minor in Mathematics. The following year he lived and worked at Jubilee Partners, an intentional Christian community in Comer, Georgia. From 2000 to 2003 he was employed as a software engineer at USAN, Inc. in Norcross, Georgia. Prior to entering graduate school, Mr. Hall spent one hundred days riding a bicycle through the United States and México.

Permanent Address: 4010 Avenue C Apt B
Austin, TX 78751
`brandon.hall@alumni.duke.edu`

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society.