Proceedings of the Second USENIX Windows NT Symposium, Seattle, Washington, pp. 157-166, August 1998

Vassal: Loadable Scheduler Support for Multi-Policy Scheduling

George M. Candea*

M.I.T. Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

candea@mit.edu
http://pdos.lcs.mit.edu/~candea/

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 9s/1
Redmond, WA 98052

mbj@microsoft.com
http://research.microsoft.com/~mbj/

Abstract

This paper presents Vassal, a system that enables applications to dynamically load and unload CPU scheduling policies into the operating system kernel, allowing multiple policies to be in effect simultaneously. With Vassal, applications can utilize scheduling algorithms tailored to their specific needs and general-purpose operating systems can support a wide variety of special-purpose scheduling policies without implementing each of them as a permanent feature of the operating system. We implemented Vassal in the Windows NT 4.0 kernel.

Loaded schedulers coexist with the standard Windows NT scheduler, allowing most applications to continue being scheduled as before, even while specialized scheduling is employed for applications that request it. A loaded scheduler can dynamically choose to schedule threads in its class, or can delegate their scheduling to the native scheduler, exercising as much or as little control as needed. Thus, loaded schedulers can provide scheduling facilities and behaviors not otherwise available. Our initial prototype implementation of Vassal supports two concurrent scheduling policies: a single loaded scheduler and the native scheduler. The changes we made to Windows NT were minimal and they have essentially no impact on system behavior when loadable schedulers are not in use. Furthermore, loaded schedulers operate with essentially the same efficiency as the default scheduler. An added benefit of loadable schedulers is that they enable rapid prototyping of new scheduling algorithms by often removing the time-consuming reboot step from the traditional edit/compile/reboot/debug cycle.

In addition to the Vassal infrastructure, we also describe a "proof of concept" loadable real-time scheduler and performance results.

1. Introduction

A primary function of operating systems is to multiplex physical resources such as CPU, memory, and I/O devices among application programs. The CPU is one of the primary resources, and hence, it is important to schedule it effectively. This raises the question: what is the best algorithm to schedule tasks on the available CPUs?

The answer, of course, strongly depends upon the mix of tasks to be run, the demands that they place on the different resources in the system, and the relative values of the various outcomes that will result from different scheduling decisions. In the limit, for an operating system to perform optimal scheduling of its CPUs, it would need perfect knowledge of the future behavior and requirements of all its applications.

Most "general purpose" systems have used algorithms which know either nothing or next-to-nothing about the actual CPU resource needs of the tasks being scheduled. Examples include "First-Come, First Served" used in early batch systems and Round-Robin in multi-tasking systems. Later algorithms such as Priority Queues ([Corbató & Daggett 62], [Lampson 68]), Fair Share Scheduling ([Kay & Lauder 88]), and typical dynamic priority boost/decay algorithms still had the property that they were essentially ignorant of the actual CPU needs of their applications.

Imperfect, but nonetheless adequate, future knowledge is possible for some fixed task sets with well-characterized computation patterns. Whole families of scheduling disciplines have arisen in the computer systems research community to provide appropriate scheduling for some such classes. Examples are: Earliest Deadline First ([Liu & Layland 73]), Weighted Fair Queuing ([Clark et al. 92]), Pinwheel Scheduling ([Hsue & Lin 96]), and Proportional Share CPU allocation mechanisms ([Waldspurger 95], [Goyal et al. 96]), plus techniques such as Rate Monotonic Analysis ([Liu & Layland 73]) and Priority Inheritance ([Sha et al. 90]). Similarly, Gang Scheduling ([Ousterhout 82]) and Implicit Coscheduling ([Dusseau et al. 96]) were developed for parallel workloads where the forward progress of members of a task set is closely dependent upon the progress of other tasks in the set.

But today's general purpose operating systems do not provide such specialized scheduling algorithms. Some of the more popular operating systems provide a primitive differentiation between the different scheduling classes by mapping them onto different priorities (e.g., System V Release 4 [Goodheart & Cox 94], Windows NT [Solomon 98]) and then scheduling higher priority tasks more often or for longer periods of time. However, it is extremely

* The research described in this paper was done while George Candea was a summer intern at Microsoft Research.

hard to properly map requirements such as predictability, throughput, fairness, turnaround time, waiting time, or response time onto a fixed set of priorities. Moreover, different applications may use different mappings, defeating their purpose. For instance, when co-existing applications do not share coordinated priority mappings or goals, it is common for applications wanting "real-time performance" to raise their priority to the highest one available under the assumption that they are "the most important" task in the system — a phenomenon known as "priority inflation". Priorities are, at best, a rather primitive way of describing the relative performance requirements of the threads and processes that belong to the different classes.

Other systems ([Northcutt 88], [Jones et al. 97], [Nieh & Lam 97], etc.) have tried to strike a compromise, providing some application timing and resource advice to the system, giving it imperfect, but useful information upon which to base scheduling decisions.

Such large numbers of different scheduling algorithms are an indication that scheduling is, and will likely remain, an active area of research. No one algorithm will work best in all cases (despite many valiant attempts by system builders to demonstrate otherwise). In fact, [Kleinrock 74] shows that any scheduling algorithm that favors a certain class of tasks will necessarily hurt another class. A single scheduling policy will always represent a compromise and the service offered by the system will unavoidably reflect this compromise.

Our Proposed Solution

As discussed above, we believe that any particular choice of scheduling algorithm will fail to address the needs of some classes of applications, particularly when independent applications with different scheduling requirements are concurrently executed. Rather than attempting to devise yet one more "good compromise" we explored a different approach.

We decided to find out whether we could dynamically extend systems with scheduling algorithms. While nearly all modern established operating systems can be partially extended via loadable modules (e.g., Linux, Solaris, Windows NT) and extensible systems are a very active area of research ([Bershad et al. 95], [Engler et al. 95], [Seltzer & Small 97]), none of these systems allowed arbitrary scheduling policies to be implemented as extensions — motivating our work on Vassal.

The results were quite positive: it was straightforward to modify a modern commercial operating system, in this case Windows NT 4.0, in order to allow independently developed and compiled schedulers to be dynamically loaded into (and unloaded from) the operating system at run-time.

The loaded schedulers can take control of as many or as few of the system's scheduling decisions as desired. For instance, in our implementation, the existing Windows NT scheduler was retained, so a loaded scheduler can always fall back upon the default system scheduler if it chooses not to make a particular scheduling decision. And in the case when no loadable scheduler is present, the system works exactly as it would have were the loadable scheduler support not there.

The modifications resulted in no measurable performance penalty when loadable schedulers are not in use. Furthermore, loadable schedulers can operate with nearly the same efficiency as the native system scheduler. Finally, having a loadable scheduler infrastructure makes it easy to experiment with different schedulers, providing special-purpose scheduling on a general-purpose system.

The present Vassal implementation is clearly a prototype, with some limitations. For instance, at present we only support the simultaneous coexistence of two schedulers: the Windows NT scheduler and a single loaded scheduler. Nonetheless, we believe that the techniques and results obtained with the prototype will remain valid once these limitations are removed. For more on this topic, see Section 8.

In the following sections we provide some background on the system we started with, describe the particular system we built in more detail, and then show what transformations we made to the vanilla system. We present a "proof-of-concept" real-time scheduler that we wrote, followed by performance measurements. We then discuss the experiences we had while building and experimenting with the loadable multi-policy scheduler support and conclude.

2. Background

This section provides background information on some of the features of Windows NT 4.0 relevant to our loadable scheduler work. We describe the native scheduler and its implementation, and also present briefly the NT driver model.

Windows NT Scheduling Model

Windows NT uses threads as its basic schedulable unit. Threads can exist in a number of states, the most important ones being: *Running* (executing on a processor), *Standby* (has been selected for execution on a processor and is waiting for a context switch to occur), *Ready* (ready to execute but not running or standing by), *Waiting* (either waiting on a synchronization object, such as a semaphore, waiting for I/O to complete, or has been suspended), and *Terminated* (the thread is not executing anymore and may be freed or recycled).

The thread dispatcher is responsible for scheduling the threads and it does this based on two thread characteristics:

- priority (higher priority threads are scheduled before lower-priority ones);
- processor affinity (threads may have preferences for a certain processor in multi-processor systems and this is accounted for when scheduling it).

Windows NT provides a set of 32 priorities, which are partitioned into three groups:

1. The *Real-Time* scheduling class, which includes the highest priorities in the system (16-31). Threads belonging to this class can gain exclusive use of all scheduled time on a processor if there is no runnable thread with a higher priority in the system.
2. The *Variable* priority scheduling class, which includes priorities 1-15. Threads belonging to this class are subject to priority drops (e.g., when the thread's time quantum runs out) or priority boosts (e.g., when awaited I/O completes). As can be seen, the priority of a CPU-bound thread in this class decays over time, unlike threads in the *Real-Time* class.
3. Priority 0 is the lowest priority and is reserved for the so-called *idle* thread. This thread runs whenever there is no ready thread available in the system.

It is important to note that under Windows NT, not all CPU time is controlled by the scheduler. Of course, time spent in interrupt handling is unscheduled, although the system is designed to minimize hardware interrupt latencies by doing as little work as possible at interrupt level and quickly returning from the interrupt. The mechanism that ensures this is Deferred Procedure Calls (DPCs). DPCs are routines executed within the Windows NT kernel in the context of no particular thread in response to queued requests for their execution. For example, DPCs check the timer queues for expired timers and process the completion of I/O requests. The way hardware interrupt latency is reduced is by having interrupt handlers queue DPCs to finish the processing associated with the interrupt and then return. Due to their importance, DPCs are executed whenever a scheduling event is triggered, prior to starting the scheduled thread, and they do not count against any thread's time slice.

Windows NT Scheduler Implementation

A scheduling request can be generated by a number of events. Some of these are:

- The time quantum of a running thread expires.
- Thread state changes, such as when a thread enters the *Ready* state or when the currently running thread enters the *Waiting* or *Terminated* state.
- When the priority or affinity of a thread in the system is changed from outside the scheduler (e.g., by the *SetThreadPriority()* call).

Whenever the hardware clock generates an interrupt, the Hardware Abstraction Layer (HAL), which exports a virtual machine to the NT kernel, processes the interrupt and performs platform-specific functions. After that, control is given to the kernel. At this point the kernel updates a number of counters, such as the system time, and inspects the queue that contains timers. For every expired timer it queues an associated DPC. After that it decrements the running thread's time quantum and checks whether it has run out. If yes, it issues a *DISPATCH* software interrupt on the corresponding processor. All events that trigger scheduling raise *DISPATCH* software interrupts.

The *DISPATCH* software interrupt then invokes a kernel handler which first runs all the queued DPCs. After this, the thread dispatcher is ready to make a scheduling decision.

The set of data structures used by the NT dispatcher are collectively known as the dispatcher database. This set contains information about which threads are running on which processors, which threads are ready to run, etc. The most important data structure is the set of thread queues that keep track of threads in *Ready* state; there is one such queue for each priority (except 0). Whenever scheduling is triggered, the scheduler/dispatcher walks the *Ready* thread queues in decreasing order of priority. It then schedules the first thread it finds, provided the thread's processor affinity allows it to be scheduled on the free processor. The thread is prepared for execution (if not currently running), a context switch is performed, and then the DISPATCH service routine returns from the interrupt.

The NT kernel provides a system call, *NtSetTimerResolution()*, which allows the frequency of clock interrupts to be adjusted. Specifically, when an application needs high resolution timers, it may choose to lower the time between clock interrupts from the default (typically 10ms) to the minimum supported (typically 1ms).

Of particular importance to scheduling is the fact that the HAL does not export a programmable timer to the kernel, which denies the kernel the ability to reschedule at a precise point in time. For instance the programmable timer available on x86 PCs is used by the HAL as a countdown timer that is repeatedly set to the current interval between interrupts (so it is essentially used as a periodic timer). Most other non-real-time operating systems running on the x86 do the same.

Windows NT Driver Model

Drivers in Windows NT do much more than traditional device drivers, which just enable the kernel to interface to hardware devices. NT drivers are more of a general mechanism by which NT can be extended. For example, under NT, filesystems, network protocol implementations, and hardware device management code are all separately compiled, dynamically loadable device drivers. Drivers reside in kernel space, can be layered on top of each other, and communicate among themselves using I/O Request Packets (IRPs) in a manner reminiscent of UNIX System V Streams modules [Ritchie 84]. Applications typically send and receive data to and from the drivers via the same path.

3. Loadable Scheduler Design

This section describes the design of the infrastructure that allows scheduling policies to be loaded and unloaded at run-time. It is intended to provide a guide to implementing such a system, while omitting OS-specific details, which are discussed in the following section.

The basic idea is to modify the thread dispatcher inside the kernel so that it handles multiple scheduling policies. It is the *decision making* component of a scheduler that contains all the policy, so we chose to externalize the decision-making by encapsulating it in loadable drivers, while leaving all the dispatching mechanism in the kernel. We wish to replace the statement "the dispatcher decides which thread to run" with "the dispatcher queries the schedulers for which thread to run." The maintenance of the thread queues, being a chore specific to the decision making process, is done by the external schedulers themselves. The in-kernel dispatcher simply expects a reference to the appropriate thread data structure from the scheduler it queries. Figure 1 shows the conceptual architecture of Vassal with an example in which there are four tasks running on the system (*T1, T2, T3, T4*) and there are currently two schedulers available (*A* and *B*).

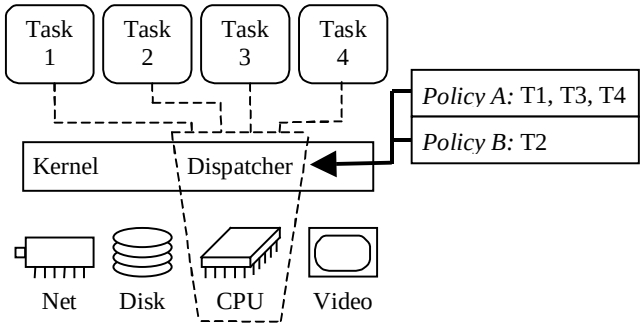

Figure 1: The Loadable Scheduler Infrastructure

The Vassal dispatcher manages the schedulers and dispatches/multiplexes messages between the kernel and the schedulers. It is responsible for:

- Receiving scheduling requests from the kernel and deciding which scheduler to query.
- Relaying the scheduler's response to the application.
- Deciding whether a scheduler that is attempting to load would conflict with already existing schedulers. The scheduler is loaded only if there are no conflicts.
- Enabling communication between threads and schedulers.

Scheduling

When a *DISPATCH* software interrupt is generated, an interrupt service routine is invoked and eventually hands control over to the dispatcher. The dispatcher then decides which scheduler to query. In our current model, we simply use a hierarchy of schedulers (with the external scheduler being at the top), so that if a higher level scheduler does not have a ready thread, then the next one (in descending order) is queried. Section 8 describes other possible ways of managing relationships between schedulers. Once a scheduler responds with a runnable thread, the dispatcher can perform a context switch (if necessary), schedule the thread, and return.

It may seem that, by using a hierarchy of schedulers, we are essentially making the decisions based on a set of priorities. However, there is a significant difference between scheduler hierarchy and thread hierarchy: the scheduler priorities have nothing to do with what the threads think is more important (which would motivate their choices for priorities) rather it has to do with implicit relationships between the schedulers that result from their CPU resource requirements.

Thread Creation

Newly created threads initially execute using the system's default scheduler. It can then make explicit requests to be scheduled by other schedulers. Other approaches could have equally well been taken. For instance, a thread could inherit its parent's scheduling class, or an explicit scheduler parameter could have been provided to an extended thread creation call.

Here is an example scenario. A task (e.g., *T2*) is created and associated with the default scheduler (e.g., *A*). After running for a while, *T2* makes a system call to inform the system that it now wants to switch to another scheduler. The kernel relays this information to the desired scheduler, which in turn removes the task from the previous scheduler's jurisdiction. From this point on, the new scheduler has sole ownership over *T2*'s schedule (until the task decides to switch again).

Communication between Threads and Schedulers

For optimal scheduling decisions, every scheduler needs semantic information about the intentions and requirements of the threads under its jurisdiction. Once provided with this information, schedulers can make the appropriate scheduling decisions. For this reason, the operating system interface needs a system call that allows threads to communicate with a scheduler of their choice. The dispatcher receives this stream of messages from the kernel and demultiplexes it. By this means, a thread could inform its scheduler that it wants to communicate with a thread on another processor and, thus, the scheduler should attempt to schedule that thread concurrently with the requesting thread.

One question that arises naturally is whether the use of the external schedulers would negatively impact performance, given that on every timer interrupt there would be a query going out to these schedulers. However, the critical path followed by these queries turns out to be very short, because the only added time is that of performing a small number of memory reads from non-pageable memory. Remember that schedulers (being drivers) reside in the kernel address space. As our results suggest, this added overhead is negligible and is clearly offset by the gains in scheduling performance. Also, in a multi-threaded kernel such as NT's, it is possible for the schedulers to avoid making decisions on the critical path by having their decisions ready before they are queried.

Another issue we considered was whether a thread that was not selected for execution by its current scheduler could be selected by another scheduler. For instance, one

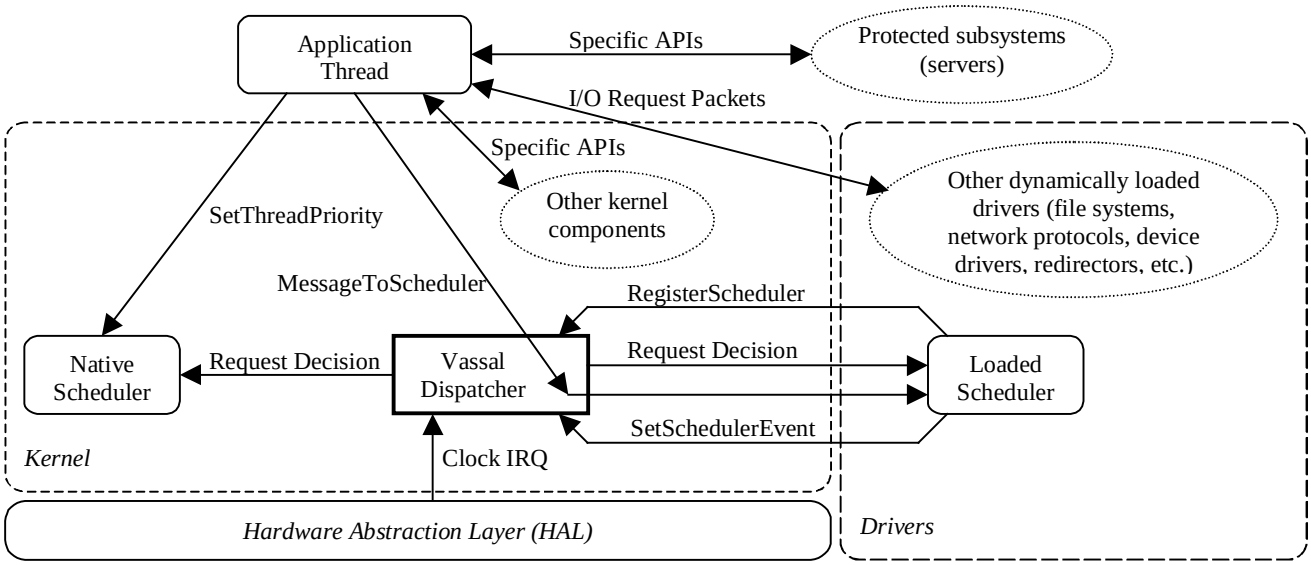

Figure 2: Integration of Vassal into Windows NT 4.0

could view this as happening in [Jones et al. 97] when a thread's CPU reservations and/or time constraints do not cause it to be selected, but it is nonetheless selected by the default round-robin policy. As this is more general, we opted to allow this (while also making it possible for a scheduler to prevent it). One example of our use of multi-policy scheduling is that a thread not selected by the sample scheduler described in Section 5 can typically still be selected by the default scheduler.

4. Vassal Implementation

This section describes the Vassal implementation, including the modifications and additions we made to the Windows NT kernel in order to support multiple schedulers. An overview of the structure of the Vassal implementation within Windows NT is shown in Figure 2.

First of all, we needed to add a way for the scheduler/driver to notify the system that it is being loaded. For this we added a kernel function for scheduler drivers:

```
RegisterScheduler(scheduler,
   decision_maker, message_dispatcher)
```

The *decision_maker* parameter to *RegisterScheduler()* is the address of a function to be called when a scheduling decision needs to be made. This function must either return a runnable thread to be scheduled or NULL, which indicates that the loaded scheduler has no opinion as to which thread should be scheduled. In this case, Vassal calls the next scheduler in the hierarchy, allowing it to make the scheduling decision. (In the present prototype this means the decision is always delegated to the built-in scheduler.)

The *message_dispatcher* parameter to *RegisterScheduler()* is the address of a function to be called to handle messages from threads to the scheduler. This routine handles requests sent via the *MessageToScheduler()* system call, which is described below.

In response to the *RegisterScheduler()* call, Vassal saves these parameters of the loaded scheduler and activates the new scheduler. We also provide a matching *UnregisterScheduler()* function, which allows a scheduler to be unloaded cleanly (using the customary procedure for unloading drivers).

For applications to be able to communicate with the schedulers, we added a new system call:

```
MessageToScheduler(scheduler, buffer,
   buflen)
```

by which an application can send a message (in *buffer*) to a loaded scheduler. Upon receiving this call, the scheduler's *message_dispatcher* routine will be invoked, passing the buffer contents as a parameter. The scheduler then performs the action specified by the buffer contents. When the routine completes, its return value is returned to the application. Note that *MessageToScheduler()* need not return immediately; it is free to block or take any action that a regular driver could take.

Of course, given that loaded schedulers are full-fledged Windows NT device drivers, one might ask why we added the *MessageToScheduler()* system call at all — why not just use standard device *Read()* and *Write()* operations to communicate with the scheduler? As described in Section 2, these user-space operations generate I/O Request Packets (IRPs) that would travel through the I/O subsystem and eventually reach the scheduler/driver. In fact, we initially did use IRPs to communicate with loaded schedulers.

We added the *MessageToScheduler()* system call because we found that, in practice, the amount of time it would take requests to reach the scheduler via the standard I/O path was too unpredictable for the time-critical services that the external scheduler is intended to provide, especially for the kinds of real-time schedulers we were

attempting to construct. The system call gave Vassal both lower and more predictable latency.

One capability that is essential for many kinds of schedulers (in particular, real-time) is the ability for the scheduler to cause an action at a designated time. As a basis for providing this capability, we added an internal kernel function available to schedulers:

```
SetSchedulerEvent(scheduler,
    performance_counter_reading)
```

This call instructs the kernel to call the scheduler's *decision_maker* function whenever the system performance counter's value (a monotonically increasing system-provided low-latency 64-bit real-time timer provided by the HAL) is greater than or equal to *performance_counter_reading*. This facility allows schedulers to set deadlines. There is a matching *CancelSchedulerEvent()* function that cancels the call.

To support the functionality described above, we made modifications to the routine that services the *DISPATCH* software interrupts. Its function is to process the Deferred Procedure Call (DPC) list, query the scheduler if necessary and then perform a context switch if a new thread has been selected for execution on the processor. We needed to add a hook that would substitute calling the loaded scheduler's decision function, when one is available, in place of querying the built-in scheduler. Additionally, for the servicing of scheduler events, we needed to further modify this routine so that it would check on every clock interrupt whether the performance counter reading had reached the desired value and, if so, trigger a scheduling decision.

Given that drivers do not have full access to kernel data structures, we also needed to add a number of simple methods that allow schedulers to manipulate and gain access to parts of those data structures. This new functionality includes finding which CPU is currently being scheduled, what the status of a thread is, removing/adding threads from/to the pool of natively scheduled threads, and preempting a thread.

5. A Sample Scheduler

We wrote a simple real-time scheduler as a proof of concept. It was rather straightforward (116 lines of C code). This scheduler allows threads to request that they be scheduled at a particular time, which exercises the key operation needed to implement more interesting time-based scheduling primitives, such as time constraints [Jones et al. 97]. Using this scheduler, we can easily write timers that have a much higher accuracy and resolution than the multimedia timers offered by Windows NT (multimedia developers choose to use buffering and a number of other tricks to circumvent the limitations of NT timers; with our scheduler, this would be unnecessary). Section 6 shows the measured latencies for these timers.

In order for the sample real-time scheduler to achieve its goal, we made two important decisions:

- We added the concept of a *settable event* and added a call to the kernel interface (as described in the previous section) that would allow a driver to set such an event. In essence, we enable a driver to request that it be given control of the CPU at a specific point in time based on the value of the performance counter. This counter is typically both a precise and accurate way of measuring time. On x86 CPUs using standard HALs, for instance, its resolution is 0.838µs. It might seem that we essentially modified the kernel to provide support for a specific scheduler. However, we made this modification because we saw it as a useful feature for many types of schedulers (for instance if they need to perform certain actions at regular intervals or they need to synchronize with other processes based on time).
- If the requested time constraint cannot be met because it is very short (e.g., a time constraint of 200µs), we choose to spin in a loop until the time comes to schedule the thread. We cannot presently count on a higher accuracy than 1ms from the HAL (see Section 8), so we used this admitted hack in the sample scheduler (not the kernel code) to achieve higher resolution for one thread.

Note that we have not implemented on-demand loading of schedulers but it would be very simple to do. Currently, in order to load the real-time scheduler and have it coexist with the native scheduler the system administrator uses the Control Panel and select the appropriate driver. It can be unloaded in the same way.

The following code snippet shows a simple thread using the real-time scheduler.

```
/* Tell system to use the real-time scheduler */
status = MessageToScheduler (rt_sched, {JOIN});
if (status != SUCCESS) {
    error ("Could not join R/T scheduling class.");
}
/* Calculate how long our loop iterations take */
estimate = Calibrate();
/* Start the loop 1 ms from now */
status = MessageToScheduler (rt_sched, {SET, 1000});
if (status != SUCCESS) {
    error ("Could not set deadline.");
}
/* We want one iteration every 300 µs */
while (1) {
    status = MessageToScheduler (rt_sched, {SET, 300 –
        estimate});
    …
}
```

The function *Calibrate()* computes an estimate of how long it will take to perform each loop iteration. Notice the use of a single system call to communicate with the scheduler. Figure 3 details the actions that are triggered by the various steps in the program.

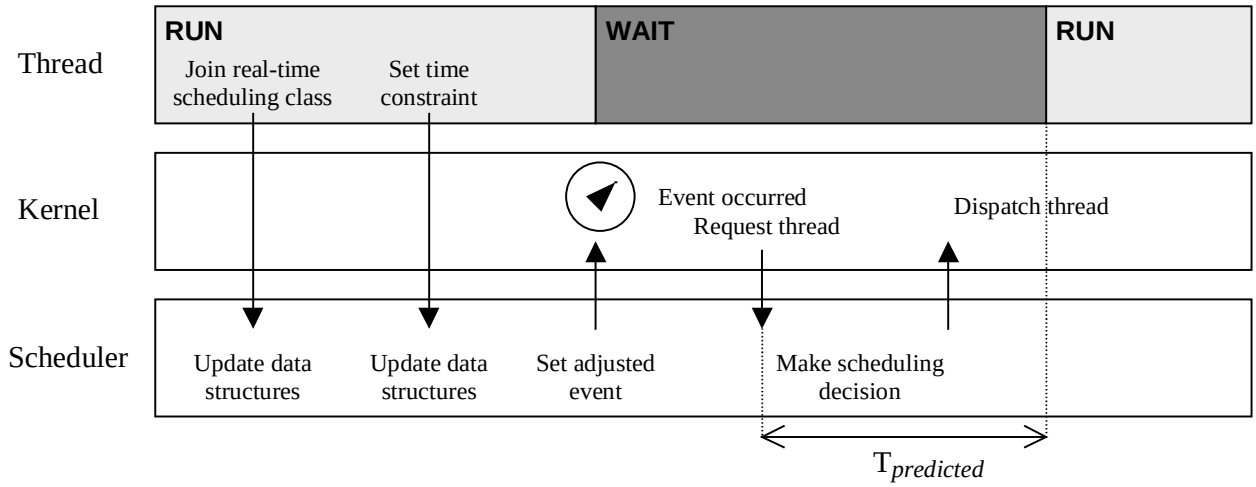

Figure 3: The actions taken for the execution of the first part of the sample code

The first two system calls translate into messages delivered directly to the loaded scheduler; if the thread's request can be satisfied, it returns a status of SUCCESS. The scheduler then updates its data structures to reflect the thread's requirements and sets the event mentioned above to the appropriate performance counter reading (the constraint interval of 1ms is large enough to do this). Then, when the event occurs, the scheduler is notified and asked for a runnable thread. As a result, the scheduler sees that the deadline has arrived for the requesting thread and instructs the kernel to schedule it. Note that the setting of the event takes into account the fixed time $T_{predicted}$, which is the platform-dependent time it takes for a message to make it through the critical path (as described in Section 3).

An important property of the scheduler is that if it is unable to satisfy the thread's request, it informs it *right away* (via the return code of *MessageToScheduler*). This is in contrast to what happens on most general purpose operating systems, where the thread expecting to meet a certain deadline finds out it cannot make it barely after it has already missed the deadline. Using the information given by our real-time scheduler, the application could adjust and decide to take some action that can compensate for the missed deadline.

The current method of keeping a thread spinning if the time constraint is very small is not a very clean solution. However, other means of obtaining the desired accuracy would require more substantial changes to the underlying kernel and, more importantly, to the HALs. Such solutions would be more difficult to adopt.

6. Results

Code Size Results

The Vassal changes made to the Windows NT kernel to support multi-policy scheduling added 188 lines of C code, added 61 assembly instructions, and replaced 6 assembly instructions.

The proof-of-concept external scheduler described earlier required only 116 lines of C code and no assembly language. We believe these are extremely low code size numbers for the increased functionality that we achieved.

Performance Results

One of the primary criteria against which any loadable scheduler support would be judged when added to a production operating system would be whether it makes things any worse for applications not using it. We are happy to report that, when a loadable scheduler is not in use, our changes have no performance impact on system performance.

One value that the use of loaded schedulers might be expected to change is the context switch time. To measure the performance impact of our changes, we ran a program that recorded actual context switch times as observed from user space by 10 threads, over a period of 10 seconds. Times were collected using the Pentium cycle counter on a 133MHz Pentium PC. Table 1 shows the results.

System Version	Median	Avg.	Std. Dev.
Vanilla NT 4.0 (released)	17.03	18.71	4.17
Vanilla NT 4.0 (rebuilt)	19.95	19.88	1.64
Vassal (no loaded scheduler)	19.71	19.71	1.56
Vassal (sample scheduler loaded)	21.32	21.17	1.28

Table 1: Measured context switch times on a Pentium-133 running the original and the modified systems (in µs).

We first explain the difference in the first two sets of data. The "Vanilla NT 4.0 (released)" figures are from the product version of NT 4.0 Workstation. The "Vanilla NT 4.0 (rebuilt)" figures are for a kernel built from the identical NT 4.0 sources with no modifications. However, the rebuilt version does not contain all of the binary optimizations contained in the product version. This explains why it is roughly 6-7 percent slower than the product version. All Vassal versions are built with the

same optimizations as the rebuilt version. Thus, the rebuilt version provides the correct basis for comparison.

The good news is that the Vassal version of NT 4.0 (with loadable scheduler support) with no scheduler loaded has essentially the same context switch time as the rebuilt version. More precisely, their times differ by less than the variations seen while measuring the times, and thus exhibit no statistically significant difference.

Finally, while loading the sample scheduler does increase the observed context switch time by about 8 percent, we believe that this is within acceptable bounds, given the increased functionality and the fact that this cost is only incurred when the added functionality is actually used. Furthermore, we believe it is likely that some of this 8 percent overhead can be eliminated, given that the current prototype is essentially untuned.

Sample Scheduler Results

The proof-of-concept sample real-time scheduler implements one primitive thread scheduling operation not otherwise found in standard Windows NT: a precisely timed thread wakeup. In a loop, this can be used to perform periodic processing, such as doing a short operation once every millisecond.

Standard Windows NT contains periodic multimedia timers that are designed for this kind of periodic processing. This section compares the effectiveness of doing periodic wakeups once per millisecond with the loaded sample scheduler and NT's multimedia timers. Table 2 shows the results.

Method	Min.	Max.	Avg.	Std. Dev.
NT Multimedia Timers	75	1566	996	82
Sample Scheduler Events	996	1485	1002	21

Table 2: Periodic wakeup times on a Pentium-133 using multimedia timers on the original system and the sample scheduler's events on the modified system (in µs). The desired value is 1ms.

Two differences are evident in the results. First, while using multimedia timers some wakeups occurred extremely early, as much as 925µs too soon. With the sample scheduler wakeups occurred at most 4µs early.

Second, predictability of the wakeups with the sample scheduler is significantly better than with the multimedia timers. The standard deviation of the sample scheduler data is only a quarter of that for the multimedia timers.

With both methods, some samples occurred up to ~0.5ms late. Initial studies indicate that these samples are due to interrupts, DPCs, and other non-scheduled system activities, although this bears further investigation.

While extremely simple, we believe that this example begins to show the potential of extending the scheduling policies available to applications through the use of loadable schedulers.

7. Related Work

Windows NT was certainly not the first operating system to use priority classes. For instance, UNIX System V Release 4 has a similar notion and supports three basic types (time-sharing, system, and real-time). The design allows for the incorporation of new priority classes, which can be described by a special class structure and compiled into the kernel ([Goodheart & Cox 94]). However, these priority classes always map their scheduling requirements onto global priority values and the scheduler runs the process with the highest global priority. Unlike Windows NT, these priorities are controlled to some extent by the in-kernel class-specific functions, which could have some global knowledge of the tasks running in the system. In spite of this, such a system is limited, primarily because:

- New scheduling policies need to be hard-coded into the kernel; this implies the need for source code, which may not always be available. In addition, a programmer writing a new scheduling policy may not want to have to dive into kernel internals in order to implement the policy. If the programmer writing extensions for a system needs to know as much about the system as the person who wrote it, then such extensions will likely never be written.
- The first disadvantage implies the second: scheduling policies cannot be dynamically added or removed at runtime, which makes the system less flexible and makes the debug cycle longer.

Solaris does allow scheduling classes to be dynamically loaded into the kernel, although these classes are still subject to the restriction that they map their scheduling decisions onto a global thread priority space.

A number of recent efforts are aimed at making operating systems extensible [Bershad et al. 95], [Kaashoek et al. 97], [Seltzer & Small 97]. However these do not have the same goals as Vassal and do not provide the same facilities.

The one that offers scheduling features closest to Vassal is SPIN [Bershad et al. 95]. It offers applications the ability to provide their own thread package and scheduler, which can then execute in kernel space. This way applications can define their own thread semantics. A global scheduler implements the primary scheduling policy, which is a priority scheduler, with round-robin execution within each priority. The global scheduler is not extensible. Application-defined schedulers are layered on top of the global scheduler. However, this global scheduler may reclaim the CPU from any given strand, therefore no application-defined scheduler has any guarantee of when it will receive time to schedule. Note that SPIN is addressing a different problem domain than we are. It loads Modula-3 extensions into the kernel, which are limited by the type-safe characteristics of the language. The purpose of this is to achieve protection. In our model, the "extensions" (i.e., the drivers) are trusted and therefore we do not need to protect against them.

Another body of work related to ours pertains to support for hierarchical scheduling. In such systems, time allocated to one scheduler is sub-allocated to other dependent schedulers. [Ford & Susarla 96] describes one such system.

Finally, [Deng & Liu 97] propose a new hierarchical scheduler for Windows NT that provides timing guarantees for multiple independent hard real-time applications, while continuing to run normal applications.

8. Limitations and Future Work

The present Vassal implementation is definitely a prototype, and is subject to certain restrictions and limitations. This section describes some of the present implementation limitations and possible future work.

Some of the most severe limitations that Vassal loadable schedulers are presently subjected to are actually limitations of the underlying Windows NT system upon which they were built. In particular, loadable schedulers (as well as the built-in scheduler) cannot exercise complete control over what code is run when.

As described in Section 2, all system timer services are actually provided to the kernel through the Hardware Abstraction Layer (HAL), which does not provide an interface to cause an interrupt at a precise time. Timer interrupts are only generated at the clock tick frequency, which while settable, is not settable to resolutions finer than 1ms for the standard x86 ISA PC HAL. Thus, sub-millisecond preemptive scheduling is effectively not implementable under NT in a clean way. The PC hardware does support this capability; removing this restriction would involve adding precise non-periodic interrupt capabilities to the HAL interface and HAL implementations.

Other sources of unscheduled time include DPCs and interrupts. The system could be modified to schedule DPCs, although this would result in both additional overhead and greater code complexity. We suspect that very little can be done to improve interrupt latencies beyond what the production system has already achieved.

Another limitation present in the current Vassal implementation is that only one external scheduler may be loaded at once. This is partly due to the slightly higher implementation complexity involved in managing the state of multiple loaded schedulers but actually mostly due to more fundamental policy issues that would need to be resolved to support arbitrary numbers of coexisting schedulers in a general way.

The key policy issue is this: schedulers might have conflicting goals, or equivalently, might be running applications with conflicting goals. For instance, if two schedulers want to run different code at precisely the same time on the same processor, the schedulers are in conflict. The question is what to do when schedulers' goals collide.

Our present implementation solves this by simple fiat — the loaded scheduler always takes precedence over the built-in scheduler. Provided the built-in scheduler is occasionally given an opportunity to run, this does not violate its invariants, since it does not depend upon being run at any particular time, unlike real-time schedulers.

Indeed, a similar strict decision hierarchy could be used among more than two coexisting schedulers, although this admits the possibility of loading two schedulers (or applications using them) with mutually incompatible requirements.

One way to achieve conflict resolution would be by allowing loadable schedulers to describe the ways they will use the CPU. For example, one scheduler may need "hard time" (i.e. it needs the CPU at definite moments in time) while another one may only require a "share" of the CPU (no matter when it occurs). A simple way to describe such requirements would be by using predefined usage patterns (i.e. "hard time," "proportional share," etc.). A more sophisticated solution would involve the use of a formal language by which schedulers could express their requested usage pattern.

Based on these usage descriptions, the dispatcher could determine whether a scheduler that is attempting to load would conflict with an already existing scheduler. For instance, multiple schedulers with "hard time" requirements will typically not coexist. This way, conflicts could be usefully detected and avoided or prevented, given sufficiently accurate CPU requirements specifications for each scheduler and/or {scheduler, application} pair.

9. Conclusion

This paper has described an infrastructure that allows multiple scheduling policies to coexist simultaneously in an operating system. It has shown that scheduling policy modules may be developed separately from the base operating system, and that these policy modules can be dynamically loaded and unloaded from a running system as needed. It has demonstrated that both this infrastructure and loadable schedulers can be simple to implement, even for commercial operating systems of the complexity of Windows NT.

The addition of these capabilities resulted in no measurable performance penalty when loadable schedulers are not in use. Furthermore, loadable schedulers can operate with nearly the same efficiency as the native system scheduler.

We found that having a loadable scheduler infrastructure makes it easy to experiment with and utilize different schedulers, providing special-purpose scheduling within a general-purpose system. One unanticipated benefit of loadable schedulers is that they enable rapid prototyping of new scheduling algorithms by often removing the time-consuming reboot step from the traditional edit/compile/reboot/debug cycle.

Furthermore, and possibly most importantly, providing this form of operating system extensibility frees the operating system designer from having to anticipate all possible scheduling behaviors required by applications. Instead, it allows new scheduling policies to be developed and used without requiring changes to the base operating system.

While this work was conducted within the Windows NT 4.0 kernel, we believe that many of the ideas and implementation techniques presented in this paper should be applicable to other operating systems, providing them the same scheduling flexibility that is present in Vassal.

Acknowledgements

We want to thank Bill Bolosky, Rich Draves, Johannes Helander, and Rick Rashid for their ideas, suggestions, and practical advice during this project. Andrea Arpaci-Dusseau provided useful information on Solaris scheduling classes. Emin Gün Sirer explained fine points of SPIN scheduling. Thanks are also due to Patricia Jones for her technical editing and moral support and to George Jones for the timely use of his computing environment.

References

[Bershad et al. 95] Bershad, S. Savage, P. Pardyak, E.G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 267-284, Dec. 1995.

[Clark et al. 92] D. D. Clark, S. Shenker, and L. Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism" *Proceedings of ACM SIGCOMM '92*, pp. 14-26, Aug. 1992

[Corbató & Daggett 62] F.J. Corbató , M. Merwin-Daggett, R.C. Daley, "An Experimental Time-Sharing System", *Proceedings of AFIPS Spring Joint Computer Conference*, pp. 335-344, 1962.

[Deng & Liu 97] Z. Deng and J. W.-S. Liu. "Scheduling Real-Time Applications in an Open Environment", *Proceedings of the 18th IEEE Real-Time Systems Symposium*, San Francisco, pp. 308-319, Dec. 1997.

[Dusseau et al. 96] A.C. Dusseau, R.H. Arpaci, D.E. Culler, "Effective Distributed Scheduling of Parallel Workloads," *Proceedings of Sigmetrics '96 Conference on the Measurement and Modeling of Computer Systems*, pp. 25-36, May 1996.

[Engler et al. 95] D.R. Engler, M.F. Kaashoek, J. O'Toole Jr., "Exokernel: an Operating System Architecture for Application-Specific Resource Management," *Proceedings of 15th ACM Symposium on Operating System Principles*, pp. 251-266, Dec. 1995.

[Ford & Susarla 96] B. Ford and S. Susarla, "CPU Inheritance Scheduling," *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pp. 91-105, Oct. 1996.

[Goodheart & Cox 94] B. Goodheart and J. Cox, *The Magic Garden Explained: the Internals of UNIX System V Release 4, an Open Systems Design*, Sydney, Prentice Hall, 1994.

[Goyal et al. 96] P. Goyal, X. Guo, H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pp. 107-121, Oct. 1996.

[Hsue & Lin 96] C. Hsueh and K. Lin, "Optimal Pinwheel Schedulers Using the Single-Number Reduction Technique," *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1996.

[Jones et al. 97] Michael B. Jones, Daniela Roşu, Marcel-Cătălin Roşu. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St-Malo, France, pp. 198-211, Oct. 1997.

[Kaashoek et al. 97] D.R. Engler, M.F. Kaashoek, J. O'Toole Jr., "Application Performance and Flexibility on Exokernel Systems," *Proceedings of the 16th ACM Symposium on Operating System Principles*, Oct. 1997.

[Kay & Lauder 88] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, Vol. 31, No. 1, pp. 44-55, 1988.

[Kleinrock 74] L. Kleinrock, *Queueing Systems. Volume 1*, New York: John Wiley, 1974.

[Lampson 68] B.W. Lampson, "A Scheduling Philosophy for Multiprogramming Systems," *Communications of the ACM*, Vol. 10, pp. 613-615, May 1968.

[Liu & Layland 73] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 46-61, Jan. 1973.

[Nieh & Lam 97] J. Nieh and M. S. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications" *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 184-197, Oct. 1997.

[Northcutt 88] J. D. Northcutt, "The Alpha Operating System: Requirements and Rationale" Archons Project Technical Report #88011, Dept. of Computer Science, Carnegie-Mellon, Jan. 1988.

[Ousterhout 82] J.K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proceedings of the 3rd International Conference on Distributed Computer Systems*, pp. 22-30, Oct. 1982.

[Ritchie 84] Dennis M. Ritchie, "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Oct. 1984.

[Seltzer & Small 97] M.I. Seltzer, C. Small, "Self-monitoring and Self-adapting Operating Systems," *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pp. 124-129, May 1997.

[Sha et al. 90] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175-1185, Sep. 1990.

[Solomon 98] David A. Solomon. *Inside Windows NT, Second Edition.* Microsoft Press, 1998.

[Waldspurger 95] C.A. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," Ph.D. dissertation, Massachusetts Institute of Technology, Sep. 1995. Also appears as Technical Report MIT/LCS/TR-667.