Scheduling in K42

Jonathan Appavoo
Marc Auslander
Dilma DaSilva
David Edelsohn
Orran Krieger
Michal Ostrowski
Bryan Rosenburg
Robert W. Wisniewski
Jimi Xenidis

In K42, responsibility for scheduling is divided between user-level and kernel-level code. Moving part of the scheduler to user level reduces kernel interaction and improves performance. Flexibility is increased because applications can tailor the user-level part of the scheduler to their own needs. K42 was designed from the beginning to support two-level scheduling in a multiprocessor environment.

1. Introduction and Motivation

In k42 we partition the scheduler between the kernel and application-level libraries. The K42 kernel schedules entities we call *dispatchers*, and dispatchers schedule threads. A process consists of an address space and one or more dispatchers. Within an address space, all threads that should be indistinguishable as far as kernel scheduling is concerned are grouped under one dispatcher.

A process might use multiple dispatchers for two reasons: to attain real parallelism on a multiprocessor, or to establish differing scheduling characteristics (priorities or qualities-of-service) for different sets of threads.

A process does not need multiple dispatchers simply to cover page-fault, I/O, or system-service latencies, or to provide threads for programming convenience. These requirements can be met using multiple user-level threads running on a single dispatcher. The dispatcher abstraction allows individual threads to block for page faults or system services without the dispatcher losing control of the processor.

Dispatchers tie up kernel resources (pinned memory); threads do not. A process that creates thousands of threads for programming convenience has no more impact on the kernel than a single-threaded process.

In designing the scheduling system, we had the following objectives:

- Performance, especially for critical operations such as in-core page faults and interprocess communication (IPC), should be as good as that of operating systems in which the kernel schedules everything.
- While the system must constrain the physical resources granted to an application (i.e., memory and CPU cycles), it should impose no limit on virtual resources, such as threads, that

use those physical resources. Application threads, for example, should consume no kernel or server resources.

- To support the abstractions that existing research and commercial systems have provided, the fundamental system primitives must provide for: quality-of-service guarantees, background work, and real-time work. Standard threading interfaces such as Posix Threads[PThreads] must be supported efficiently.
- The scheduling system should enforce fairness across entities larger than processes. A user should not be able to gain an unfair advantage merely by creating many processes or threads.
- To support large-scale NUMA multiprocessors, the scheduling infrastructure must allow scheduling operations to proceed independently on different processors.
- To support large-scale parallel applications, the fundamental system primitives must give the application scheduler the ability to manage how work is distributed across the multiprocessor and allow for such specialized policies as gang scheduling.
- Finally, applications with special needs should be able to customize scheduling to support different priority or quality-of-service models, or even to implement concurrency models other than threads (e.g., work crews[Roberts89a]).

We believe our two-level scheduling scheme satisfies these objectives. K42 has been designed from the beginning to both support and exploit this scheduling model.

Section 2 describes the kernel scheduler. Section 3 describes the interface the kernel provides to dispatchers and the interactions between kernel and dispatcher. Section 4 describes the default user-level thread library provided with K42. Section 5 discusses some of the motivation behind our design, and Section 6 relates K42's scheduling work to other multi-level scheduling research.

2. Kernel Scheduler

2.1. Resource Domains

Each dispatcher belongs to a resource domain, and it's the resource domain, not the dispatcher, that owns the rights to a share of the machine's CPU resources. We expect that separate resource domains will be associated with each user, so that users will receive fair shares of the machine no matter how few or how many processes they create.

On a multiprocessor, a resource domain owns rights to each CPU independently. A resource domain might, for example, own rights to half of each of four CPUs and to no part of any others. Each dispatcher is bound to a particular CPU and uses the CPU resources of its resource domain on its CPU. The kernel may move a dispatcher from one CPU to another for load-balancing purposes, but such migrations are expected to occur only on a fairly coarse time scale.

2.2. Scheduling Algorithms

The kernel scheduler runs independently on each processor. At each decision point it chooses a runnable resource domain and then chooses a runnable dispatcher from that domain.

The runnable dispatchers in a given domain are linked in a ring, and every time the scheduler chooses to run the domain, it moves to the next dispatcher in the ring. The scheduler makes no

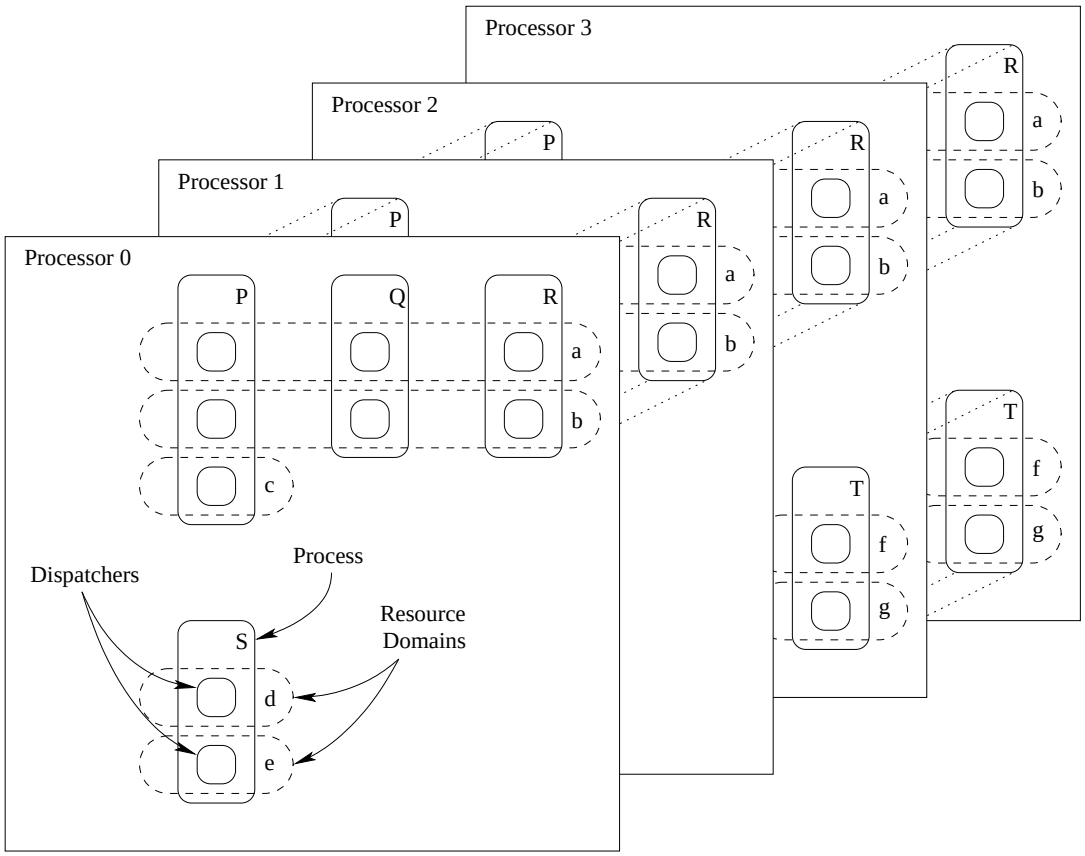

Figure 1. Processes, Resource Domains, and Dispatchers

attempt to give equal time to the different dispatchers. If some sort of fairness among dispatchers is needed, they should belong to different resource domains.

The kernel's scheduling policies apply to resource domains. A resource domain is said to be runnable if at least one of the dispatchers that belong to it is runnable, and "running" a resource domain means running one of its dispatchers. Each resource domain belongs to one of five absolute priority classes:

1. system and hard-real-time
2. gang-scheduled
3. soft-real-time
4. general-purpose
5. background

On each processor, a runnable resource domain belonging to a given level will run to the exclusion of all domains belonging to higher-numbered levels.

Proportional-share scheduling is used within each priority class. Each resource domain is assigned a weight (independently on each CPU to which it has rights), and at each priority level, whatever portion of a CPU is not consumed by higher-priority domains is apportioned among

domains at the given level according to their weights. Exponential decay is used to ensure that no domain can accumulate an excessive claim on a CPU by remaining unrunnable for a long period.

In K42, many "system" services (e.g. file systems, pseudo-terminals, etc.) are implemented as user-level processes. We expect such processes to reside in the general-purpose priority class but with weight sufficient to ensure that they never get squeezed out by ordinary applications. The only services that must run in the system priority class are those that are directly involved in kernel scheduling.

A higher-level resource manager is in charge of assigning dispatchers to resource domains and of setting the priority levels and weights of those domains. By limiting admission to the gang-scheduled and real-time priority classes, the resource manager can guarantee CPU resources to privileged processes, while allowing whatever is left over to be shared fairly by a general-purpose workload. A separate white paper is devoted to real-time support in K42. We include gang-scheduled applications in the real-time discussion because we expect to schedule such applications using localized algorithms based on hardware- or software-synchronized clocks. But for that to work, the application must have a very high priority during the intervals it's supposed to run.

All CPU time-accounting (which drives scheduling decisions) is done at the clock resolution of the machine. We use the term *quantum* only to refer to the maximum amount of time the kernel will allow a process to run before making a new scheduling evaluation. Along with the priority class and weight, the quantum size is a parameter of a resource domain that can be set by the resource manager.

See Section 4.6.2 for an example of how resource domains are used to implement a particular policy, in this case a policy that tries to provide low latency for I/O-bound threads and high throughput for CPU-bound threads.

2.3. Hard- and Soft-Preemption

When the kernel scheduler determines that the currently running dispatcher should be suspended in favor of some other dispatcher, it initiates either a *soft* preemption or a *hard* preemption.

A soft preemption is attempted if the two dispatchers are in the same priority class and if the currently running dispatcher has been well-behaved. In a soft preemption, the running dispatcher is interrupted (via a mechanism described later) but is allowed to continue long enough to get itself into a clean state and to voluntarily yield the processor. After a successful soft preempt the preempted process "owns" all of its own machine state. No machine state for the process is saved in the kernel.

If the priority class of the currently running dispatcher is lower (higher-numbered) than that of the should-be-running dispatcher, or if the current dispatcher hasn't responded in a timely manner to a previous soft preempt attempt, the current dispatcher is hard-preempted. In a hard preempt, the dispatcher is stopped in its tracks and its machine state is saved in the kernel. Because it's not in a clean state, a hard-preempted dispatcher cannot accept incoming synchronous interprocess messages.

A running dispatcher is soft-preempted when its quantum expires, even if its domain remains at the head of the kernel dispatch queue. This policy gives other dispatchers in the same domain a chance to run (albeit with no attempt at fairness), and it lets dispatchers use preemption requests to drive internal time-slicing.

2.4. Synchronous Interprocess Communication

The primary interprocess communication mechanism in K42 is the *protected procedure call*, or *PPC*, which at a high level is the invocation of a method on an object that resides in another address space. The mechanism is designed to allow efficient communication between entities in different address spaces but on the same physical processor. The PPC design, rationale, and implementation are the subjects of another white paper. Here we address the interaction between the PPC mechanism and the kernel scheduler.

The kernel service that supports the protected-procedure-call mechanism is a pair of system calls (one for call and one for return) that transfer control from one dispatcher to another, switching address spaces along the way. Most register contents are preserved from sender to receiver, so registers can be used to pass arguments and return values.

A call or return is an explicit handoff of the processor from one dispatcher to another, and we had to decide how to integrate such handoffs into the resource-domain-based kernel scheduling infrastructure. We considered and rejected two "pure" approaches.

Under one approach, we would formally switch from the sending dispatcher's resource domain to the receiver's domain on every call and return. CPU time would always be charged accurately to the resource domain of the running dispatcher. A call or return to a dispatcher whose resource domain wasn't entitled to run (given its priority class, weight, and past CPU usage) would result in an immediate soft or hard preemption. This approach is easy to describe and it meshes cleanly with the kernel scheduler, but involving the scheduler in every call and return would severely degrade the performance of the PPC mechanism, and extreme PPC efficiency is fundamental to the overall design of the K42 system.

Under the second approach we considered, services invoked via PPC would always execute in the resource domains of their callers rather than in their own domains. A call would switch dispatchers (and address spaces) but would not change the current resource domain (to which CPU time is being charged). A return would switch back. Servers would consume the CPU resources of their clients and would be subject to their clients' resource constraints. We would provide a mechanism by which a dispatcher in a server process could switch from working in the resource domain of one of its clients to working in the domain of another if it internally switches the client it's servicing. This model allows for an efficient PPC implementation, and it has the advantage that CPU time spent in server processes is charged to the appropriate clients. It adds complexity to the kernel scheduler, because server dispatchers would sometimes have to be scheduled as if they belonged to their clients' resource domains, and because the kernel would need some way to authenticate the requests servers make to re-enter client domains. The real drawback to the model, however, is that it leads to priority inversion problems in server processes. A server thread working on behalf of a highly constrained client might manage to acquire a lock just before running out of CPU time, and might thereby degrade the service provided to other clients. The traditional solution to such problems would involve changing all our locks to ones that support priority inheritance. Such locks are inherently more costly in both

time and space than our lightweight locks, and we want to use them only where they're really needed.

We've settled on a compromise approach that, while it's not as clean semantically as the approaches outlined above, allows for an efficient implementation and does not induce priority inversion problems in servers. On a call, we switch dispatchers and address spaces without involving the scheduler and without switching resource domains. We say that the called dispatcher is executing in a resource domain *borrowed* from the caller. If the server completes the request without incident (the common case), the return transfers control back to the caller and the kernel scheduler never knows that the call happened. All the service time is effectively charged to the caller's resource domain. If, however, an interrupt, page fault, or other event invokes the kernel scheduler while the request is being serviced, the scheduler notices that the currently-running dispatcher is executing in a borrowed domain. At that point, it charges any accumulated CPU time to the borrowed domain and makes the server dispatcher runnable in its own domain. The remaining request processing will be charged to the server's domain and will run with the server's priority and weight, precluding any priority inversion problems in the server. When the scheduler takes a dispatcher out of a borrowed domain, it also arranges things so that the eventual PPC return from that dispatcher is diverted from the normal fast path and instead goes through the scheduler. Otherwise the original client would wind up executing in the server's resource domain, which would constitute an unacceptable transfer of resources. The PPC return diversion is accomplished without adding cycles to the normal fast return path.

The two approaches we considered and rejected are pure in a theoretical sense, in that it's always known who pays for service requests. In the first approach, the server pays for all request servicing and would have to have an alternative mechanism for charging costs back to clients. In the second approach, the clients pay for all requested services. In the compromise model we've adopted, the accounting is not as precise. Most request-servicing times are charged to the clients making the requests, but more-or-less random events will cause some fraction of the service times to be charged to the server. We believe the imprecision is acceptable, given the efficient PPC and locking implementations the model allows.

2.5. Exception Level

Kernel scheduling, along with low-level interrupt and exception handling, constitutes a layer of the kernel we call *exception level*. Exception-level code is characterized by the fact that it runs with hardware interrupts disabled and accesses only pinned data structures that are not read-write shared across processors. Higher-level code that needs to interface with exception level synchronizes with it by disabling hardware interrupts.

2.6. Kernel Process

Non-exception-level kernel functionality runs in the context of a kernel process that in most respects is like any other process. It has dispatchers (a primary and an idle-loop dispatcher) on each physical processor, and the dispatchers use the same library code as user-level processes to support multiple threads in the kernel process. Kernel-process threads (or more simply, kernel threads) can access non-pinned kernel data structures and structures that are read-write shared across processors. Locks are used for synchronization, just as they are in user-level processes. Almost all kernel services are implemented as services exported by the kernel process

via the normal protected procedure call mechanism. That is, a user process accesses most kernel services by making calls on the kernel process in the same way that it makes calls on other server processes. The only real system calls are those that implement the protected procedure call mechanism itself plus a few that directly affect low-level scheduling (request-timeout and yield-processor, for example).

2.7. Reserved Threads

When a user-level dispatcher makes an exception-level request (either an explicit system call or a page fault or trap), and the request cannot be handled entirely at exception level, the request is handled on a kernel thread working on behalf of the user-level dispatcher. Examples of such requests are messages that cannot be delivered locally but must instead be delivered to other processors, and all user-mode page faults. A dispatcher is allowed to have at most one such request outstanding at any time, and we guarantee that the kernel process has enough threads so that every user-level dispatcher can have one working on its behalf. We use the term *reserved thread* to refer to a thread working on behalf of a particular dispatcher. Reserved threads, however, are not permanently bound to dispatchers but are allocated from the normal thread pool as needed and then released. Threads are reserved in the sense that attempts to allocate threads for other uses will fail if the allocation would leave fewer threads in the pool than might be needed, and attempts to create new dispatchers will fail if the kernel can't spare the requisite number of threads. Reserved threads are dynamically assigned, not to reduce the number of threads the kernel must have, but to allow threads to be serially reused with different dispatchers, resulting in better cache behavior for kernel thread stacks.

A reserved thread is responsible for its dispatcher. It keeps track, if necessary, of the dispatcher's machine state and sees that the dispatcher is resumed at an appropriate time. While handling its request, a reserved thread may block for locks or may suffer page faults on pageable kernel data structures. When it has completed its request, it resumes its dispatcher immediately unless conditions have changed such that its dispatcher isn't the one that should be running. In that case it waits until its dispatcher is again chosen to run by the kernel scheduler and then resumes it. A hard preemption is a special-case use of the reserved thread in which the thread has no particular task to perform but goes straight to waiting for its dispatcher to again be chosen to run.

For kernel scheduling purposes, a dispatcher's use of its reserved thread is handled much as if the dispatcher made a protected procedure call to the kernel process. That is, we switch to running a kernel-process dispatcher but we remain in the client dispatcher's resource domain as long as the reserved thread isn't interrupted or blocked. In this way most interactions with the kernel process bypass the kernel scheduler.

2.8. Page-Fault Handling

When a thread running under a user-level dispatcher suffers a page fault, a kernel thread is allocated to be the dispatcher's reserved thread and the fault information is passed to it. It traverses kernel data structures, which may themselves be pageable, to classify the fault into one of three categories: a bad-address fault, an in-core fault, or an out-of-core fault. A bad-address fault is reflected back to the dispatcher as a trap (via a mechanism described later). An in-core fault is resolved (by establishing a valid mapping for the faulting address) and the dispatcher is

resumed where it left off. For an out-of-core fault, the reserved thread initiates the I/O operation needed to resolve the fault and then hands control back to its dispatcher with an indication that it has suffered a page fault. The dispatcher sets aside the thread it was running and runs another thread if it has one. Later, when the I/O operation completes, the dispatcher gets a notification that the page fault has been resolved, and at that point it can resume the thread that had been running. The mechanisms for reflecting a fault back to the dispatcher and for notifying it of a fault completion are described in more detail later.

Kernel-mode page faults are handled in another way because they occur in the process (the kernel process) that is itself responsible for resolving faults. A kernel-mode fault can occur only when a kernel thread is running, because exception-level code and critical dispatcher code in the kernel never access pageable kernel memory. Such a fault is handled on the thread that suffered it. The fault-time machine state is pushed on the thread's stack, and the thread is resumed in fault-handling code. This code resolves the fault if it is an in-core page fault, and it initiates the I/O operation and waits for it to complete otherwise. In either case it eventually restores the machine state from its stack and returns to the point at which the fault occurred. We guarantee that the data structures that are traversed in handling a kernel-mode page fault are pinned, so that we never take a second kernel-mode fault.

3. Kernel/Dispatcher Interface

The interface the kernel provides to a dispatcher is in many ways analogous to the interface a hardware processor provides to an operating system.

3.1. Dispatcher Structure

The interface is centered around a memory region, called the dispatcher structure, that is read-write shared between the dispatcher and the kernel. The structure contains constructs corresponding to a disable-interrupts bit, a pending-interrupts bit vector, a machine-state save area, and various other control and status registers and message buffers. Constructs corresponding to an interrupt-dispatch vector and a timer control register are also provided, but these are manipulated through a system-call interface rather than through the shared-memory structure, so that the kernel doesn't have to poll for changes.

When a dispatcher is running, the kernel provides a pointer to the corresponding dispatcher structure in a well-known location in the process's virtual address space. This virtual location is part of a read-only page of information that the kernel shares with all processes. Some of the information in the page is specific to the processor on which it is accessed, so the virtual page is mapped to different physical pages on different processors. The kernel changes the dispatcher pointer for the current processor when it chooses a new dispatcher to run.

Library code can extend the dispatcher structure with user-level scheduling information that is not part of the kernel/dispatcher interface. Extending the dispatcher structure in this way lets us address all dispatcher-specific information, both shared and private, via the single dispatcher pointer the kernel provides.

3.2. Entry Points

When it's created, a dispatcher initializes (via system call) a vector of entry points, which are basically code addresses at which it wants to be entered under various conditions. There are entry points for starting to run in a clean state, for being interrupted to handle an asynchronous event, for being informed that a synchronous trap has occurred or that an out-of-core page fault has been suffered, for accepting an incoming protected procedure call or reply, and for being informed that an outgoing call or reply has failed. Each entry point has its own set of semantics concerning the state of various machine registers and constructs in the dispatcher structure.

As an example, consider the trap entry point. This entry point is invoked when a running dispatcher executes a trap instruction, divides by zero, or otherwise does something that raises a synchronous hardware exception. Bad-address page faults are also reported as traps. When the trap code is entered, the fault-time content of a subset of the machine registers will have been saved in a reserved area of the dispatcher structure. The particular set of registers saved in this way is architecture-dependent, but it generally includes the registers that are volatile according to the architecture's C-language calling conventions, plus basic things like the program counter and stack pointer. All registers that were not saved in the dispatcher structure are preserved from the time of the fault. The current content of the registers that were saved is mostly undefined. The obvious exception is the program counter, whose current content is the address of the trap entry point code itself. Other exceptions are a few volatile general-purpose registers that are used to convey architecture-dependent information about the particular trap that occurred. The content of the stack pointer register is not defined when the entry point is invoked. The lowest-level interrupt and fault handlers in the kernel are programmed to save necessary machine state directly in the current dispatcher structure, so that the state can be passed up to the dispatcher, if necessary, without copying.

Other entry points have semantics of the same flavor. They range from the *run* entry point, for which the content of nearly all the registers is undefined, to the protected procedure call and return entry points, for which the content of nearly all the registers is defined to be preserved from the sender.

Note that the page-fault entry point does not ask the dispatcher to handle a page fault (ala ExoKernel[Engler95]), but merely informs the dispatcher that a fault has occurred so that it can run something else. A tag identifying the fault is passed to the dispatcher, and a later asynchronous notification tells it when the faulting code associated with the tag can be resumed. Actually resolving the fault is the kernel's responsibility.

3.3. Disabled Flag

Dispatcher code has critical sections during which new entry point invocations cannot be tolerated. A *disabled* flag in the shared dispatcher structure is used to protect such critical sections. The kernel sets the disabled flag before invoking an entry point, and until the dispatcher clears the flag the kernel will not invoke the same or any other entry point. What it does instead depends on the particular entry point it would like to invoke. For example, if it's a protected procedure call or return that the kernel is attempting to deliver, the message is instead reflected back to the sender so that it can be re-sent later.

A dispatcher can set the disabled flag any time it enters critical code. In particular, it can use the flag to avoid race conditions involving any of the low-level scheduling system calls (for exam-

ple, to avoid yielding the processor just as a thread becomes runnable because of a page-fault completion notification). The kernel expects the disabled flag to be set when a dispatcher makes a scheduling or IPC system call, and it clears the flag once the system call has been validated. This protocol leaves the dispatcher in a clean state when it makes an outgoing protected procedure call, so it can be rescheduled (perhaps to run another thread) if the call happens to block in the called process.

A page fault while disabled is an interesting case. The dispatcher cannot be informed of such a fault even if it's an out-of-core fault, so instead the reserved kernel thread for the dispatcher blocks in the kernel until the I/O completes and then resumes the dispatcher without it ever knowing the fault occurred. Disabled page faults are expensive in that they block the entire dispatcher while they are resolved, so dispatcher code should and does limit its references to virtual storage while disabled to those structures directly involved in scheduling. It's hoped that those structures will stay "hot" so that disabled page faults are infrequent.

The trap entry point is the one exception to the rule that no entry point will be invoked while the dispatcher is disabled. If the dispatcher divides by zero or references a bad address, the trap has to be reported even if the dispatcher is disabled. For that reason the trap entry point uses a state save area in the dispatcher structure separate from the save area used by all the other entry points. Also, the old value of the disabled flag is passed to the trap entry point so that it can restore it if the trap is one that can be handled (a breakpoint trap, for example). Traps in the trap entry point code itself cannot be handled.

3.4. Software Interrupts

All types of asynchronous events for a dispatcher are mapped to bits of a software pending-interrupts vector in the dispatcher structure. There are bits that signal timeouts, soft preempts, page-fault completions, asynchronous message arrivals, and dispatcher-to-dispatcher interrupts. All setting and clearing of software interrupt bits is done with non-blocking atomic operations to avoid losing interrupts.

Many of the software interrupt bits are associated with secondary data structures that carry more information about the associated asynchronous events. The primary interrupt bits are collected in one word so that they can all be checked at once. A primary bit being on is an indication that its associated secondary structure needs attention. For example, the primary page-fault-completion interrupt bit is associated with a second bit vector that is indexed by the tags that identify outstanding page faults. Several such faults may complete while a dispatcher isn't running, and the secondary bit vector records them all. Another example is the asynchronous message mechanism. The dispatcher structure contains a pair of circular buffers for incoming asynchronous messages, together with their head and tail pointers. One buffer is for messages generated locally and the other is for messages arriving from other processors. The buffers are split because the synchronization requirements are different for the two cases. A single primary software interrupt bit is used to indicate the fact that new messages have been deposited.

A dispatcher checks its software interrupt vector whenever it's entered at its *run* entry point. It processes any bits that are on while it's still disabled, but often handling an interrupt involves little more than creating or unblocking a thread to do the real work after enabling. If the dispatcher is already running when an event arrives, it is re-entered at its *interrupt* entry point. The machine state as of the time of the interrupt is passed up much as it is for the trap entry point.

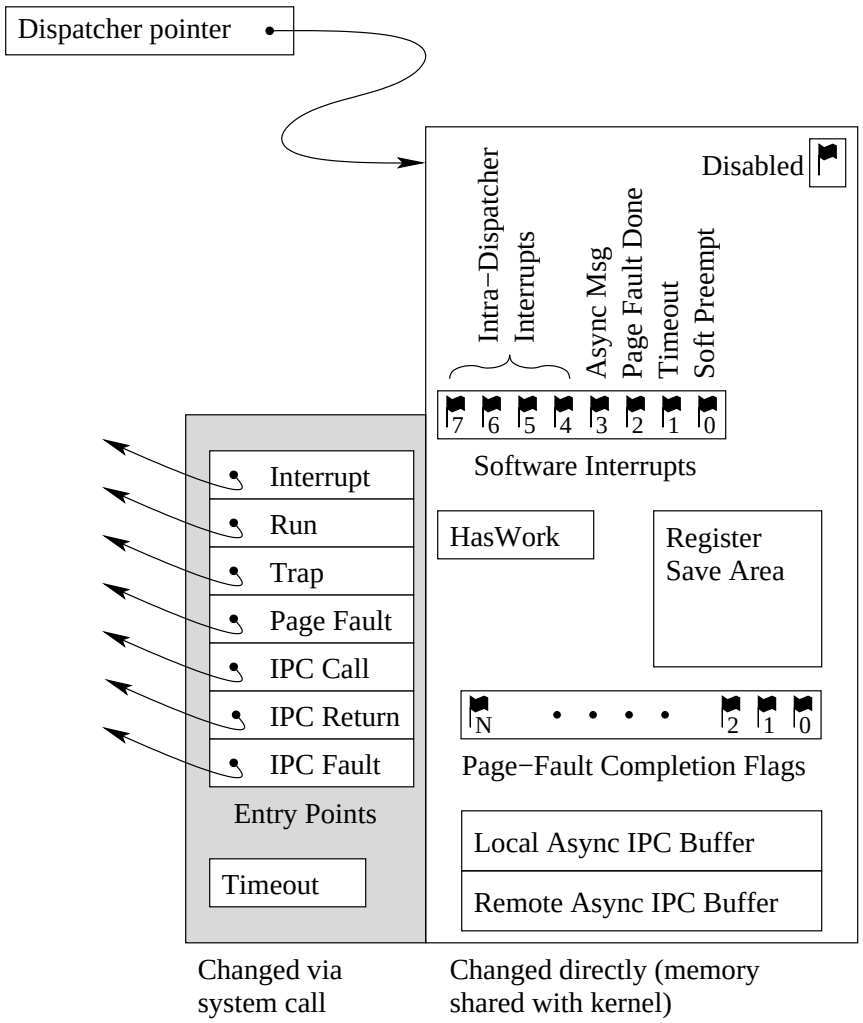

Figure 2. Dispatcher Structure

The interrupt entry point code processes the new software interrupt(s), and then has the option of resuming the thread that was running or of suspending that thread and running another. If it chooses to resume the current thread, it does so without ever having explicitly saved the non-volatile registers. It simply reloads the volatile registers from the save area in the dispatcher structure before re-enabling. Otherwise, it copies the state stored in the dispatcher structure onto the current thread's stack, saves the non-volatile registers, also on the current stack, and then chooses another thread to run.

If a dispatcher is disabled when a software interrupt arrives, the kernel sets the interrupt bit but does not interrupt the dispatcher. Dispatchers are expected to check their software interrupt vectors each time they clear their disabled flags in order to catch interrupts that arrived while they were disabled.

When a dispatcher does something to cause a kernel-process reserved thread to begin working on its behalf, the dispatcher itself is marked as disabled. Otherwise it might be re-entered, allowing it to do something that would require a second reserved thread. When the reserved thread finishes its task and returns control to the dispatcher, it clears the disabled flag if it was not set originally. In that case the kernel must check for pending interrupts that may have appeared while the dispatcher was disabled, and if there are any it must resume the dispatcher with a software interrupt rather than where it left off.

The setting of a software interrupt bit when the interrupt vector was previously clear is the only point at which an unrunnable dispatcher can become runnable. The kernel notices such transitions and uses them to cause new scheduling evaluations. No re-evaluation is necessary if other software interrupt bits were already set or if the target dispatcher was already runnable. A dispatcher remains runnable as long as its disabled flag is set, it has pending software interrupts, or a third dispatcher structure field, the *HasWork* field, is non-zero. The dispatcher uses the HasWork field to keep itself runnable after soft-preemptions and after making an outgoing protected procedure call. The kernel treats the HasWork field as a zero/non-zero boolean, but the field is in fact a full 64-bit word so that dispatcher code can store useful information there. K42's default dispatcher implementation uses the field to anchor its list of runnable threads.

Note that soft-preempt requests are delivered as software interrupts rather than via a special entry point. This design allows a dispatcher to respond to such a request (and avoid a hard preemption), even if the request arises at a time when the dispatcher happens to be disabled (assuming the disabled interval is short).

3.5. Dispatcher-to-Dispatcher Interrupts

The kernel provides a service by which code running in one dispatcher can raise a software interrupt in another dispatcher in the same process. The service is efficient because the kernel can set the requested bit (using an atomic *fetch-and-OR* operation) in the target dispatcher structure, even if that structure is on another processor. If the interrupt flags word was already non-zero, no further action is necessary. Otherwise the kernel must make the target dispatcher runnable, if it isn't already. If the dispatcher is on another processor, the kernel must send an exception-level request to that processor, but it needn't wait for the request to be processed.

3.6. Timeouts

A dispatcher can have one outstanding timeout request registered with the kernel. Changes to the requested timeout time are made via system call so that the kernel doesn't have to repeatedly check some field of the dispatcher structure. When the requested time arrives, a software timer interrupt is generated. This interrupt may make the dispatcher runnable when it wasn't previously, but whether or not it actually runs depends on the priorities of other runnable dispatchers.

4. User-Level Scheduler

The K42 two-level scheduling model allows library code to provide user-level thread implementations tailored to specific applications. This section describes the default thread implementation provided with the system.

4.1. Thread Objects

Each thread is represented by a small object that contains the facts needed to manage the thread. Any thread other than the one currently running will have its stack pointer saved in its thread object. All other machine state for the thread (including the thread's program counter) is stored on the thread's stack. Resuming a thread requires loading its stack pointer value into the stack pointer register, retrieving the program counter value from a fixed offset from the stack pointer, and branching to that location. Any further state restoration is the responsibility of the branched-to code, and can range from none (for a newly-created thread that hasn't run before) to a full machine-state restore (for a thread that was suspended involuntarily because of a page fault or preemption). Threads that suspend themselves voluntarily (by blocking or yielding) save and restore just that part of the machine state that is "non-volatile" according to the the C-language calling conventions of the architecture.

4.2. CurrentThread

An always-accessible location known as *CurrentThread* is used to hold a pointer to the thread object of the currently-running thread. On architectures whose ABIs reserve a register that can be used for this purpose (PowerPC, for example), CurrentThread is simply a symbolic name for this register. On other architectures, CurrentThread is bound to a well-known location in the process's virtual address space. Like the dispatcher pointer, this location is mapped to different physical pages on different processors so that each processor can have its own content. This "pseudo-register" is supported by the kernel on architectures that need it.

The ability to obtain the current thread pointer directly, without going through the dispatcher pointer, is necessary because a thread can migrate from one dispatcher to another. Once the dispatcher pointer is loaded into a register, the register value may no longer point to the current dispatcher. A thread can make itself non-migratable by setting a flag in its thread object (which it can reach through CurrentThread). Only when it is non-migratable is it safe for a thread to access the dispatcher structure. Migration is discussed in more detail later.

4.3. Thread IDs

Every thread is assigned a *threadID*. This 64-bit handle identifies both the dispatcher on which the thread is running and the particular thread within that dispatcher. A new threadID is assigned when a thread migrates from one dispatcher to another, so a remembered threadID remains valid only while the thread it designates remains non-migratable.

We use threadIDs rather than thread-object pointers to refer to threads for two reasons.

First, it's easy to check the validity of a threadID, so it's safe to use threadIDs to refer to threads in other processes. For example, in an outgoing protected procedure call, we include the threadID of the thread making the call. We expect the callee to return the threadID to us in its reply so that we can match the reply with the thread waiting for it. We validate the threadID provided by the callee before using it, a precaution that wouldn't be so easy if we were passing around thread pointers instead of threadIDs.

Second, it's possible to tell from a threadID alone whether or not the designated thread resides on the current dispatcher, and therefore whether or not it's possible to perform a scheduling operation locally. A prime example is the *unblock* operation, which takes a threadID as its parameter. A quick check of the threadID tells us whether we can complete the operation locally or instead must ship it to the target thread's dispatcher. In the latter case, we know where to send the operation just by looking at the threadID, avoiding the expensive cross-processor cache miss that accessing the thread object itself would most likely involve.

4.4. Thread Creation

New threads are created either by explicit request or implicitly as a result of incoming protected procedure calls. The parameters to a thread creation request are a pointer to a function the thread is to execute and a one-word argument to be passed to that function. The thread library handles the request by allocating a thread object from a per-dispatcher free-list (assuming the list isn't empty), initializing a few words of the associated thread stack to cause the thread to execute the requested function when it runs, and appending the thread to the dispatcher's ready queue. Creating a thread to handle an incoming PPC is even less work, because the thread is made to run immediately and most of its register state comes directly from the caller.

When threads terminate, their thread objects are pushed back on the per-dispatcher free-list of threads available for reuse. The free-list is managed as a stack so that recently-terminated threads are reused before their thread objects and stacks are flushed from the cache.

Creating a new thread (assuming the free-list isn't empty) is cheaper than unblocking an existing one, because there's less machine state involved. For this reason we tend to avoid using long-running threads that block waiting for timers or asynchronous notifications, and instead create new threads to handle such events.

When the free-list is empty and a new thread is needed, we allocate space for a thread object and stack together. The thread object is located at the base of the stack so that the object doesn't wind up in a virtual page that is otherwise unused. In an application, stacks can be ridiculously large, given the 64-bit virtual address space. Kernel-process thread stacks are pinned, so they should be as small as possible. The new thread object is initialized and a threadID is assigned. At this point the new thread can be used as if it had been allocated from the free-list.

4.5. Block/Unblock Semantics

Two principal operations provided by the K42 thread implementation are *block* and *unblock*. A thread calls block to make itself unrunnable. It must first store its own threadID in some data structure so that other code can later unblock it. The thread must be non-migratable before blocking, in order for its threadID to be well-defined. As described above, unblock takes a threadID as parameter and makes the designated thread runnable again, either directly or by forwarding the request to the thread's dispatcher. Matching block and unblock calls are allowed to occur in either order, so the blocker's store-threadID-then-block sequence does not have to be atomic.

4.6. Thread Migration

Threads are migrated from one dispatcher to another for two reasons – for balancing a workload across processors and for changing the quality of service provided to particular threads.

4.6.1. Load Balancing

When a workload consists of a number of independent tasks whose execution times are not known a priori, it can happen that several long-running threads wind up on one dispatcher while dispatchers on other processors sit idle. Our default threading library solves this problem by moving threads from busy to idle dispatchers. The time scale for migration decisions is fairly large, because even though it's mechanically easy to move a thread on a shared-memory multiprocessor, the cost in terms of cross-processor cache misses can be large.

4.6.2. Quality-of-Service Changes

The threads running on a given dispatcher are not visible to the kernel and are therefore all of equal importance as far as kernel scheduling is concerned. The only way to change the kernel-level scheduling characteristics of a thread is to move the thread to a dispatcher whose resource domain has the desired characteristics. Migrating threads between dispatchers on the same processor is much less expensive than moving them across processors, so applications can use this technique for even relatively fine-grained priority changes.

The K42 default threading library uses migration to give precedence to I/O-bound threads over CPU-bound threads, both within a process and across the processes belonging to a particular user. By default, every user is assigned two general-purpose resource domains, and every process the user creates will have one dispatcher in each domain (on each processor the process uses). The two domains have the same priority class and weight, but they are used in such a way as to maintain a higher precedence for the I/O-bound domain under the kernel's proportional-share scheduling algorithm. Specifically, threads that run for a long time without blocking are migrated to the dispatcher in the CPU-bound domain, while threads that block frequently are moved to the I/O-bound dispatcher. By under-utilizing its share of the processor, the I/O-bound domain can provide low-latency response for its threads when the I/O operations on which those threads are blocked complete. (Note: The I/O-bound/CPU-bound migration machinery is in place, but the assignment of domains to users is as yet incomplete.)

4.7. Intra-process Communication

If an application is spread across several dispatchers, either for parallelism or because parts of the application run with different service requirements, the different parts may need to communicate among themselves. Shared memory is the natural transport mechanism for such communication, and the dispatcher-to-dispatcher interrupt mechanism provides the asynchronous notification capability needed for a complete intra-process communication facility. Several software interrupt bits are reserved for this purpose.

The K42 library provides an efficient facility for sending messages to other dispatchers. Messages are aligned on cache-line boundaries to minimize cross-processor cache traffic, and interrupts are sent only when a message queue makes the transition from empty to non-empty. Separate queues are used for requests that can be handled at dispatcher level (with the dispatcher disabled flag set) and requests that must be handled on threads. Both one-way and round-trip communication models are provided.

4.8. Timers

The K42 library maintains an ordered set of outstanding timeout requests. Only the "nearest" timeout is registered with the kernel. All other requests are managed in the application's own address space. A fixed set of kernel resources can thereby support an arbitrary number of application timers. Moreover, many timeouts may be requested and later cancelled without ever involving the kernel, a clear performance advantage. Cancelling a timeout is as common an operation as requesting one, because most timeouts are established in order to catch exceptional events that don't happen.

4.9. PThreads

The K42 system provides a Posix Threads[PThreads] implementation layered on top of the default K42 threading library. The pthreads implementation extends (via subclassing) the basic thread object with pthreads-specific information. Since the pthreads extension and the basic thread object are co-located, the CurrentThread pointer can be used to reach either, resulting in a very efficient implementation of *pthread_self()*.

5. Discussion

This section is still a work in progress.

6. Related Work

This section is still a work in progress.

References

[Engler95] *Exokernel: An operating system architecture for application-level resource management*, Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr., 1995, Proceedings 15th Symposium on Operating Systems Principles, 251-267.

[K42 LKIntern] *Utilizing Linux Kernel Components in K42*, Jonathan Appavoo , Marc Auslander , Dilma DaSilva , David Edelsohn , Orran Krieger , Michal Ostrowski , Bryan Rosenburg , Robert W. Wisniewski , and Jimi Xenidis , October 2001, www.research.ibm.com/K42.

[K42 Linux Environment] *K42 Linux Environment*, Jonathan Appavoo , Marc Auslander , Dilma DaSilva , David Edelsohn , Orran Krieger , Michal Ostrowski , Bryan Rosenburg , Robert W. Wisniewski , and Jimi Xenidis , October 2001, www.research.ibm.com/K42.

[K42 Overview] *K42 Overview*, Jonathan Appavoo , Marc Auslander , Dilma DaSilva , David Edelsohn , Orran Krieger , Michal Ostrowski , Bryan Rosenburg , Robert W. Wisniewski , and Jimi Xenidis , October 2001, www.research.ibm.com/K42.

[Pike93] *The Use of Name spaces in Plan 9*, Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom, 27(2), 72-76, April 1993, Operating Systems Review.

[PThreads] *The Single UNIX® Specification, Version 2*, The Open Group, 1999, http://www.opengroup.org/ .

[Roberts89a] *WorkCrews: An Abstraction for Controlling Parallelism.*, E. Roberts and M. Vandevoorde, Digital Equipment Corporation, Systems Research Centre, February, 1989.