1 Multiprocessors

As we saw earlier, a **multiprocessor** consists of several processors sharing a common memory. The memory is typically divided into several modules which can be accessed independently. Thus several (non-conflicting) memory requests can be serviced concurrently A switching network is used to direct requests from processors to the correct memory module. Each processor has its own cache.

Converting a uniprocessor operating system to a multiprocessor requires a way to exploit the concurrency provided by the hardware. Here are some approaches:

Let each processor handle a different set of devices and run different operating systems. This situation resembles the distributed case except that we have shared memory.

Let one processor execute processes and another handle the devices.

Let one processor support user processes and another support communication among them. This is a useful idea when interprocess communication is widely used, as is the case in server-based operating systems.

Let one processor be the master processor and others be slave processors. The master processor runs the complete OS, while the slave processors simply run processes, leaving the master processor to handle system calls and interrupts.

All the cases above are asymmetric. We can also develop a symmetric multiprocessor system: Treat the machines equally, and divide the ready processes among them. Under this approach, a device interrupts whichever machine currently allows interrupts. All the processor equally share the OS code and data structures. We will consider only the symmetric case in this course.

In all cases except the first one, code running on different processors shares common data structures, and thus needs process coordination primitives. The coordination schemes we have seen so far will not work since disabling interrupts on a processor does not prevent processes from executing on other processors. Thus these multiprocessor systems support busy waiting or *spin-controlled* coordination schemes in which processes spin in a loop waiting to be unblocked. We will not look explicitly at multiprocessor coordination schemes: take Jim Anderson's courses to learn more about them, in particular, lock-free schemes.

1.1 Kinds of Processes

A multiprocessor system can execute application processes simultaneously. What kind of processes should it support? One simple approach is to use a Unix or Xinu like approach of supporting one kind of process in the system, which are scheduled by the OS. In the Xinu case, it would be a lwp while in the Unix case it would be a hwp. The kernel would simply now schedule as many processes as there are processors.

A problem with this approach is that any practical system, unlike Xinu, would support hwps. If we support only one kind of processes, then the cost of context switching the hwps would be high (requires entering the kernel, changing page tables). Thus, this approach does not encourage fine-grained parallelism because of the high cost of concurrency.

Therefore the solution is to support both lwps and hwps. For a process to be truly a lwp, not only must it not require loading of translation tables, but it must also not require kernel code to do the context switching, because of the high cost of entering the kernel, which not only involves the cost of the kernel trap but also copying and checking of parameters. Threads supported by the kernel have been found to be ten times as

costly to schedule as threads supported by user-level code. (The cost of scheduling of the latter has been found to be within an order of magnitude of the cost of a procedure call.) Kernel-scheduled threads also incure the space cost of a kernel stack per thread. Another issue is fairness - the more threads an application has, the more of the CPU time it gets. Therefore, the most efficient solution seems to be to support lwps in user code within a hwp, as in Java or 730-xinu.

The problem with this solution of course is that user-level code does not have the rights to bind processes to processors and thus cannot schedule lwps on multiple processors. The kernel has the rights to do so, but it does not know about the lwps, since they are implemented entirely in user-level code. So we need a more elaborate scheme to give us fine-grained concurrency at low cost. One solution is to have a mixed scheduling scheme for lwps wherein both the kernel and user-level code work together to schedule threads. This is what happens with p-threads in the BSD OS - the thread data structures (context blocks) are known to both user-level and kernel code. When a p-thread makes a blocking call, the kernel invokes a callback in the user-level scheduler informing it that a blocking call has been made. The scheduler can then schedule some other thread and reschedule the original thread when it gets another notification saying that the blocking call has completed. As a result, a system blocking call by a thread does not block the entire application. We can imagine similar cooperation between the kernel and and user-level scheduler to allow threads to be scheduled on multiple processors. This solution is intended to retain compatibility with existing operatint systems in which address spaces and processes are coupled together.

To understand how we might support thread concurrency in a more fundamental way, we must ask ourselves what the role of hwps is in a mixed system that has both lwps and hwps. The real computation is done is done by lwps - hwps simply schedule them, and are essentially virtual processors that are scheduled on the real processors. In Java and 730-xinu, the threads of an application are bound to a single virtual processor. Thus, all of these threads share a single physical processor, as the kernel does not understand threads. The key to solve this problem is to schedule the threads on multiple virtual processors, which can then be bound to muyltiple physical processors by the OS.

This solution is described in Mcann et al. It requires is three kinds of entities. One, *applications* or *jobs*, which like Unixs hwp define address spaces but unlike Unix do not define threads. These are known to the kernel. Second, *virtual processors*, which are known to the kernel, and are executed in the context (address space) of some application. Each application creates a certain number of vps based on the concurrency of the application (that is, the number of threads that can be active simultaneously) and other factors we shall see later. As far as the kernel is considered, these are the units of scheduling. It divides the physical processors among the virtual processors created by the different applications. Third, *threads* or *tasks*, which are lwps supported in user-level code and scheduled by the virtual processors. Like a Xinu kernel, a virtual processor schedules lwps: the difference is that multiple virtual processors share a common pool of lwps. Thus, an application thread is not bound to a virtual processor - All ready application threads are queued in a ready queue serviced by the multiple virtual processors.

Now we have a scheme that provides the benefit we wanted. Fine- grained concurrency is provided in user-level code by a virtual processor, which switches the threads. Large-grained concurrency is provided by kernel-level code, which switches among the virtual processors. The net results is that multiple threads of an application can be executing at the same time (on different virtual processors) and the cost of switching among threads is low!

1.2 Scheduling

So now we have two kinds of scheduling: scheduling of threads and scheduling of virtual processors. Scheduling of threads is analogous to the scheduling on uniprocessor machines in that multiple processes

are assigned to a single (virtual) processor. Scheduling of virtual processors to multiple physical processors is more tricky.

A straightforward generalization of the uni-processor case is to have a single queue of virtual processors that is serviced by all the physical processors. Tucker and Gupta implemented this scheme and tested it using several concurrent applications. They found that the speedup of applications increases with increase in its virtual processors as long as the number of virtual processors is less than the number of physical processors. However, when the number of virtual processors exceeds the number of physical processors, the speedup dramatically decreases with increase in virtual processors!

There are many reasons for this decrease:

Context switch: We now have the cost of switches to kernels, loading of registers, and possibly loading of translation tables.

Cache corruption: When a physical processor is assigned to a virtual processor of another application, it must reload the cache. The cache miss penalty for some of the scalable multiprocessor systems (such as the Encore Ultramax using Motorola 88000) is high and can lead to a ten times performance degradation.

Spinning: When a virtual processor is preempted, the thread it is executing is also preempted and the thread of some other, previously preempted virtual processor is executed. No useful work may be done by the latter thread since it may be spinning waiting for the former to release a lock or signal a semaphore or send a message. As long as the first thread remains unscheduled, all scheduled threads waiting for it will do no useful work.

One solution to the cache problem is to try to execute a virtual processor on the processor on which it last executed, which may still have some of the data of the virtual processor. However, this *affinity-based scheduling* approach reduces the amount of load balancing the system can do. One solution to the spinning problem is to let each virtual processor tell the system if it is executing a critical section or not and to not preempt a critical VP. However, this *smart scheduling* approach allows applications to cheat and hog more than their fair share of processors.

Tucker and Gupta propose a better solution, called the *process control policy* by Mccan et al. The basic idea is to ask (controlled) applications to dynamically reduce the number of ready virtual processors (VPs) when the number of virtual processors created by them exceeds the total number of physical processors. It tries to equally partition the number of processors among the applications, modula the problem that the number of applications may not evenly divide the number of processors and an application may need fewer processors than its quota. The exact algorithm for calculating quotas is as follows. We first assign each application zero processors. We then assign a VP (virtual processor) of each application a processor, and remove an application from consideration if it has no more VPS. We repeat the above step until all processors/applications are exhausted.

When an application is created or terminated in a busy system (that is all physical processors are busy) the system recalculates the quotas of the applications and if necessary asks existing applications exceedings their quotas to preempt existing virtual processors at their next ‘safe points’. A virtual processor reaches a safe point when it finishes a task or puts it back in the queue. The number of virtual processors may exceed the number of physical processors temporarily, since virtual processors wait until safe points.

The implementation of this scheme is done by a scheduling server. The server keeps track of how many ready virtual processors an application should have. The root virtual processor of each application periodically polls the server to inquire how many virtual processors it should have. Each virtual processor, when it reaches a safe point, checks the current number of virtual processors with the quota. If the application has too many virtual processors, the processor suspend itself, it it has too few, it resumes suspended processors. A process suspends itself by waiting on an unused signal, and a processor resumes another process by

sending that signal to that processor.

The process control policy is just one of the possible policies for keeping the number of virtual processors equal to the number of processors. Mccann et all describe several other such policies.

1.3 Equipartition

The *Equipartition* policy is a minor variation of the process control policy. Unlike the latter, the former does not ever let the number of ready VPs exceed the number of processors. Instead of assigning a new application a physical processor and then waiting for an existing VP to relinquish one at a safe point, Equipartition assigns the new application a processor after an existing VP has relinquished a physical processor.

Both the process control and Equipartition policies are quasi-static policies in that they recalculate their quotas only when an application is created/terminated. They work well when the number of VPS an application has is fixed - that is the parallelism of an application does not change dynamically. In some parallel applications, the number may change dynamically. For instance, most parallel applications have an initialization phase in which a single thread forks other threads. This phase could be take substantial time, depending on the application. Other applications may gradually increase and decrease the number of threads as waves of computations come and go. The policies we have seen so far create a static number of VPS for each application (which get suspended/resumed dynamically) which must equal the maximum concurrency the application will ever have. This is wasteful when the application is in a phase that cannot use its maximum concurrency.

1.4 Dynamic

The Dynamic policy tries to solve this problem. Under this policy, when a VP of an application cannot find any application thread, it tells the system that it is willing to yield. Conversely, an application advertises to the system how many additional VPs it could use. When an application asks for additional VPS, the system tries, first, to give it any unallocated physical processors. If none are found, it tries to give it those busy processors whose VPS are willing to yield. If none are found, it tries to enforce equal partition of the processors like the process control and Equipartition policies.

Unlike the Equipartition and process control policies, this policy forces immediate preemption, not waiting for the next convenient point. Of course this has the disadvantage that the VP preempted might have been executing a critical section. Therefore, the system tries to choose a VP of an application that is not executing a critical section. If it cannot find such a VP it simply picks one. Each thread tells the system whether it is currently executing critical or non critical code. Of course a thread can cheat, but it is competing with threads of the same application, so there is no incentive to cheat.

But there is the chance that an application's virtual processor may cheat and not be willing to yield even when it has no application thread to give up. It may be sufficient to assume that everyone uses a library that enforces this policy. Even then, for fairness sake, it may be useful to preferentially treat applications that have been willing to yield, just as it is useful to preferentially treat applications that do not take too much compute time. One approach is to define a credit function, that keeps track of the history of an application's allocation. A job can ask for more than its fair share of processors if it has sufficient credit. We can define the credit at time T for an application as:

$$\frac{\text{Sum } (t = 1 \text{ to } T)\ E(t) - A(t)}{T}$$

where E(t) is the application's fair share (as defined by Equipartition) of the processors at time t.

It may be useful to delay a small amount before advertising that a VP is willing to yield, just in case a new thread is created during that time. This *lazy yielding* scheme can significantly reduce the number of preemptions.

The policy has been implemented by McCann et al on top of the DYNIX operating system. As before, a server process, called the processor allocator, implements the policy outside the kernel. An existing OS (DYNIX) has been modified to allow server processes to dynamically request binding and unbinding of virtual processors to physical processors, which is used by the processor allocator to do the preemptive scheduling. A modified (PRESTO) library is used to implement threads, which communicates with the processor allocator using shared memory. Whenever a VP finds the thread Queue empty, it sets a flag to notify the allocator, and continues to poll the Queue. If the allocator suspends it, then the polling is stopped until the time it is rescheduled. At that time it continues to examine the Queue and resets the flag if the Queue becomes non empty.

Since the allocator is doing the scheduling, it has the responsibility of unbinding a VP that blocks because of an asynchronous event such as I/O that it does not know about. It needs an indirect method to determine if a process is blocked or not. To detect blockings, it puts a null process behind the bound VP in the queue of a physical processor. The null process simply notifies the allocator (using shared memory) that it has been activated, and does a wait. The allocator then unbinds the VP and sends a signal to the blocked VP. When the VP is unblocked, it executes a special signal handler that tells the allocator that it has unblocked and is ready to run. At this point, the system has to find a new processor.

It uses the following algorithm to decide if a suspended VP should run immediately and the processor it should execute on. If the VP was executing critical code and the quota of its application was fully used, then it tries to find a willing to yield VP and preempts that virtual processor. If it cannot find one, it simply preempts some VP of the same application.

If the VP was not executing critical code and the quota was fully used, then it puts its thread in the application Q, so that some other VP can execute it. At this point the application may request additional VPs.

If the quota was not used then it simply lets the VP run, preempting a VP of some other application if necessary using the algorithm we discussed earlier.

1.5 Round Robin

All three policies we have seen so far do 'space scheduling', that is, multiplex the set of processors among the applications. The other alternative is to do 'time scheduling', that is, multiplex the time (on as many processors the applications can use) among the various applications. This is a more direct extension of the single-processor case and is called *RRJob* by McCann et al and *co-scheduling* in Singhal/Shivratari. The basic idea is assign each ready application in turn a fixed quantum. During an application's quantum, it gets as many of the physical processors as it can use and then the application following it in the ready Q is given as many as it can use, and so on. The idea is inspired by the working set principle: give an application as many processors as it needs because if you do not, the scheduled threads many not get any useful work since they need to communicate with some crucial unscheduled threads. It was first implemented in Medusa, which supported the notion of a process team, a team of processes that need to be coscheduled. It was useful in Medusa since it had no processor cache so the cost of switching a processor among threads of different applications was not so high.

1.6 Issues and Evaluation

We have seen above several different schemes for processor scheduling. These can be distinguished by how they handle three orthogonal issues:

Do they support space scheduling or time scheduling?

Do they support dynamic or static partitions, that is, do they keep the number of schedulable virtual processors of an application static or dynamic? A policy is more dynamic than another if it changes these partitions more frequently.

Do they coordinate or not with the applications when they reallocate processors, that is, use information provided by the application (e.g. whether a VP is at a safe point, whether a thread is executing a critical region) when they reallocate?

The following table describes how the four policies we have seen handle these issues:

	Time vs Space	Static vs Dynamic	Uncoord. vs Coord
RRJob	Time	Static	Uncoordinated
Equipartition	Space	Quasi-Static	Coordinated
Process Control	Space	Quasi-Static	Coordinated
Dynamic	Space	Dynamic	Coordinated

Equipartition and Process Control are not distinguished by this taxonomy because it does not distinguish between different coordination schemes. They are Quasi-Static since even though an application's need (number of VPs it creates) is static, its quota (number of unsuspended VPs) is dynamic and changes as applications are created/killed. As a result, the partition of an application changes less frequently under McCann's Dynamic policy.

McCann et al evaluated these policies based on the following criteria:

Response Time

Fairness

Response Time to Short Jobs

In all of their experiements, they used 16 processor machines.

Let us consider each in turn.

1.6.1 Response Time

A technique is superior to another in this dimension if it completes the same set of jobs faster than the latter. The response time of a technique depends on how it handles the three issues. Consider, first, the issue of time vs space scheduling.

The impact on the performance of this factor can be measured by comparing RRJob and Equipartition on static application mixes in which each application's maximum parallelism is equal to the total number of processors.

Since the application mix is static, there is no (coordinated) preemption in Equipartition. Moreover, since the number of VPs of each application is equal to the number of processors, we do not get the RRJob drawback of uncoordinated preemption discussed below. So these mixes isolate the time vs space scheduling factor. Mccann et al find that space sharing can be as much as 25 percent faster than time sharing. There are several reasons for this, some of which we saw earlier: the cost of context switching and cache invalidations. Moreover, the extra threads scheduled by time sharing during a quantum may not be able to much useful work either because they are simply spinning, or they cause extra synchronization contention. Finally, an

application's speedup saturates as the number of concurrent threads increases, so it may not be a good idea to give it all the processors it wants. It is better, as under space sharing, to use some of the processors for other applications.

Now consider coordinated vs uncoordinated. To determine the influence of this factor, McCann et all took a job consisting of interdependent threads that needs 11 maximum processors, and ran 4 copies of it simultaneously. However, when they did the experiments they found that the time taken by the four copies was about 4 times the time taken by a single application. This is surprising, since in a time quantum, two jobs run, one with 11 processors and one with 5. So this implies that the job with partial allocation could not make much use of its 5 processors. The reason must be that when the system suspended some of the VPs, it also suspended the threads that these VPS were running. As a result, other threads could not make much progress. Had it done some coordination, it could have allowed the suspended VPS to release the threads to other VPS and to keep critical threads going.

To calculate the impact of uncoordinated preemption, they also calculated the optimistic bounds on how much time the job would take. Assume that a quantum is of length Q, and that the application takes T(n) time if it is given n virtual processors to execute its threads (with no preemption). Then, the portion of each of the 4 applications completed in 4 time quantums Q, is:

$$f = Q/T(11) + Q/T(5) = Q * (T(11) + T (5)) / T(11) * T(5)$$

So the time required to complete all 4 applications completely is 4Q/f, which gives us:

$$4 * (T(11)*T(5)) / (T(11) + T(5)).$$

This does not take into account the context switching overhead. They measured values for T(11) and T(5) and found the optimistic bound to be about 1/2 the actual time taken. Obviously, this problem would not occur in applications that do not have much interdependencies among the threads.

Finally, consider dynamic vs static. The impact of this factor was isolated by comparing Equipartition with Dynamic. They found that, depending on the application, Dynamic gave between 6 and 13 percent better performance. This can be attributed to the fact that Dynamic keeps fewer processors idle. On the other hand, it makes more frequent reallocations and these allocations are more expensive. As a result, it increases the chances of a VP being assigned to different processors, thereby increasing cache misses. The results show that the benefits far outweigh the drawbacks, this despite the fact that allocations were done in user-space.

To measure the cost of eager vs lazy yielding of processors, they measured performance for different delay factors. While the number of reallocations dramatically decreased when a processor delayed before declaring itself ready to yield, the performance did not dramatically increase. This indicates that the cost of reallocation is actually negligible compared to its benefits.

1.6.2 Fairness

A fair scheme will try to make sure that two applications submitted together take the same time. According to this criteria, time sharing is less fair than space sharing, since an application gets not only processors during its time quantum but also during the time quantum of the application in front of it. Depending on who is in front of it, an application may get more or less cycles than another similar application submitted at the same time. Space sharing tries to be more fair, but it cannot be perfectly fair because the extra processors found in the last iteration are assigned to some and not others. The Dynamic version of it is more fair because it does more frequent reallocations and also because it reduces a process's priority if it does not yield enough processors.

1.6.3 Short Jobs

We might also want short (interactive) jobs to complete quickly. RRJob is not bad in this regard, since it puts the new job at the front of its queue.

Equipartition is the worst, since it politely waits for an existing VP to reach a safe point before assigning a processor to the new job. Process control is better than equipartition since it makes the new application's VPs schedulable immediately. However, these VPs may have to wait for the next time quantum before they can get the processors. Dynamic is the best, since it does not even wait for the beginning of the next time quantum and immediately preempts running VPs and gives the freed processors to the new application.

1.7 Kernel Support for User-Level Threads

The threads executed by virtual processors are much like Xinu processes/threads in that they share a single virtual address space. An important difference is that the Xinu processes are managed by the kernel. Therefore, they are more like the processes implemented in your project, which run in user-space. The user-level threads were meant to be simulations we created of the real kernel-supported threads so we don't have to work on the bare machine. As we see in the dicussion above, they are useful in the presence of kernel supported (heavyweight or lightweight) processes. Switching among kernel supported threads (lwps) requires trapping to the kernel. Switching among hwps, in addition, requires switching of address spaces. Moreover, operations executed by kernel supported hwps and lwps such as resume(), signal() and wait() require trapping to the kernel, and checking and copying of parameters for security reasons. In the case of user-level threads, these costs are not incurred. Anderson et al did experiments in 1991 to quantify these results. They measured the cost of (a) forking a process that executes a null procedure, and (b) executing a signal followed immediately by a wait. In the case of user-level threads, kernel-supported lwps, and hwps, the costs were (34, 37), (948, 441) and (11300, 1840) micro seconds, respectively.

Performance is not the only reason for supporting user-level threads. Different applications can use different types of user-level threads. For example, a real-time application can use threads with priorities, while a matrix multiplier can use threads withour priorities, which we have seen are simpler to implement and more efficient as they don't involve maintaining and looking-up a priority queue. On the other hand, kernel threads are shared by all applications, and thus must satisfy the requirements of all applications. The result is that an application such as the matrix multiplier that has simple needs must pay the cost of features it does not need.

On the other hand, user-level threads raise some conceptual and implementation issues. What happens when a user-level thread performs a blocking I/O call? In your project implementation, such a call blocks the whole Unix process. One "solution" to this problem is to use asynchronous calls instead of synchronous ones. However, as we have seen before, such calls are harder to program. Moreover, if such calls return results, then a mechanism must be provided to get the results. If the kernel does not know about the thread, how does it deliver the results to it? One solution has been to poll for results of asynchronous calls.

Another issue, which we saw above, has to do with what happens when a virtual processor executing a user-level thread has to suspended. This is required in space scheduling when a new application enters the system. Process control and equiparition ensure that a VP cannot be preempted if it is not at a safe point, but we saw this has negative consequences. In process control, the number of vps can exceed the number of processors, while in equipartition, the new application must wait for a processor. Dynamic addresses this problem by preempting a currently running vp, ideally one not in a critical section. However, this means the preempter must know if a process is in the critical section. Moreover, the thread executed by the vp does not get a chance to execute until its vp is reassigned to the application. Ideally, we would like to schedule the

thread on some other virtual processor. But if the kernel does not know about the thread, it cannot do so.

Yet another issue is how a vp is assigned to a processor, preempted, and resumed. In process control, a scheduling server kept track of the scheduling policy, calculating quotas. A vp suspended itself to preempt itself by waiting on a signal. The root vp of an application took the responsibility of resuming the vp by sending the signal. This policy does not work for dynamic, which requires immediate forced preemption.

Addressing all of these problems requires some form of kernel support. We saw an example of such support when we studied dynamic. The kernel allowed a user-level process to bind and unbind vps to processors. This together with a null process and the signalling mechanism we talked about addressed the second and third problem, but not the first one (preemption of a vp executing a thread). It is attractive to find an integrated way to address all of them.

One such way is to provide kernel support for user-level threads. This may seem like a contradiction in terms: how can a thread be user-level if it is supported by the kernel? The idea is that a user-level thread is known to both a user-level manager and the kernel, and its context block is kept in memory shared between the user-level manager and kernel. Some operations are implemented by the manager and some by the kernel. Moreover, the manager informs the kernel of some of the operations it implements. Conversely, the user-level manager is informed about some of the operations implemented by the kernel. This is a variation of the coordination we discussed in space scheduling - the coordination happens between the kernel and user-level manager rather than vps and the global scheduler. It is also a variation of the idea of a global user-level scheduler communicating with the kernel to bind a vp to a physical processor. As long as the kernel operations are executed infrequently, and the communication between the kernel and user-level thread manager happens rarely, we can have the benefits of user-level threads without the drawbacks.

We would, of course, like to have as few operations as possible executed by the kernel. Operations that block on I/O, however, must involve the kernel, as it is the kernel that processes the I/O. If the kernel knows about the user-level thread that executed the call, it can inform the user-level manager that the thread is blocked, allowing it to make use of the processor on which the call was executed for another thread. This is particualry important when the associated application has only one vp allocated, as in our project. However, some coordination mechanism must be used to inform the user-level manager that the vp can be reassigned. Similarly, when the I/O operation completes, some upcall mechanism must be found to inform the manager that the thread that executed it is now ready to be scheduled. Similarly, when a vp bound to a thread is preempted, some mechanism must be provided to ensure that the user-level manager can schedule it on another vp.

So far, we have assumed that virtual processors of an application are simply kernel threads sharing the address space defined by the application. In other words, as far as the kernel is cocncerned, they are simply kernel threads. If the kernel knows that these are virtual processors, it can provide a simple and uniform mechanism to address the three issues.

To understand the nature of this mechanism, let us consider the differences between general kernel threads and vps. There is no need to share a processor among different vps as long as space sharing is used, with one vp assigned to each processor. In other words, as round robin scheduling is provided at the user-thread level, there is no need for it at the kernel level, as long as it is doing space multiplexing. Moreover, the only kernel-level blocking call made by a vp is an I/O call. Synchronization, IPC, and suspend/resume operations apply to the user-level threads and not vps. Furthermore, the I/O call a vp makes is on behalf of a user-level thread. When the call completes, the vp can be destroyed as long as the user-level manager can be informed about the readiness of the user-level thread, and the thread can be assigned to another vp. Finally, a regular (kernel or user-level) thread is explicitly invoked by a user-program to perform some user-task. A vp, on the other hand, is simply a vehicle for executing a user-level thread. As a result, it can be created

automatically by the kernel rather than in response to a request by the application program. This works if the kernel is the one determining how many processors are assigned to an application.

1.8 Activations vs. Threads

These properties have been used to design a system developed by Anderson et al (Scheduler Activations: Effective Kernel Support for User-Level Management of Parallelism), in which vps are called scheduler activations. A scheduler activation is like a kernel thread in that it has both a kernel stack and user-level stack, used in the kernel and user mode, respectively. Moreover, it is associated with a context block and an id, called an activation id or activation number. It is called an activation rather than a thread/process because once it blocks, it needs to resume only to complete the operation for which it blocked. After that, it can be destroyed. The activation is created by the kernel rather than application program. At any point, the number of kernel activations is equal to the number of processors in the system. These activations, of course, are shared among the various applications. Like a Xinu thread, an activation invokes a function asynchronously. As the activation is created by the kernel rather than the application, it upcalls callback procedures at well known locations in user-space. (Invocation of a callback procedure in a client by a system component/server is called an upcall in systems research.) These procedure do not directly execute application tasks. Instead, they execute code that performs user-level thread management. Automatically creating a new process that executes a well known user-defined procedure is a generalization of the idea of automatically creating a Unix process executing the main procedure at a well-known location. Below we see the signatures of the procedures, when they are called, and the tasks they are expected to perform.

1.9 Assigning Processors

When an application is created, at least one virtual processor must be assigned to it. Therefore, the kernel creates an activation/virtual processor for it that executes the upcall, *processorAllocated (processorNumber)*, in the application. This procedure keeps track of whether it has been invoked before or not. As this is its first invokation, it binds the main application (user-level) thread to the activation. This thread can subsequently ask the (user-level) thread manager to create additional threads. In response to such requests, the manager creates the threads, puts them in the ready queue, and invokes the *addMoreProcessors (numberOfProcessors)* (down) call in the kernel to request new processors to execute the ready threads.

The kernel notes the processor need of the application, and based on its Equipartition quota, assigns some number of processors (less than or equal to numberOfProcessors) to the application. For each of these processors, it invokes the *addThisProcessor(processorNumber)* upcall in the application, which can then execute the next thread in the ready queue. The processor number is passed to the upcall so that the thread manager can use it to reschedule a thread executing on that processor, as we see later. The processor number may also be needed to get the activation id bound to a processor.

So now we have some number of threads executing on processors and some waiting the ready queue. The thread manager can then use a kernel-supported timer to context switch among the threads using some scheuduling policy of its choice. Each time an activation reschedules to a new thread, it can set a timer, binding it to a timer handler. As in your project, the handler can reschedule when it is called. We can assume that the cost of setting and calling the timer handler is small compared to a regular kernel call. Moreover, threads can be expected to block on IPC and synchronization calls before they use up their entire time quantum.

When a thread executes a synchronization, ipc, or other kind of blocking call supported by the thread manager, the manager, without any kernel involvement, can block the executing thread, and assign the

associated activation to some other ready thread. If such a thread does not exist, it can declare to the kernel that it is willing to yield the associated processor by invoking the *thisProcessorIsIdle()* downcall in the kernel. The processor number need not be passed to the kernel as the kernel keeps track of the processor on which an activation was created. The kernel now assigns the processor to some application's whose processor need has not been met and has enough credit (as defined by the dynamic or some other policy), and invokes the *processorAllocated(processorNumber)* in the application as before.

1.10 Preempting Processors

The kernel might have to take back the processor of some application in order to reassign the processor to some other application. The processorAllocated() upcall is invoked in the latter application to inform it about the new processor. The kernel does not check with the first application to check if it is safe to reassigns its processor. As in dynamic, it simply preempts the processor. To allow the thread executed by the original application to be rescheduled, it makes the upcall *processorHasBeenPreempted (preemptedActivationNumber, machineState)*. The thread manager now knows that the processor attached to the thread is no longer available, and puts the thread in the ready queue after storing its register state. This upcall need to be made by some activation, which needs a processor. If the application has a free processor, the kernel uses it. Otherwise, it preempts another processor of the application to inform it about the first preemption. Thus two processors (and associated threads) of an application can be preempted when a processor is taken away from an application! If the kernel was to make another upcall to inform the application about the second preemption, we would end up with an infinite sequence of preemption events and very soon would run out of processors. Therefore, the kernel invokes the *processorHasBeenPreempted()* call only when it preempts a processor of an application to give it to another application. Thus, in the above scenario, the second preemption does not result in an special upcall as the processor is still available to the application (to schedule the same thread it was originally executing or some other thread). The upcall made for the first preemption must be able to determine the processor that was preempted to execute it. We can imagine a downcall, *getProcessorNumber()*, that can be invoked by an activation to get this information, though the paper does not explicitly mention it. (Or the machine state could possibly provide this information.) Once the upcall knows the processor number, it can determine if this was an idle processor or one that was executing a thread. In the latter case, it can (use its mapping of processor number to thread to) determine the thread, and place it on the ready queue (along with the first thread that was preempted, whose identity can be determined from preemptedActivationNumber).

When a processor is taken away from an application, the application might have no other processor to use for the processorHasBeenPreempted() upcall. In that case, the upcall is delayed until the application is assigned a new processor.

When the kernel preempts an activation of an application, it does not know the priority of the thread bound to the activation, which is maintained by the thread manager. It is possible that the thread has a higher prority than one currently executing on a processor assigned to the application. To maintain priority-based scheduling, the thread manager can invoke the downcall *interrupt(processNumber)* to ask the kernel to preempt the thread and activation. Once the thread and its activation have been preempted, the kernel first invokes the upcall *processorHasBeenPreempted(preemptedActivationNumber, machineState)* to let the thread manager put the thread in the ready queue. Next, it creates a new activation on the preempted processor, and invokes the upcall *addThisProcessor(processorNumber)*, which can schedule a higher priority thread.

1.11 I/O Calls

Let us consider now an I/O blocking call made by a thread. Such a call is processed not by the thread manager but the kernel. When it is made by a thread, the kernel unbinds the associated activation from the processor on which it was executing. To reassign this processor to the application, it creates a new scheduler activation and binds it to the processor. At this point the application needs to know that: (a) The thread has blocked on an I/O call and thus is not eligible to run. (b) A new activation is available on the processor on which the thread was running. The kernel does not directly know the thread that was blocked, because a thread is a user-level abstraction. On the other hand, it knows the activation the thread was running. Let us assume that the thread manager can determine the association between activations and processors by invoking the call, getActivationNumber(processorNumber). (The paper by Anderson et al does not directly mention this call.) It knows the association between processor numbers and threads. Therefore, the kernel provides the thread manager with the two pieces of information by making the new activation (bound to the processor) invoke the following upcall: *activationHasBeenBlocked (blockedActivationNumber)*. The thread manager can now change the state of the blocked thread to blockedOnI/O and schedule some other thread on the activation.

When the I/O operation of the blocked thread completes, the thread may need to perform further steps of the operation in kernel mode. After that, the thread needs to resume execution in the user mode. The additional steps are executed by the original or a new activation assigned to a new processor (the paper is silent on this issue), and this or another activation (the paper is silent on this issue) can now execute the upcall *activationHasUnblocked(unblockedActivationNumber, machineState)* in user-code. From the unblockedActivationNumber, the thread manager can determine the thread that was unblocked. Next, it can store the register state in the context block of the thread, and put the thread in the ready queue. It can then use the new activation to schedule the next thread. As mentioned above, this activation executes on a new processor. If there are no idle processors, the kernel preempts some processor (of the same or different application, depending on the application's quota and allocation). If it preempts the processor of some other application, then it invoked the upcall processorHasBeenPreempted (preemptedActivationNumber, machineState) as we saw before. If it preempts the processor the same application, then it does not invoke this call, as we also saw before. The upcall *activationHasUnblocked()* can use the machine state/getProcessorNumber() in the manner described before to determine if a new processor was assigned to it or an old processor was preempted, and in the latter case, to put any thread bound to the processor in the ready queue.

1.12 Page Faults

Another issue has to do with page faults. If a thread bound to an activation page faults, it is useful to assign its processor to some other thread so while the page fault is being serviced. A page fault can be considered an implicit I/O blocking call, and handled like an explicit one. Whenever an activation page faults, the kernel can unbind it from its processor, and create a new activation on the processor that invokes the activationHasBeenBlocked() upcall, indicating the activation that page faulted. The manager can then sechedule a new thread on this activation. Page fault completion similarly is like completion of an I/O call.

One difference between a page fault and an I/O call is that it can occur in an upcall when the thread manager is in the middle of scheduling a thread. Similarly, a preemption can occur in the middle of an upcall, when an activation is unbound to a user-level thread. In these two cases, it is the thread manager's state that is saved by the kernel. The upcall executed next to inform the (reentrant) thread manager about the blocked activation must recognize that no user-level thread was bound to the blocked activation and can context switch back to the thread-manager activity.

Under this scheme, an upcall that page faults on a particular location can result in an upcall (informing the thread manager of the event) that also page faults on the same location. To prevent an infinite sequence of such upcalls, the kernel detects this situation and delays invoking the next upcall (notifying the thread manager of the second page fault) until the page fault has been serviced completely. During this period, the processor could be reassigned to some other application (the paper is silent on this.)

1.13 Critical Sections

A related issue arises when we consider preemption of threads in critical sections. If the thread holds a lock on an application data structure, this is not an issue, as the notifying upcall can put the thread back on the ready list to be later rescheduled on a different processor. However, if the thread holds a lock on a thread-manager data structure, in particular the ready list, then it cannot be placed in the locked (and potentially inconsistent) ready list. Therefore the notifying upcall must check if the thread data structures were locked by the preempted upcall and temporarily resume its execution until the data structures are unlocked.

How does the thread manager know that a preempted thread was in a critical section? The thread can set/reset a flag when it enters/leaves a crtical section, but the paper reports that this can have significant impact on performance. The problem with this approach is that this overhead is incurred even when threads in crtical sections are not preempted. The paper provides an interesting solution that follows the principle of making the normal case efficient and the abnormal case (preemption of threads in critical sections) possible but not necessarily efficient. It assumes that each crticial section is delimited by special assembler labels. For each section, a copy of the critical section that has additional postprocessing code that handles this special case. Normally, the original copy is executed. The activation created to notify the user-level thread manager about a preemption checks if the preempted thread was executing a critical section. If so, it jumps to the corresponding location in the copy before making the upcall. The postprocessing code in the critical section simulates an invocation to the upcall rather than continuing executiuon of the thread beyond the critical section. (This is my understanding of the confusing description in the paper.)

1.14 Performance

Another potential source of inefficiency is creation of an activation on each upcall. To reduce this cost, activations can be put in an activation pool much in the way threads are put in a thread pool. A pool of activations (threads) is simply a bounded buffer of activations (threads). Deletion/garbage collection of a thread (activation) does not deallocate the resources allocated for it. Instead it puts the item in the buffer. When a new activation (thread) is needed, the bounded buffer is checked to see if a free item exists in the buffer. If so, the item is reused, otherwise a new item is created. When the kernel starts, it can put an initial set of activations (threads) in the buffer.

Even if an activation is not expensive to create, each upcall it makes results in a switch from the kernel to user mode. However, we can argue that the number of mode switches made is the minimum neded to execute the application threads and efficiently multiplex them onto the processors. When a thread makes a blocking I/O call, a switch must be made from user space to kernel space to process the call. If the processor of the blocked thread is to be reused for another thread, a reverse switch to the user space must be made. Similarly, when the I/O call completes, a switch to the kerneal is needed to process the associated interrupt, and another switch is needed to resume the calling thread. One potential cause of inefficiency is that an upcall executes thread-management code before switching to application code. Again, this is necessary so that the appropriate thread can be executed.

Anderson et al created an implementation of these ideas and evaluated its performance. They found that the costs of the null fork and signal-wait calls was 37 and 42 microsoeconds, respectively, slightly more (and significantly less) than the cost of these calls in pure user-level (kernel) threads. Interestingly, without the optimization that copies critical sectons, the costs would have been 49 and 48 microseconds, respectively.

These numbers to not take into account other thread-manager calls and kernel calls, the frequency of calls, and the costs of preemption. These factors are needed to determine the impact of scheduler activations on an actual application. To evaluate the impact scheduler activations on application performance, we can consider two cases: when an application makes few I/O calls and when it makes a significant number of I/O calls. In the first case, threads executed by activations were found to be perform as well as pure user-level threads because no additional mode switches were made. In the second case, they were found to perform better than user-level threads because they allowed the processor of a blocked thread to be reused for another thread. In both cases, they performed better than kernel threads, which required mode switches even for non I/O calls.