COMP 242 Class Notes
Section 9: Multiprocessor Operating Systems

1. MULTIPROCESSORS

Reading: Chapter 17, Singhal (sections 17.1 tp 17.4, 17.6).

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. Therefore, the solution seems to be support lwps in user code within a hwp.

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,

So we need a more elaborate scheme to give us fine-grained concurrency at low cost.

The solution, described in Mcann et al, is to support 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 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 the data of the 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 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 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. One way to define the credit at time T for an application is:

$$\text{Sum } (t = 1 \text{ to } T)\ N - A(t) \quad \text{———————} \quad T$$

where A(t) is the number of processors allocated at time to the job and N is the total number of processors in the system. Unfortunately, this approach does not take into account the importance of yielding processors during high contention 'rush hours'. So the policy actually supported in Dynamic is:

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

where E(t) is the application's fair share of the processors at time t. Thus, it gives extra credit to applications that yield processors during high contention.

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 Q empty, it sets a flag to notify the allocator, and continues to poll the Q. If the allocator suspends it, then the polling is stopped until the time it is rescheduled. At that time it continues to examine the Q and resets the flag if the Q 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 another 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

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 25faster 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 than 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 ap-

plication, Dynamic gave between 6 and 13 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.