CMPSCI 677 Operating Systems **Spring 2013**

Lecture 4: February 4

Lecturer: Prashant Shenoy *Scribe: Vani Gupta*

4.1 Threads

4.1.1 User-level Threads

User level thread packages are those that implement the entire threading functionality in user space. The kernel is not aware of these threads, and the OS kernel does not explicitly support threads. The entire abstraction is implemented as a library in user space, and applications link to that library and make use of thread management. A multi-threaded process consisting of user level threads looks to the OS like a traditional single-threaded process. The kernel schedules the process in kernel space, and the library scheduler schedules the thread in user space. So, this is a two-level scheduling decision one in the kernel space where the process is scheduled, and one in the user space where a thread in that process is scheduled.

Advantages:

- There is flexibility in choosing a thread package implementation based on the needs of the applications. The application designer is not restricted to the thread package implementation provided by the kernel. Different processes can choose different thread package implementations that can coexist in the system.
- The CPU scheduler of the kernel is not the only choice. Application designers can pick a library scheduler by picking an appropriate thread package implementation, or even write their own.
- User-level threads allow for a more lightweight solution. All thread management (like creation, deletion of threads, switching between threads) is done in user space as library calls, not system calls, which are more heavy-weight.

Disadvantages:

- CPU share is not equal for all threads: Threads in processes that have a large number of threads can get a smaller CPU share as compared to threads in processes with fewer threads.
- If a user level thread starts doing I/O, the entire process blocks, even though there may be other threads in the process ready to run.
- Threads from a single process cannot be scheduled in parallel on multiple processors. This is because for user-level threads, the kernel scheduleable entity is a process, and each process is scheduled on one core.

4.1.2 Kernel-level Threads

In kernel-level thread packages, the kernel supports all thread functionality, and implements threading support for applications. The kernel deals with thread management like creation and deletion of threads via

system calls. The kernel scheduler is aware of threads and schedules threads, instead of processes. Advantages and disadvantages are the exact reverse of the ones listed above for user-level threads.

Advantages:

- The kernel scheduleable entities are threads and so each thread can get its fair share of CPU time.
- If a user level thread starts doing I/O, the other threads in the process do not block.
- Threads from a process can be schedule to different cores. So a single process can take advantage of the presence of multiple cores, and one can get thread-level parallelism.

Disadvantages:

- The application designer does not have flexibility in choosing how thread management and scheduling is implemented
- This solution is more expensive and thread management involves system calls which are more heavy-weight.

Note that in user-level threads there is a many to one mapping between threads and entities that are scheduled by the kernel. In contrast, in kernel-level threads there is a one-to-one mapping between thread and kernel scheduleable entities.

4.1.3 Light-weight Processes (LWPs)

Lightweight processes are more generalized, and enable the application designer to pick the mapping of threads in a process to scheduleable entities in the kernel space based on the needs of the application. The application designer is free to use a one-to-one, many-to-one, or a many-to-many mapping. If this mapping is one-to-one, i.e. there is one thread for each light-weight process, then this resembles what kernel-level threads give you by default. When all the threads in a process are mapped to a single kernel scheduleable entity (many-to-one mapping), then this resembles what user-level threads provide by default. However, under LWPs you can choose to have 'n' threads mapped to 'k' light-weight processes. The decision could be based on, for example, what type of work a thread is doing - if a thread is doing important work, the application designer can give it its own light-weight process. The scheduling decision is again a two-level scheduling decision. The kernel schedules the light-weight processes, and the library scheduler schedules the threads mapped to that light-weight process (if there are more than one). The notion of LWPs was first implemented in Solaris. In Solaris, if the application designer did not want to do the mapping, the kernel could dynamically do the mapping by observing thread execution.

4.1.4 Thread Packages

- Posix Threads (pthreads): Widely used in C/C++ applications, and defines only an interface that needs to be implemented. This can be implemented as user-level, kernel-level, or via LWPs.
- Java Threads: Threading support is built-into the language and the JVM schedules threads. This is a two-level schedule decision, the kernel schedules the JVM and the JVM schedules threads. If thread support is implemented as kernel-level threads in the JVM, then they may be visible all the way to the kernel.

Example: An example of processes running on a mac and the threads associated with them is shown. The threads listed are kernel-level threads, as the kernel is aware of them.

4.2 Multiprocessor Scheduling

A hardware schematic of a multiprocessor or a multicore system is shown. There schematic consists of cores (or CPUs) and caches (L1, L2, L3). Caches consist of data/instructions fetched from memory, and are used to speed up execution. Some caches (L1) are not shared with other cores, where other caches (L2, L3) are shared.

In the multi-core/multi-processor setting, each core/processor makes an independent scheduling decision. The kernel runs on a processor, and the kernel scheduler picks a process for execution. Once a process starts to run, it either runs for the entire timeslice, or does I/O and gives up its timeslice. Once this happens, the kernel runs again and picks the next process, and so on. This happens in parallel on all the processors.

Two different ways of implementing a multiprocessor scheduler:

- Central queue: In this case, there is a single, global queue. Whenever a scheduler runs, it picks the next task (i.e. thread or process) from the head of the global queue. Given this setup, the global queue is a shared data structure, which must be locked by a scheduler before it picks a task from it. Once the queue is locked, other cores must wait for it to be unlocked before they can pick the next task. As the number of cores grows, the queue becomes a bottleneck. To overcome this problem, the centralized queue must be distributed to form multiple queues.
- Distributed queue: In this case, we have more than one queue for a set of processors. In the degenerate case, we have one queue per processor. With multiple queues, the central queue bottleneck issue can be resolved. However, we have to now figure out how to assign tasks to queues so the load is balanced across processors. If a queue has more CPU bound tasks, its corresponding processor will be more heavily loaded, versus one which has a lot of I/O bound tasks. To solve this problem, we need to periodically redistribute tasks so the queues are load-balanced.

Load-balancing queues alone is not enough to improve performance. *Cache affinity* needs to be considered as well - *cache affinity* means that if a task was previously run on a core, it has an affinity to that core. If a task is assigned to a core, reassigning that task to the same core will enable the use of existing task related data/instructions that are already in the core's cache from the previous processing of that task. So, when multiprocessor schedulers reassign tasks to a set of distributed queues, they keep cache affinity in mind and rerun tasks on cores they ran on before. Central queue does not respect cache affinity as there is no control over which task is picked next - it is just the next task in the queue. However, with a single queue per core, cache affinity can be respected by making the task go to the end of its current queue. If loadbalancing is done, it may not be possible to respect cache affinity every time. While reassigning tasks to queues, there is a trade-off between load-balancing queues and respecting cache affinity, and every OS makes its own choice.

In multiprocessor settings, where tasks can get reassigned and have to start with cold caches, timeslices are somewhat longer to allow the caches to warm up.

Question asked in class: Even with single queues per core, given caches are shared, as the number of tasks grows, can the data of older tasks not get ejected from the cache? Answer: Yes, as the cache is shared between tasks, even if you have a single queue per core, some of the data from an earlier task may get ejected. This is why you have multiple levels of caching (L1, L2, L3) - so if the data was ejected from the L1 cache, it may still exist in the L2, L3 cache, which are typically larger.

4.2.1 Parallel Applications

These are massively threaded applications, where threads are running in parallel to achieve some tasks. For such specialized applications, we need specialized schedulers. So far the schedulers we have seen try to

schedule tasks on different processors while trying to be fair across all threads. These specialized schedulers schedule all the threads from an application on all cores, all at once - this is called co-scheduling or gang scheduling. These schedulers are used in special purpose machines that run massively parallel applications. As these threads are scheduled in a coordinated way, if a thread wants to communicate with another thread of the application, it does not have to wait for it to be scheduled at a later time. Interesting design decisions need to be made - e.g. if one thread blocks, are all threads blocked?

4.3 Distributed Scheduling

We now move to scenarios where there are 'n' machines in the system. A distributed scheduler assigns tasks to machines in a system of 'n' machines.

4.3.1 Motivation and Implications

If the system is lightly loaded, and a task comes in, then with high probability it will be assigned to a machine that is idle. In such a case, a distributed scheduler is not really needed to move the job elsewhere. On the other extreme, if the system as a whole is heavily loaded, then a task could come in and with high probability it would get assigned to a machine that is loaded. In this case too, there is no need to do distributed scheduling as there is no other better machine to send the job to. Distributed scheduling is beneficial when the system as a whole is moderately loaded. In this case, there is potential for performance improvement via load distribution.

4.3.2 Design Issues

- Measure of load: When should the scheduler be invoked? We need to be able to measure the load on systems so we can decide whether to move a job from one system to another or not. Some metrics are queue lengths at CPU and CPU utilization.
- Types of policies: We need to define policies based on which the scheduler will decide to move load around. These policies could be static policies that are hardwired into the system. Policies can also be dynamic. Dynamic policies are based on querying the systems for their loads and deciding what to do based on loads. We can also have adaptive policies. Adaptive policies are those in which the policies or algorithms themselves change based on different loads and situations.
- Preemptive versus non-preemptive: Preemptive distributed scheduling means that you can move a currently executing process to another machine. Non-preemptive distributed scheduling means that scheduler cannot move a task that is executing. Non-preemptive schedulers are easier to implement. Preemptive schedulers are more flexible, but are more complicated to implement as they involve process migration.
- Centralized versus decentralized: In centralized policies there is a coordinator that keeps track of load on machines and decides where to move jobs. In a decentralized world each machine takes its own decision, there is no central coordinator and local decisions are made.
- Stability: The design needs to ensure that when you send a job to another machine, that machine does not get overloaded. Suppose a machine becomes idle and all other machines in the system send jobs to it. The machine then becomes overloaded, tries to send jobs elsewhere, and the load keeps oscillating. If policies are not in place to deal with such situations, the system becomes unstable, no real progress is made, and resources are wasted in sending jobs from one machine to another.

4.3.3 Components

A policy can be decomposed into four different components:

- Transfer Policy: The transfer policy deals with questions like: when should distributed scheduling be invoked? or when should a process be transferred?
- Selection Policy: Which process to send?
- Location Policy: Where to transfer the process?
- Information Policy: What information needs to be tracked in the system in order to make the above decisions?

Question asked in class: Is unstability likely to occur in centralized or decentralized settings? Answer: Unstability is more likely to occur in the decentralized case, as each machines is making local decisions without coordinating with other machines. In centralized policies it is easier to avoid instability as you have a global view of the system, however, there are no guarantees that instability can always be avoided when using centralized policies.

4.3.4 Sender-initiated Policy

The sending machine does all the distributed scheduling.

- Transfer Policy: A very simple threshold based policy is used. If the load on the system is above a certain threshold, it sends the tasks somewhere else.
- Selection Policy: Here a non-preemptive scheduling policy is assumed. Only newly arrived tasks can be shifted, processes that are executing cannot.
- Location Policy: A machine can be picked at random. Machines can be polled sequentially or in parallel, and jobs can be sent to the least loaded machine. If multiple machines are searching for machines to send jobs to, many jobs can start arriving at one system, leading to unstability. To avoid unstability, the top k least loaded systems can first be chosen, and then one of those can be randomly picked to send the job to.

4.3.5 Receiver-initiated Policy

The receiving machine makes decisions.

- Transfer Policy: If a system's load falls below a threshold, the system becomes a receiver, and looks for jobs.
- Selection Policy: The system can either accept newly arrived processes, or if it supports process migration, it can accept partially executed blocks as well. (Sender-initiated policies can also decide to send partially executed tasks if they support process migration)
- Location Policy: The system can poll machines and look for the most heavily loaded machine, and relieve its load by asking for jobs. Sequential or parallel polling can be done.

4.3.6 Symmetric Policies

Both sender-initiated and receiver-initiated techniques co-exist in this system. Depending upon your load, you may be a sender waiting to off-load jobs, or a receiver looking for jobs.

- There are two thresholds - a low threshold and a high threshold. If any machine's load goes below the low threshold, it becomes a receiver, and similarly, any machine whose load goes above the high threshold becomes a sender. Machines whose load falls in between the two thresholds are neither senders nor receivers. Such machines just continue to work on the jobs they currently have.
- When you are a sender the sender-initiated policies will apply. If you become a receiver, the receiver-initiated policies will apply.
- In a very large system, having say 10,000 nodes, when senders and receivers actively trying to do send and receive tasks, the chances of finding a match increase. Otherwise, a sender may have to poll several machines before it can find a receiver.