CMPSCI 677 Operating Systems **Spring 2014**

Lecture 3: January 29

Lecturer: Prashant Shenoy *Scribe: Sneha Shankar Narayan*

3.1 Review of Processes

Multiprogramming, or *multitasking* is when a system has multiple processes and it switches between them (based on a scheduler). This can be done on a uniprocessor and without any parallelism. Multiprocessing, however, is still having multiple processes (still multiprogramming environment) but being able to run more than one of them at a time. This requires more than one CPU (or core). There will be concurrency in this case.

The operating system kernel maintains a data structure for processes called the process control block (PCB). Each process has an address space with a code (or text) segment, heap, and stack. A process also has a state which varies during its lifetime. A process begins in the *new* state and moves to the *ready* state when it is ready to run. At some point the scheduler will select the process to run and this changes the process' state from ready to *run*. Each process is run for a time slice (or quantum) unless it finishes (moves to the *finish* state) or starts some I/O operation. An I/O operation will move the process' state from run to *wait*. When the I/O operation is finished, the process will then be changed from wait back to the ready state so that it can be scheduled to run again. If a process' time slice expires without the process finishing or starting an I/O operation, then the process moves back to the ready state.

3.1.1 Process Scheduling

Which process is selected to run from the potentially many processes that are in the ready state is determined by the scheduler (as previously mentioned). The CPU scheduling policy can use a variety of methods:

- **FIFO:** First-in First-out takes processes and runs them, in order of arrival, to completion. In this policy there are no time slices.
- **Round Robin:** This takes FIFO and adds a time slice so that jobs are not just run from start to finish without interruption. When a process' time slice expires it has to move to the back of the list so that other processes that are ready can run for a while.
- **SJF:** Shortest Job First selects the process that is ready and will run for the smallest amount of time. This is a provably optimal greedy solution for minimizing the average wait time. The problem is that it is impossible to implement because it involves predicting the future i.e the length of the jobs are not known. It also can starve (not run) jobs that will take a long time to run if smaller jobs keep being generated.
- **Lottery:** This is a randomized scheduling algorithm where each job is given some number of lottery tickets and the scheduler runs a lottery to see which job "wins" and gets to run next. The number of tickets given to each process can be controlled to alter the probabilities (or priority) of each job.
- **EDF:** Earliest Deadline First just picks the job which needs to finish the earliest. This is another greedy solution which is mostly used for embedded or real-time systems. One issue is that a process might

have an impossible deadline (i.e. needing to finish in 30 seconds but it will take 45 seconds to run). This is used in mission critical applications like Air Traffic control. In general purpose applications one can think of this being used in Media players where a certain number have of frames have to be displayed in a certain amount of time.

Choice of choosing a scheduling algorithm depends on various performance metrics like throughput, CPU utilization, response time etc. The choice depends on what has to be maximized and there is no one particular scheduling algorithm that is good for all situations, it's always a trade-off.

The last process scheduling algorithm that we will discuss requires that we first mention the two types of process behavior. The first type is CPU bound which means that the process is doing computation (using the CPU). The other type is I/O bound, which means that the process is doing I/O operations. Of course a process can switch back and forth between the two types of behavior during its lifetime. CPU bursts are typically short but can more rarely be long.

Either priority queues or multi-level feedback queues (MLFQ) can be used for scheduling here.

- **Priority queues:** Higher priority processes get to run first. For example, real time tasks are executed before non real-time tasks.
- **Multi-level feedback queues (MLFQ):** MLFQ involve some number of queues. Each queue has a priority level and processes can move from one queue to another depending upon if it is CPU bound or I/O bound. The scheduler always picks a process from the highest priority queue that has a job in it (using round robin to pick a job within a queue). I/O bound tasks are inherently smaller. So we can give higher priority to I/O bound tasks and can thus approximate to SJF scheduling algorithm. To find out if a job is I/O bound or CPU bound we can observe what the job does for a short span of time and then label it as I/O bound or CPU bound. If the job runs to the end of its time slice without doing I/O, then it is considered CPU bound and it is moved to a queue on level lower in priority. Otherwise, if the job instead did I/O, then it is moved to a queue on level higher in priority because it is I/O bound. Lower priority queues get longer time slices (and the time slices increase exponentially).

3.2 Threads

A process traditionally is a single execution sequence with a large (possibly sparse) address space. The address space of a process might be shared with other processes if they are using shared memory for interprocess communication. A process also may have some system resources associated with it (e.g. files, etc). This view of a process can also be thought of as a process with only a single thread.

A thread is a light-weight process with a stream of execution through an address space. The difference is that there may be multiple threads allowing concurrent streams of execution through an address space. Typically there is one controlling thread that manages the others. Every thread can access the entire address space but each has its own stack (program counter and registers also). The code and heap segments of the address space are shared. This shared data means that data structures for synchronization (such as locks) must be used to prevent errors.

Threads are beneficial because they allow for a greater degree of parallelism on large multiprocessor/multi-core systems. Without threads, parallelism is limited to process-level parallelism. This means that each process gets its own CPU/core but that is the smallest division possible. With threads a single process can utilize multiple processors or cores. Switching between (and creation/deletion of) processes is also more expensive than switching between threads. Threads basically just have to save the registers that they are

using which means that the context switch is more efficient than for processes. Threads also allow for easier sharing because they share the same address space.

A traditional single threaded process will block for a system call and there is clearly no parallelism. It is possible to use an event-based finite-state machine to allow for non-blocking with parallelism. This type of system would use interrupts and an event handler. A multi-threaded process is much simpler in that it still allows normal blocking system calls but with parallelism. This works by just blocking on the thread that needs to wait and the rest of the threads can continue.

It can be fairly straightforward for software engineering an application as a bunch of threads. This is because each thread will perform some generally independent task.

3.2.1 Examples

An example for a multi-threaded client can be a web browser. There are different threads to go fetch different parts of the page namely the HTML content, images concurrently. This will reduce the latency of page load.

An example for a multi-threaded server is the Web Server which has a pre-spawned pool of threads than can be used to service various incoming requests. There is a dispatcher thread that dispatches the task to one of the threads in the pool which goes ahead and services the request. Yet another way to design a server is to spawn a new thread for every new thread that comes in. But this has the extra overhead of creation and deletion of threads. Apache uses both models. Choice of server design depends on the scale and processing power available.

3.3 Thread management

3.3.1 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 schedule-able entity is a process, and each process is scheduled on one core.

3.3.2 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 schedule-able 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 schedule-able entities.

3.3.3 Light-weight Processes (LWPs)

Lightweight processes are more generalized, and enable the application designer to pick the mapping of threads in a process to schedule-able 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.