CMPSCI 677 Operating Systems **Spring 2019**

Lecture 3: January 30

Professor: Prashant Shenoy *Scribe: Josh Sennett*

3.1 Overview

This lecture covered the following topics.

Module 1: Processes Review

Module 2: Introduction to Threads

Module 3: User-Level versus Kernel-Level Threads

3.2 Lecture Notes

3.2.1 Module 1: Processes Review

3.2.1.1 Review of Processes

Multiprogramming refers to multiple processes running concurrently in a single processor. Operating Systems virtualize the CPU, providing concurrent access to a single processor via **time-sharing** which creates the illusion of parallelism. **Multiprocessing**, on the other hand, refers to using multiple processors on a single machine to achieve true parallelism.

The **Process Control Block (PCB)** is a kernel data structure that keeps track of current processes. It has data about active processes, such as how much memory is allocated to each and where memory pages are stored in RAM for each process. Each process has its own address space with its corresponding code, global and local variables, stack, and heap.

Process State Transitions refer to the states a process may be in at any given time, including:

- New (newly created)
- Running (currently running on the CPU)
- Waiting (waiting for something, such as I/O, an event, or a lock)
- Ready (waiting for its turn to run on the CPU)
- Finished (terminated, to be cleaned up)

3.2.2 Uniprocessor Scheduling

Scheduling refers to the policy of the operating system in choosing which process to run next on the CPU among all processes that are ready to execute. Different scheduling policies optimize different **performance metrics**, and no single scheduling policy can optimize all metrics together.

Common uniprocessor scheduling policies include:

- Round-robin: Each process gets a quantum of time in CPU. Processes are executed in a circular order and without any priority.
- Shortest Job First (SJF): an optimal policy in reducing the average waiting time among processes; however, it has the limitation that the length of the job needs to be known beforehand.
- First-In First-Out (FIFO): "non-preemptive" scheduling; simple, but it can lead to longer average wait times due to short processes being stuck in queue behind long processes.
- Lottery Scheduling: This involves random selection of which processes runs next. Users can control the proportion of time each process gets by deciding on the number of lottery tickets allocated to each process.
- Earliest Deadline First (EDF): an application decides the deadline for a CPU task. EDF is useful in real-time systems where time guarantee is important (for example, in video streaming).

3.2.3 Performance Metrics

Typical performance metrics include:

- Throughput of jobs
- CPU utilization
- Turnaround time (completion rate of jobs)
- Response time
- Fairness (all jobs treated equally)

The scheduler needs to be carefully selected based on the desired performance metrics. Improving one metric may make others worse. For example, one might want to emphasize response time in for an interactive application, while throughput needs be prioritized over response time in cluster computing.

3.2.3.1 Process Behavior

Processes may be CPU-bound, I/O-bound, or most often, alternate between being CPU-bound and I/O-bound. **CPU bursts** refer to the time of CPU usage between two periods of I/O. CPU bursts can be modeled using a hyperexponential behavior, which is useful when deciding an optimal schedule.

3.2.3.2 Process Scheduling

In a simple **priority queue**, each job is assigned a priority level. The kernel first looks to the top-most priority queue that is not empty, and executes processes in the queue until the queue is empty. If the top-most priority queue is empty, the scheduler will look at a lower level, and so on. The highest priority task keeps running until it is complete or it goes to do I/O. Kernel tasks usually have higher priority than the tasks that run in the user space. Priorities can also be assigned among the user applications. For example, Skype gives higher priority to audio processing.

In **multi-level feedback queues (MLFQ)**, the OS uses a priority queue with round-robin scheduling within queues and priority scheduling across queues. However, a process's priority is determined dynamically. When a new process starts, it is put at the highest priority level; if it uses an entire quantum of its CPU time, its priority level is lowered. If it does not use its whole time (if it switches to I/O), it is promoted to a higher priority level.

This scheduler tries to achieve the performance of the shortest job first scheduler. I/O bound processes get higher priority because they spend relatively short duration of time hogging the CPU. New processes always get the highest priority. If a process spends the entire time quanta assigned to it, its priority level drops by 1 and the process moves to the lower-priority queue. If a process does not use entire time slot of CPU, priority level is increased by 1 and the process moves to the higher-priority queue. This results in I/O bound jobs always moving to the highest-priority queue. This scheduler is implementing shortest job first without a prior knowledge of the job length. The assumption is that the I/O bound jobs are shortest because I/O operations are not executed on CPU. Priorities for a job changes every quantum depending on whether the job spent its last time quantum doing computation or I/O.

3.2.3.3 Process vs. Threads

Traditional processes have one stream of execution and are also called single-threaded. But, a process can also have multiple streams of execution if it is multi-threaded. A **thread** is is a flow of control through an address space; each thread gets its own registers, including its own **program counter (PC)**. Two threads of a single process share a heap and code, but not necessarily the stack.

3.2.3.4 Why use threads?

Multithreading allows for concurrency within a single process. In a a uniprocessor machine, threads achieve concurrency through time-sharing of the CPU. Even on a single processor, this can cause a process to improve in performance, because while one thread is doing I/O, another thread can execute on the CPU. On a multiprocessor machine, threads can be run truly in parallel by running simultaneously on different cores and can therefore improve performance. Today, most systems (such as smartphones and laptops) have multiple cores.

Threads allow faster execution of processes by being scheduled on additional cores independently. Creating and switching to a thread is more efficient than creating or switching to a process due to the lower overhead cost of context switches. In addition, threads have full access to the address space which give programmers greater flexibility, allowing for shared data rather than message passing. Another important advantage of a multi-threaded process compared to a single threaded process is that the entire process does not block in case of I/O. In a multi-threaded process, other threads can continue to make progress while the thread doing I/O blocks.

In between single- and multi-threaded programming, there is **finite-state machine** (event-based) programming. Event-based programming attempts to achieve concurrency with a single threaded process, using

non-blocking calls and asynchronous communication (which is more complex to program).

From a software engineering perspective, writing multi-threaded applications is more challenging because a developer has to deal with event-based or blocking system calls which need to take care of the rest of the task when a thread finishes its execution. In comparison, writing a single-threaded application already assumes that all data is there at the moment of the next task execution. An example of such case could be reading a text file.

Examples for use of multi-threaded programs include browser actions such as clicking on a link for a web page. Upon clicking on a web link, images can be sequentially downloaded from the server then sent to be parses and then rendered. In a multi-threaded browser, images can be downloaded in parallel while the page is parsed and parts are rendered as they are ready. The browser does not have to wait for everything to be downloaded and parsed before rendering; as a result, the user will see parts of the page more quickly.

Multithreading is also used within servers. If a server only runs a single thread, it might have a queue of requests from clients which are blocked until the first request is completed. Using a pool of threads allows the server to respond to multiple requests simultaneously, thereby reducing latency. The idea is for the server to have a dispatcher and a few worker threads. When a client request comes in, the dispatcher assigns one of the idle worker threads to handle this requests. Efficiency is achieved because some of the worker threads are I/O bound and some are doing computation.

3.2.3.5 Thread Management

When programming with threads, developers can use **synchronization primitives** (such as locks, mutexes, or semaphores) to protect from shared access conflicts. For example, synchronization primitives prevent two threads from updating an object at the same time (in which one tramples the changes of the other). Without using these, programs may have **race conditions** in which concurrent operations have non-deterministic behavior, which can cause unexpected behavior or bugs which are difficult to find and resolve.

3.2.4 Module 3: User-Level vs Kernel-Level Threads

There are two types of threads: **kernel-level** threads (created and managed by the kernel) and **user-level** threads (created and managed by user libraries). For user-level threads, the entire functionality of threads is implemented in the user library. The kernel is *unaware* of there being multiple threads; the kernel sees the address space of the threads as a traditional single-threaded process. Whenever the thread as a process gets scheduled by the kernel, the user-level library will run and pick a thread with its own thread-scheduling technique. So the scheduling now becomes a two-level process: scheduling a process by the kernel and picking a thread to run by the user-level library. Creation of threads is very lightweight because it doesn't require system calls. It is also flexible because the developer can customize the scheduler rather than relying on the OS's scheduler. However, if any thread makes a blocking call, the entire process (all of its threads) will be blocked. Since the kernel is unaware of other threads, there is no real parallelism that can be achieved through multiple cores.

The operating system is aware of **kernel-level** threads, and can schedule them explicitly in the same way that it can schedule processes. The advantages of kernel-level threads are that they allow for real parallelism between threads, and only a single layer of scheduling is needed. The disadvantages, however, are that they are more expensive (requiring context switches), and less flexible (since they must use the OS's scheduling policy).

3.2.4.1 Scheduler Activation

Scheduler Activation tries to bridge the gap between user- and kernel-level threads. Scheduler activation is a mechanism for providing kernel-level thread functionality with user-level thread flexibility and performance. The library explicitly tells the kernel of the number of threads; while there is still no true parallelism, this allows for concurrency: you do not need to block a process just because a thread blocks.

3.2.4.2 Light-Weight Processes

Light-weight processes (LWP) are an abstraction between processes and threads, originally used in Solaris and some UNIX systems. For each process, the developer specifies the number of schedulable entities, and maps threads to entities. Schedulable entities are scheduled by the kernel like processes, while threads mapped to an entity are scheduled at the user level. You may have multiple LWPs per process, and depending on the mapping of threads to LWPs, the developer can decide the level of parallelism in a process. At the two ends of the spectrum, 1:1 mapping of threads to LWPs is similar to kernel-level threading, since each thread will be scheduled by the operating system and can be scheduled concurrently; N:1 mapping, in which all threads of a process are mapped to a single LWP, is similar to user-level threading.