CMPSCI 677 Operating Systems **Spring 2016**

Lecture 3: January 27

Lecturer: Prashant Shenoy *Scribe: Sandhya Sankaranarayanan*

3.1 Processes : Review

Multiprogramming: Running more than one process concurrently. This can be done on a uniprocessor, but it will not be true parallelism.

Multiprocessing: Running multiple processes concurrently on multiple processors. Hence, more than one process can be run in parallel.

Kernel Data Structure: Also called Process Control Block(PCB). This tracks details used by the operating system for process management like active processes, allocated resources, RAM, CPU time.

Address Space: Region of memory the process can access. Has a minimum of three segments:
Code/Text: Stores machine code for the process
Stack: Stores static allocated structures like local and global variables
Heap: Stores dynamic allocated structures like malloc

Process States: The state of a process describes what the process is going through at a given time. The possible states in the lifetime of a process are:
New: When the process is being set up and loaded.
Ready: When the process has been loaded in memory and is ready to run, but has not been scheduled on the CPU yet.
Run: When the process is executing in the CPU.
Wait: When the process needs some I/O operations. Process will be blocked until I/O has completed.
Terminate: When process is done, and is shut down.
Each process is run for a time slice (or quantum) until it either finishes, or starts an I/O operation. An I/O operation will move the state of the process from *run* to *wait*. When the I/O operation has ended, the process will then be changed from *wait* back to the *ready* state so that it can be scheduled to run again. If the time slice expires before the process ends, the process moves back to the *ready* state. Some processes have only computations and no I/O, in which case, they do not enter the *wait* state at all.

Process Behavior: A process usually alternates between being CPU bound or I/O bound. A CPU bound process primarily does computation, and might do a small quantity of I/O work. An I/O bound process primarily does I/O operations along with little computation.
When a process is doing some computation, it results in a CPU burst, which are typically short. Graphically, the CPU utilization of a process that alternates between being CPU bound and I/O bound can be represented by a hyper-exponential function.

3.1.1 Uniprocessor Scheduling Algorithms

The policy that decides which process from the ready queue is selected and given CPU time to run is determined by the scheduler. Some algorithms for this are:

- **FIFO:** First-in First-out takes runs the processes in the ready queue in order of arrival, until they terminate. In this policy there are no time slices.
- **Round Robin:** In combination to the first come, first served concept of FIFO, this policy creates time slices, so that the other processes in the ready queue do not get starved of CPU time if the process before them is heavy. When the time slice assigned to a process expires, it is moved to the end of the ready queue, and the next process moves into the run state.
- **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 drawback is that the length of the job must be known, and this is not always realistic. It also can starve heavy jobs if lighter jobs keep joining the queue.
- **SRTF:** Shortest Remaining Time First works same as Shortest Job First, but instead of considering shortest run time for jobs, it considers shortest remaining time to run.
- **Lottery Scheduling:** This is a randomized scheduling algorithm where each job is given a number of lottery tickets. 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 a greedy solution and is mostly used in mission-critical 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).
- **Priority queues:** Higher priority processes get to run first. For example, real time tasks are assigned higher priority than others, and hence are run first.
- **Multi-level feedback queues (MLFQ):** MLFQ contain multiple round robin queues. Each queue has a different priority level and processes can dynamically move from one queue to another depending on their current priority. The scheduler always picks a process from the highest priority queue that is non-empty, and uses round robin to pick a job within the queue. I/O bound processes are inherently smaller, and usually leave the queue before their time slice expires, whereas CPU bound processes mostly use the CPU for the entire allotted time slice. Thus, a higher priority is assigned to I/O bound processes than the CPU bound processes. While the OS can dynamically assign/change the priority assigned to tasks, users can also specifically set process priority.

Choice of a policy depends on the scenario, workload and system. By picking a suitable policy, the right performance metrics can be optimized. Some popular metrics are throughput, CPU utilization, response time, turnaround time and fairness.

3.2 Processes & Threads

A thread is a stream of execution. A traditional/single-threaded process executes using a single thread. In a multi-threaded environment, multiple streams of execution(threads) share a single address space and can execute in parallel.

Within the shared address space, each thread executes its own code(execution can be tracked through a program counter: a register that tells us where the execution is) from the code segment, contains its own stack because any function might need all the variables from the stack, but shares the heap with the other threads in the same process. Each thread has a copy of stack, program counter and registers unique to that particular thread.

True parallelism while running a multi-threaded process can be achieved on a multi-core system by executing different threads on different cores concurrently.

Synchronisation must be implemented when different threads are trying to access/modify data in shared data structures. This can be done in many ways, like creating locks, monitors or semaphores.

3.3 Advantages of Threads

- Threads are useful when you need true parallelism. Threads can utilize the additional CPU resource provided by multi-core systems.
- If an application is distributed, switching between threads is cheaper and more lightweight than switching between processes. This is because threads share the same address space. Switching between processes incurs higher context switch overhead. Hence, multi-threaded processes make better use of resources.
- Thread creation, deletion, switching are less expensive(only requires modifications in an already existing address space) than process creation, deletion, switching(requires the creation/removal/copying data of an entire address space). Hence, thread management is more efficient than process management.
- - In a single-threaded process, blocking calls such as I/O in the thread blocks the entire process.
  - This can be overcome in a single-threaded process by implementing a finite state machine. This is an event based design that has non-blocking asynchronous I/O calls. However, it requires specific changes the code in order to change how I/O is handled, and is complicated.

  In a multi-threaded process, true parallelism is achieved and a non-blocking call in one of the threads in a process will not block the process if even one of the other threads are still executing.

Hence, introducing multi-threading in operations will bring down the system overheads.

3.4 Examples of Multi-threaded Systems

3.4.1 Multi-threaded Client : Web Browsers

Web browsers are not distributed, they run on just one machine. There are multiple tasks associated with the loading of a page by the browser, such as loading the graphical user interface, event handling, communicating with remote servers, parsing the contents of the HTML page, downloading and rendering images.

In a single threaded browser, these tasks have to execute sequentially in one thread, and the page will not load completely until each and every task related to loading a web page has been completed. In most cases, this will result in a long waiting time, as each task will block all the following tasks until it has completed.

This can be overcome by modularizing these tasks into different threads, so that they can occur in parallel and the page will load faster.

3.4.2 Multi-threaded Servers

Servers receive requests from clients, process them, construct responses and send them to the clients. In most cases, multiple requests arrive at the server at the same time.

In a single-threaded sequential server, the later requests are forced to wait until the earlier ones have been processed, resulting in a longer waiting time by the clients.

In a multi-threaded server, an example is a thread per request model is used. In this model, a new thread is dynamically spawned for every new request that comes in. Since multiple threads are concurrently servicing different requests, parallelism is achieved. Once the request has been serviced, the thread that was spawned for that service is terminated.

Another example for a multi-threaded server design is creating a static pre-spawned pool of threads(worker threads) every time a server is started, and keeping them alive in an idle pool. The number of threads created(n) is the degree of concurrency supported by the server. A dispatcher thread sends out the requests to an idle thread. If an n+1 th request arrives at the server while n requests are currently being serviced, it will have to wait until one of the threads free up.

The Apache Web Server uses the pre-spawned worker threads model with an improvement - it can increase or decrease the number of worker threads on the fly, if there are too many or too few requests coming in.

3.5 Thread Management

3.5.1 User-level Threads

No operating system features are used for multi-threading, it is implemented entirely using libraries. The operating system is not even aware of the multi-threading in a process, it assumes that each process is single-threaded. The process is scheduled by the kernel, and a second-level library scheduler schedules the threads within the process. This exhibits many-to-one mapping: many user level threads map to one process.

Advantages:

- Operating system does not have to support multi-threading. Multi-threaded code can be run by linking in the appropriate libraries.
- Thread management is cheap and efficient as no involvement from the operating system is necessary, and can be carried out with function calls instead of system calls.
- A variety libraries are available that have a range of implementations and scheduling algorithms. Hence, the policies in the libraries can be used if they are more suitable for the application, independent of the policies in the kernel scheduler.

Disadvantages:

- Since the operating system is not aware of threads, true parallelism will not be achieved for a process even if multiple cores are available, as more than one core will never be resourced. However, concurrency is achieved.

- I/O operations will block the entire process.

3.5.2 Kernel-level Threads

The kernel provides explicit support for multi-threading. Kernel knows exactly how many threads are involved in the processes. The threads are scheduled, not the processes.

Advantages:

- True parallelism can be achieved.
- I/O operations in one thread will not block the process if other threads are not blocked.

Disadvantages:

- Thread creation is heavy weight, it involves system calls.
- Only the scheduling policy in the kernel can be used by the application.