CMPSCI 677 Operating Systems **Spring 2018**

Lecture 3: January 29

Professor: Prashant Shenoy *Scribe: Assan Toleuov*

3.1 Overview

This lecture covered the following topics.

P2P architecture: Collaborative Distributed System. BitTorrent

Edge-server (client-proxy-server) Systems

Self-managing or Autonomic Architecture

Review of Processes and Threads

3.2 Lecture Notes

3.2.1 P2P architecture: Collaborative Distributive System

3.2.1.1 BitTorrent

BitTorrent is a collaborative distributed system for distributing large amounts of content. The key feature is the idea of parallel downloads, i.e. when a client tries to download a large file, it will connect to a number of other peers that already have that file and try to download different chunks (pieces) of the file in parallel. Clearly, parallel download results in a speed-up over sequential download. The degree of parallelism depends on many factors including the internet connection speed and amount of bandwidth one's machine has. There is also a notion of explicit incentives. A BitTorrent client is tracked for the amount of outgoing and incoming content on their machine. If the client is not willing to share enough data, they end up with a slower download speed.

To download a file, a BitTorrent client first downloads a torrent file from the Internet. The torrent file contains the list of tracker nodes. Trackers can have the content themselves or point to the other node with the content. The client can then figure what chunks to get from which node. The degree of parallelism is the number of nodes that contain some chunks of the file.

3.2.2 Edge-server (client-proxy-server) Systems

Edge-server architecture (or client-proxy-server) is often useful when the actual server is geographically away from the client. Proxy servers have lower latency or higher bandwidth than the actual server. Edge servers are at the edge of the network i.e, one or two hops away from the client when the actual server might be many hops away. Edge servers sit between client and server. Edge servers can cache some of the content and serve clients without sending requests to the actual server, therefore reducing the load on the server.

3.2.3 Self-managing or Autonomic Architecture

Self- managing or autonomic systems are adaptive and intelligent systems. They make autonomic decisions without human intervention. Self-healing systems, self-scaling systems are some examples of an autonomic architecture. Self-healing systems are able to fix failure or bring other systems online to take over the role of the failed node. Self-scaling systems such as cloud network make autonomous decisions about the size of the computing power need. Such autonomic systems typically do self-observation or monitoring of their own work, modeling and prediction, and making decisions based on that. For example, a web server can scale its capacity as the load on the server rises.

3.2.4 Processes and Threads

3.2.4.1 Review of Processes

Multiprogramming: Multi-programming stands for multiple processes/ applications running at the same time. All programs that are loaded in main memory and ready to execute compete for CPU time. The goal of multiprogramming is to use CPU time maximally and keep CPU busy unless there is no new processes coming. A process has three segments: stack, heap, and code segment. Each process has its own address space, code, and global and local variables.
Multiprocessing: Muti-processing is a different idea from multi-programming. It signifies multiple processors/cpus or multiple cores. With a system that supports multi-processing, we can get a true parallelism.
Scheduling: Kernel schedules processes. There are preemptive and non-preemptive schedulers. In the case of a preemitve scheduler, each processes gets a time quantum to use the cpu. When the time ends, the process gets preempted and the next process get the CPU time. In case of a non-preemptive, the process runs through completion or gives up the CPU for i/o. The kernel will not preempt the process.
Process Control Block: PCB is a data structure that helps kernel to schedule. It has data about what processes are active, how much memory is allocated to each process and where memory pages are stored in RAM for each process. Each process gets its own stack; heap, code and BSS are shared. The process code, stack, heap are stored in the RAM.

Uniprocessor Scheduling Algorithms:

- Round-robin: Each process gets a time quanta in CPU. Processes are executed in a circular order and without any priority.
- Shortest job first: SJF scheduling has the minimum wait time. In practice, SJF is not completely feasible because the length of the job needs to be known beforehand.
- FIFO: The first process that comes in gets to run first. FIFO is non-preemptive scheduling.
- 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.
- EDF: In Earlier Deadline First scheduling, an application decides the deadline for a CPU task. EDF is useful in real-time systems where time guarantee is important.

Performance metrics: Throughput, turnaround time, response time are some measures of performance. There are trade-offs between these metrics because the resources available to the processes are fixed. Developers can decide which performance metrics they want to optimize. 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. important. Scheduler needs to be carefully selected based on the desired performance metrics.

3.2.4.2 Process Behavior

I/O bound and CPU bound: Processes alternate between being CPU-bound and I/O-bound. For scheduling it is important to know that most CPU bursts are short, a few are long. CPU-bound processes spend most of their lives doing computation and I/O bound processes spend most of their lives doing I/O operations.
CPU bursts: Cpu bursts show a hyper-exponential behavior. Most CPU tasks last for a small duration. CPU burst graph has a long tail because there are a few tasks that last for a long duration.

3.2.4.3 Process scheduling for independent jobs that exploit CPU behavior

Priority queue: Each job is assigned a priority level and jobs are scheduled based on the priority. The highest priority task keeps running until it's 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.
Multi-level feedback queue(MLFQ): In this scheduling, the OS keeps multiple priority queues with different levels of priority. A round-robin scheduling is used within queues and priority scheduling is used across queues. The OS run the processes in the highest priority non-empty queue first.
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.

3.2.4.4 Process vs. Threads

Traditional processes have one stream of execution and are also called single-threaded. A process can also have multiple streams of execution. Each stream is called a thread. If two threads belong to the same process, they share the memory of that process. Different threads of a process share the same code but are executing different parts of the code. Each thread has its own registers and stack. The registers store program counter and stack pointer. Threads share the same address space. They share the heap.

If we have multiple cores, we can put threads on different cores which would give true parallelism. However, if there is only one core, it will look like the threads are executing in parallel while in reality they are executing concurrently. Two threads can concurrently write to the same variable if synchronized properly.

With the advent of technology, machines have got more cores. To take advantage of the new resources,

threads allow faster execution of processes by being scheduled on additional cores independently. Creation of a new thread is much cheaper than creation of a new process. Programmers can modularize their codes keeping in mind that switching between threads is less expensive than switching between processes because threads are lightweight and share the same memory. Threads have full access to the address space which give programmers greater flexibility. 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 even-based programming. Event-based programming attempts to achieve concurrency with a single threaded process. I/O has to be non-blocking. From a software engineering perspective, writing multi-threaded (or asynchronous) 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 the parse and then rendered rendered. Another way to render a web page is to use multiple threads. N images can be downloaded in parallel, parser and render working in parallel as things get downloaded, n images can be downloaded in parallel. Images that have been downloaded can be sent to the parser and then rendered. The browser does not have to wait for everything to be downloaded and parsed before rendering.

Multiple threads are also applicable from the server perspective. If a server only runs a single thread, it might have a queue of requests from the client. This increases latency that the users see. Multi-threaded server with pool of threads can reduce this 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.

3.2.4.5 Thread Management

There are two types of thread: kernel-level threads (created and managed by the kernel) and user-level threads (created and managed by user libraries).

In the user-level threads, the entire functionality of threads is implemented in the user library - OS kernel is not aware of and does not support them; OS kernel only supports processes. OS kernel does not see the presence of threads; the kernel sees the address space of the threads as a traditional 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 in terms of a scheduler. Among disadvantages, if any thread makes a blocking call, the entire process gets blocked i.e. all threads get blocked. Also, since the kernel does not know of the presence of the threads, there is no real parallelism that can be achieved through multiple cores.
[kernel-level threads are part of the next lecture]