CMPSCI 677 Operating Systems **Spring 2013**

Lecture 3: January 30

Lecturer: Prashant Shenoy *Scribe: Demetre Lavigne*

3.1 Distributed Systems Examples

3.1.1 Edge-Server Systems

Edge server systems take the standard client-server model and combine it with a proxy. This client-proxy-server model uses the proxy, which is located at the *edge* of a clients network, to answer a clients request if it can. The proxy works like a cache for the network. It is often the case that multiple clients will make the same request and server load can be reduced if a proxy can just resend the appropriate response. This is used for content distribution networks where many users are requesting the same resources.

3.1.2 Collaborative Distributed Systems

A collaborative distributed system is one where multiple nodes work together, or *collaborate*, to accomplish a goal. A popular example of this is BitTorrent which allows collaborative peer-to-peer (P2P) downloads. In BitTorrent the user first acquires a tracker file which includes metadata about how to contact peers who have the desired content. The user then connects to active nodes, in parallel, to download chunks of the file that they want. The system is collaborative because it forces altruism by limiting users who download more than they upload to the system.

3.1.3 Self-Managing Systems

Self-Managing systems are autonomous in that they have the ability to resolve problems without administration. The system monitors itself and takes action if it is needed. An example of this type of system is a web server that can perform automatic capacity provisioning. This means that the system monitors the demand on the webserver and estimates what the demand will be in the future from past information and adjusts the capacity of the server accordingly.

3.2 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).

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 processes 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 processes 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 processes time slice expires without the process finishing or starting an I/O operation, then the process moves back to the ready state.

3.2.1 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 not 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 processes 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 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. 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 next. 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).

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.

This leads to multi-level feedback queues (MLFQ) which 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). 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 higher in priority. Otherwise, if the job instead did I/O, then it is moved to a queue on level lower in priority because it is I/O bound. Lower priority queues get longer time slices (and the time slices increase exponentially).

3.3 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 a software engineer an application as a bunch of threads. This is because each thread will perform some generally independent task. For example, a web browser would use threads to fetch separate parts of a web page so that it can display some data before it has all downloaded. A web server is another example where a pool of threads could be given requests by a listener thread to process. A single threaded version of a web server might need to queue requests and handle them individually.