Lecture 3: February 4

Lecturer: Prashant Shenoy *Scribe: Ishita Dasgupta*

3.1 Some other Architectures

3.1.1 SuperPeers

Instead of flooding as in Unstructured peers, peers can come together as groups where a SuperPeer is elected(one with more bandwidth, resources or better network connectivity might be chosen).It is responsible for keeping track of objects all the peers have, thus handling the queries and offloading the burden from other peers. They act as a server on behalf of other nodes. The choice or election is dynamically done by the group. Example:- Skype super peer. Skype needs to keep track of your ever-changing IP address and online status in a directory, if someone needs to call you. This directory is maintained in a P2P network. Here, groups of peers elects a SuperPeer who is delegated the responsibility for answering queries about the status of other peers.

3.1.2 Client-proxy architectures (Edge-Server Systems)

In a client-server architecture, a server may get overloaded if there are lots of clients. In that case,**Edge Servers** are added, which are also called proxy servers that have lesser responsibilities than an actual server. It caches data thats on the main server. The client goes to the nearest edge server and sends the request, where the requested item is already cached. Proxies can also be used for purposes more than caching.

3.1.3 Collaborative Distributed Systems(BitTorrent)

Nodes are going to collaborate with one another and each node contributes to some content.
BitTorrent is used to download large content. Large content is divided into smaller pieces and distributed into multiple nodes and each small piece is downloaded in parallel. Before downloading any object, a posted torrent file is needed. A Torrent file has the information about which node is storing what content and how to get it, as well as who is live and what chunks do they have. Tracker tracks the required nodes,and client downloads the chunks from them.

This architecture encourages collaboration. If it sees a particular nodes has repeatedly more downloads than uploads, then the node isn't sharing resources, hence its reduces the rate of downloading for that particular node, the next time, for selfishly not contributing its resources.

3.1.4 Self managing/Autonomic Distributed Systems

– They make their own intelligent decisions without users telling them what to do.

- Vary capacity of a server based on demand. It monitors(requests per second)and makes future predictions on what the workload is going to look like, and then accordingly adjust allocation. For ex:- If it sees that the workload has increased, it fetches and keeps more resources ready.
- Makes capacity decisions.
- Handles failures automatically

3.2 Processes

Multiprogramming, or *multitasking* is when a system runs multiple processes and it switches between them (based on a scheduler). This can be done on a uniprocessor and without any parallelism. Multiprocessing, however, runs different processes ,more than one at a time on multiple processors(making them multiprogrammed by definition). Thus, there will be concurrency in this case. For example:- Quad core processors. A process has a state which varies during its lifetime.The states of a process are:

* **new**: When a process starts up and is loaded.
* **ready**: The process is ready to run, but not yet scheduled on the CPU.
* **running**: The process is executing in the CPU.
* **waiting**: Mainly during I/O operations.
* **terminated**: Once the process is done, its shut down.

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.

The operating system maintains a data structure for keeping track of all the processes in an Operating system kernel called the process control block (PCB). PCB needs to keep to track of all aspects of the processes and which state they are in. For ex:- All memory allocation, Disk I/O, state I/O, network I/O information,File I/O, state of the process, user of the process, etc are tracked in this data structure and used for process management.
Logical view of the process as it resides in RAM is a linear sequence of bytes associated with the process. Each process has multiple segments- An address space with a code (or text) segment, heap, stack and maybe shared libraries.The heap is dynamically allocated and stack has the variables.

3.2.1 UniProcessor Scheduling Algorithms

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 different CPU scheduling policies are:

- **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(converted from running to a ready state) so that other processes that are ready can run for a while. This process ensures that no process has to wait for a long time to run.
- **FIFO:** First-in First-out takes processes and runs them, in order of arrival, to completion. In this policy there are no time slices.
- **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 practical implementation for this method is a drawback 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.
- **SRTF:** Shortest Remaining Time First works same as Shortest Job First, just 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 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 in 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) and little I/O work. The other type is **I/O bound**, which means that the process is doing mostly I/O operations along with little computation. Of course a process can alternate 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 more than one Round-Robin queues. Each queue has a different 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 non-empty queue.(using round robin to pick a job within the queue). I/O bound tasks are inherently smaller, and use CPU lesser whereas CPU bound tasks mostly use the CPU for the entire allotted time slice. Thus, since I/O bound tasks wait lesser fraction of the time slice, we give higher priority to I/O bound tasks. This process is sort of an implementation of SJF scheduling algorithm, where instead of unknown job lengths,known process behavior is taken into account. 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).

3.3 Processes & Threads

Each stream of execution is referred as a thread. A typical process executes using a single thread. In multithreaded environment, instead of one thread, each of the multiple threads concurrently executes different instructions or functions.
Each thread in a process is assigned one address space. Each process has its own stack because any function might need all the variables from the stack, but shares the heap. Each thread executes some arbitrary part of the process.Each thread has a copy of stack, program counter and registers unique to that particular thread. Threads can be scheduled exactly like processes. It implements concurrency & parallelism. True parallelism can be executed in a multi-core system. Locks can be used to implement synchronisation by prohibiting multiple threads to override each other.
Web browsers are an example of multithreaded environment.They are a faster alternative to single-thread environment because the later would consume much more time to execute each process sequentially whereas the former speeds up the process via parallelism.