Cheng | Midterm 1: Chapter 5 | Study Guide

Chapter 5: CPU Scheduling

CPU Scheduling Criteria:

CPU-Scheduling Decisions

1. When a process switches from a running state to the waiting state. [Invoking a wait().]
2. When a process switches from the running state to the ready state. [When an interrupt occurs.]
3. When a process switches from the waiting state to the ready state. [Completion of I/O.]
4. When a process terminates.

Non-preemptive vs. Preemptive Scheduling:

Non-preemptive: When scheduling takes place under circumstances of 1 & 4, once a CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.

Preemptive: A running process may be forced to release the CPU even though it is neither completed nor blocked.

Take the following case study for example…

Two processes that share data. While one process is updating the data, it is prempted so that the second process can run. The second proces then tries to read the data which are in an inconsistent state.

CPU Scheduler and Dispatcher:

Dispatcher: Module that gives control of the CPU's core to the process selected by the CPU scheduler.

How is this done? It must switch context from one process to another, then switch to user mode, and then it must jump to the proper location in the user program to resume that program. The time it takes for the dispatcher to stop one process and start another running is known as dispatch latency.

Scheduler: When the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler which selects a process from the processes in memory that are ready to execute and allocated the CPU to that process.

CPU Scheduling Criteria: How a CPU-scheduler chooses a process is due to it's algorithm, and different algorithms have different properties.

CPU Utilization: We want to keep the CPU as busy as possible.

Throughput: If the CPU is busy executing processes, then work is being done.

Turnaround Time: How long it takes to execute that process, the interval of time from submission to execution is turnaround time.

Waiting Time: It affects only the amount of time that a process spends waiting in the ready queue.

Response Time: Time interval from submission until the first response is produced, it is the time it takes to start responding, not the time it take to output the response.

Meeting-the-deadline: Real-time systems.

Scheduling Algorithms:

FCFS: First-Come, First-Served Scheudling:

With this scheme, the process that requests the CPU first is allocated the CPU first. This is implemented using a FIFO queue, but the downside is that the average waiting time is often long.

But what if a short process is behind a long process? This may lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle, hence the convey effect.

SJFS: Shortest-Job, First Scheduling:
This algorithm associates with each process the length of the process's next CPU burst. When the CPU is avaliable it is assigned to the process that has the smallest next CPU burst. If there is a tie, FCFS is used to break the tie.

But how do we know what the length of the next CPU request is?

This algorithm can either be preemptive or nonpremptive:

Preemptive: Take the idea that a new process process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process, so the **SRTF Shortest-remaining-time-first**, will preempt the current executing process.

Nonpremptive: Where the algorithm will allows the currently running process to finish it's CPU burst.

Round-Robin Scheduling: Similar to the FCFS scheudling, but preemption is added to enable the system to switch between processes.

You might still be lost and asking yourself how does it work exactly?

A time quantum is defined, which is just a unit of time. Imagine the ready queue as a circular queue, where the CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of up to 1 time quantum. The CPU scheduler picks the first process from the ready queue sets a time interupt after after 1 time quantum, and dispatches the process.

> The best way I can put this is you have a set of tasks to do, each task is given the same amount of time to complete. If you finish the task early, great you can hit a buzzer that says yay and then you can start the next task early, but if you don't finish the task under the required time, you have to put down that task and start the next task but the task that you had to drop gets put at the end of list of things to do.

The time quatum must be not be so large as to make the RR algorithm like the FCFS algorithm but not smaller then the context switch between processes.

Priority Scheduling:

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order.

Starvation: Low priority processes may never execute.

Aging: As time progresses increase the priority of the process.

Multi-level Queue Scheduling:

The scheduler then selects the process wit the highest priority to run and depending on how the queues are managed with an O(n) search may be necessary to determine the highest-priority process. It is easier to have different queues with different priorities, like one for higher-priority and one for lower-priority.

A multi-level queue algorithm can be used to partition processes into several queues based on the process types.

Foreground: Interactive processes which have higher priority than background processes.

Background: Batch processes.

Well what the heck is **Multi-level Feedback Queue Scheduling**? It kind of sounds the same.

When a process enters either a foreground or background queue, it is inflexible, it can't move between queues. The feedback version of this type of queueing allows a process to move between queues. But if the nature of a process doesn't change then what is the point? The idea is to separate processes according to the characteristics of their CPU bursts. If one process is taking too much CPU time, it gets moved to the lower-priority queue.

This is a form of aging that prevents starvation!

Linux Version 2.5: This was taken from his power-point, the book does not go into this level of detail on this.

Version 2.5 moved to constant order O(1) scheduling time.

- Preemptive, priority based
- Two priority ranges: time-sharing and real-time
- Real-time range from 0 to 99 and nice value from 100 to 140
- Map into global priority with numerically lower values indicating higher priority
- Higher priority gets larger q
- Task run-able as long as time left in time slice (active)
- If no time left (expired), not run-able until all other tasks use their slices
- All run-able tasks tracked in per-CPU runqueuedata structure
- Two priority arrays (active, expired)
- Tasks indexed by priority
- When no more active, arrays are exchanged
- Worked well, but poor response times for interactive processes

Linux Version 2.6.23 and Beyond: This was taken from his power-point, the book does not go into this level of detail on this.

- **Completely Fair Scheduler (CFS)**
- Scheduling classes
    - Each has specific priority
    - Scheduler picks highest priority task in highest scheduling class

  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    - Default
    - real-time
- Quantum calculated based on nice value from -20 to +19
  - Lower value is higher priority
  - Calculates target latency –interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task virtual run time in variable vruntime
  - Associated with decay factor based on priority of task –lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time