CPU scheduling is the basis of multi-programmed operating systems. By switching a CPU core between different processes or threads, the operating system can make the computer more productive. This chapter introduces the basic scheduling concepts and discusses in details on CPU scheduling algorithms including FCFS, SJF, SRTF, Round-Robin, Priority, multi-level queue, multi-level feedback queue (MLFQ), and real-time scheduling algorithms. It also examines issues relevant to multiprocessor scheduling and ways of evaluating scheduling algorithms.

Basic on Scheduling

- Each process or thread during its lifetime of execution typically goes through a sequence of *CPU bursts* and *I/O bursts*. Hence, the lifetime of a process can be modeled as switching between bursts of CPU execution and I/O wait.
- **The length of the next CPU burst** is critical in some scheduling algorithms such as SJF and SRTF, which can be estimated based on the historical CPU burst time lengths using an **Exponential Averaging** algorithm. The predicted value for the next CPU burst length depends on 1) all the actual CPU burst time lengths of this process prior to the estimation and 2) the initial estimation.
- There are different criteria relevant to scheduling, and they may be conflicting with one another: (1) *CPU utilization*, calculated by the fraction of time that CPU is busy within a time duration; (2) *throughput*, defined as the number of processes or jobs completed per unit time - this metric apparently favors short processes; (3) *turn-around time*, the amount of time to execute a particular process. In case that only one CPU burst time is considered, the turn-around time is the sum of waiting time (on the ready queue) and the CPU burst time; (4) *waiting time*, the sum of all waiting times of a process on the ready queue; and (5) *response time*, which is the time between the arrival time (the time that process joins the ready queue) and the first CPU time, commonly only measured in RR or in interactive systems.
- ***Turn-around time*** in general is the sum of the time that a process is spent on waiting, executing on the CPU, and doing I/O, which essentially measures the amount of time it takes to execute a process or complete one CPU burst time.
- ***Response time*** measures the time between a process request and the first response produced, typical the first CPU burst time or the end of first quantum if a RR is used. This is more relevant to interactive jobs.
- Short response time favors RR type of scheduling, while it may result in longer turn-around time and longer waiting time than FCFS.
- CPU scheduling is the task of selecting a waiting process from the ready queue, and CPU is then allocated to the selected process by the **dispatcher**.
- The **dispatcher** gives control of the CPU to the process selected by the CPU scheduling. To perform this task, *a context switch*, a switch to user mode, and a jump to the proper location in the user program are required. The dispatcher should be made as fast as possible. The time elapse is termed *dispatch latency*.
- **Non-preemptive** scheduling policy is invoked only when the current process running on the CPU gives up the CPU voluntarily either due to the termination

of the process or the completion of its current CPU burst (e.g., waiting for I/O). Otherwise, the scheduling is **preemptive** in nature, where the CPU can be taken away from a process involuntarily. For instance, scheduling might occur when a new process with higher priority joins the ready queue either as a new process or as a process from waiting to ready (e.g., completion of I/O). Almost all modern operating systems are preemptive.

FCFS

- The scheduling is based solely on the arrival orders of the processes on the ready queue. This is non-preemptive in that each process once occupying the CPU is allowed to run to the completion of its current CPU burst.
- The main problem with FCFS is that a short process might get stuck behind a long process. This phenomenon is referred to as the **convoy effect**. This can result in excessively long average waiting time.

SJF

- Shortest Job First or SJF scheduling is based on the next CPU burst lengths of the processes. It is proved to be the ***optimal*** scheduling algorithm (non-preemptive) resulting in the minimum average waiting time for a given set of processes.
- Shortest Remaining Time First or SRTF is the preemptive version of SJF, in which a newly arrived process (from new to ready) or a process just joining the ready queue (from waiting to ready) can preempt a process currently running on CPU if its CPU burst length is shorter than the remaining CPU burst time of the process running on the CPU. Note that it is obvious that the CPU burst lengths of all processes on the ready queue are larger than the CPU burst length of the process running on the CPU, that is the SJF part of the SRTF.
- Implementing SJF or SRTF scheduling is difficult, however, as predicting the length of the next CPU burst can be complex or inaccurate. Hence, this is usually used as a comparison for other more practical scheduling algorithms.

RR

- Round Robin or RR scheduling is done based on an ***FCFS order*** but restricts each process to occupy the CPU no more than a pre-determined **quantum**. This allows each process to receive a fair share of CPU usage, and reduces the response time.
- A process completing its quantum will join at the end of the ready queue (FCFS).
- By ignoring the context switching time, RR scheduling results in better response time for each process, which is calculated as the time between the process arrival time to the ready queue and the completion of the first quantum or CPU burst time (whichever is shorter) for that process.
- The value of quantum has to be carefully designed based on the characteristics of the process CPU bursts. At one extreme, if the time quantum is sufficiently large (larger than all CPU burst times), the RR policy reduces to the FCFS policy. If the time quantum is extremely small, the RR approach is called *processor sharing* and creates the appearance that each of **n** processes has its own processor running at

`1/n` the speed of the processor (assuming the context switch time is negligible).

The Estimation of Next CPU Burst – Exponential Average Algorithm

- Each process execution goes through many rounds of *CPU bursts* and *I/O burst*s during its life time. The estimation of the next CPU burst length is determined by the actual CPU burst length measured and estimated in the previous round.
- Simple algebra expands the formula, which states that this estimation is determined by the actual CPU burst lengths in all past rounds and the initial estimation value. The CPU burst lengths in the recent rounds weigh more than those in earlier rounds, i.e., the exponential factor of 1-alpha.

Priority Scheduling

- Priority scheduling assigns each process a ***priority***, and CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling (also FCFS order).
- Noticing that SJF and SRTF are both priority scheduling, in which the priority is the inverse of predicted next CPU burst time.
- ***Starvation*** occurs when a process is ready to run but is stuck waiting indefinitely for the CPU. This occurs, for example, when there are always higher-priority processes that prevent low-priority processes from acquiring the CPU.
- ***Aging*** involves gradually increasing the priority of a waiting process so that the process will eventually achieve a priority high enough to execute if it has waited for a long period of time. In practice, this requires a timer to be issued for each process when joining the ready queue.

Multi-level Queues

- Multilevel queue scheduling partitions processes into separate queues arranged by a priority, and the scheduler executes processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.
- The CPU is allocated to different queues, typically either by priority or a percentage (time slice) of CPU time.
- A process is usually assigned permanently to one queue, it cannot be moved to a different queue.

Multi-level Feedback Queues

- Multi-level feedback queue scheduling algorithm allows a process to move from one queue to another.
- There are several parameters that need to be specified for this type of scheduling algorithm, including, number of queues, which queue to join initially at a specific time, scheduling algorithm for each queue, and the policy on how to move a process between different queues.

Multi-level Feedback Queues Example

- Scheduling – preemptive among different queues. Noticing that it is only feasible that a new process (joining queue 0) can preempt a job from either queue 1 or queue 2 running on CPU. In another word, it is not possible that a job from queue 1 preempts a job from queue 2 even if jobs from queue 1 have higher priority than those from queue 2. Because there is no new arrival to queue 1 directly, only when a process completes its quantum from queue 0 on the CPU and yet completes its current CPU burst time, it will join the queue 1.

- A new job always enters queue 0 which is served in FCFS order (RR). When it gains CPU, the job receives up to 8-time units. If its CPU burst time is longer than 8-time units, RR timer expires and the job moves to queue 1. If it finishes, it releases the CPU. The jobs from queue 0 can never be preempted.
- A job from queue 1 is also served in FCFS order (RR), but only when queue 0 is empty. It receives up to 16 additional time units. If it still does not complete its CPU burst time (meaning its CPU burst time is longer than 24-time units), RR timer expires and the job moves to queue 2. If it finishes, it releases the CPU.
- This implies that if a job's CPU burst length is no larger than 24 (8+16) time unit, it will never enter queue 2.
- A job from queue 2 is served in FCFS order, but only when both queue 0 and queue 1 are empty.
- A process in queue 1 or queue 2 can be preempted by a process arriving at queue 0. The process being preempted will join at the **head** of the corresponding queues, queue 1 or queue 2 respectively, and continue execution later with the remaining quantum on queue 1 (total only 16 time units) or with the remaining CPU burst time on queue 2.

Real-time Scheduling

- *Soft real-time scheduling* gives priority to real-time tasks over non-real-time tasks. *Hard real-time scheduling* provides timing guarantees for real-time tasks.
- **Rate-monotonic (RM) real-time scheduling** schedules *periodic tasks* using a static priority policy with preemption. As the name suggests, it selects a process with a larger or largest rate (i.e., the smaller or the smallest period).
- **Earliest-deadline-first (EDF) scheduling** assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. This is also preemptive in nature. EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable – thus specifies its priority.

Thread Scheduling

- On operating systems that support both user-level and kernel-level threads, it is the kernel-level threads that are being scheduled and executed by the operating system. User-level threads are managed by the thread library, and the OS is not aware of them. User-level threads must ultimately be mapped to a corresponding kernel-level thread, indirectly or using a lightweight process (LWP).
- The thread library schedules user-level threads within one process to run on an

available LWP (under many-to-many and many-to-one models). This scheme is known as **process-contention scope** (PCS). Typically, PCS is done according to priority (set and determined by programmers) among the user-level threads belonging to the same process.

- The kernel or OS uses **system-contention scope** (CSC) to determine which kernel-level thread to schedule onto a CPU, among all kernel-level threads in the system. This is essentially the CPU scheduling we are discussing in this chapter.

Multiprocessor Scheduling

- In **asymmetric multiprocessing**, all scheduling decisions, I/O, and other system activities are handled by a single processor (the master). The other processors execute only user codes. This is simple in that only one processor accesses the kernel data structures, eliminating the need for kernel data structure sharing.
- In **SMP** or **symmetric multiprocessing**, each processor is self-scheduling. All processes can be in a common ready queue or in a separate ready queue for each processor. It must ensure that two processors do not choose to schedule the same process and processes are not lost from the queues.
- Multicore processors place more than one CPU core on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU. This results in a two-level scheduling, (1) schedule software thread (kernel thread) to run on a logical CPU (that is what we have covered in this Chapter), and (2) each core decides which hardware thread to run on the physical core.
- Two general methods of load balancing in SMP systems: (1) **push migration**, a specific task periodically checks the load on each processor and — if it finds an imbalance—evenly redistributes the load by moving processes from overloaded processors to idle or less-loaded processors; (2) **pull migration** occurs when an idle processor pulls a waiting task from a busy processor. Both push and pull migration strategies are often implemented in load-balancing systems.
- There is a high cost involving invalidating and repopulating cache contents during migration of processes from one processor to another. **Processor affinity** implies that a process has an affinity for the processor on which it is currently running. Some systems provide system calls to support **hard affinity**, thereby allowing a process to specify a subset of processor(s) on which it prefers to run. When an OS has a policy of attempting to keep a process running on the same set of processors, but not guaranteeing that, this situation is known as **soft affinity**.
- If an architecture features non-uniform memory access (**NUMA**), in which a CPU has faster access to some parts of main memory (for example, CPU and parts of the main memory are on the same board) than other parts of the memory, this could also result in affinity to both processor and main memory.
- Load balancing on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.

Algorithm Evaluation

- Analytical methods use mathematical analysis to compute the performance of scheduling algorithms, which requires exact input sequences to be known *a priori*. Queueing model can capture the performance of scheduling algorithms, but with only a few known mathematical distributions. Simulation methods evaluate performance by imitating a scheduling algorithm on a "representative" sample of processes (benchmark workloads) and quantifying the resulting performance, which can provide an approximation of the actual system performance.