CPU Scheduling

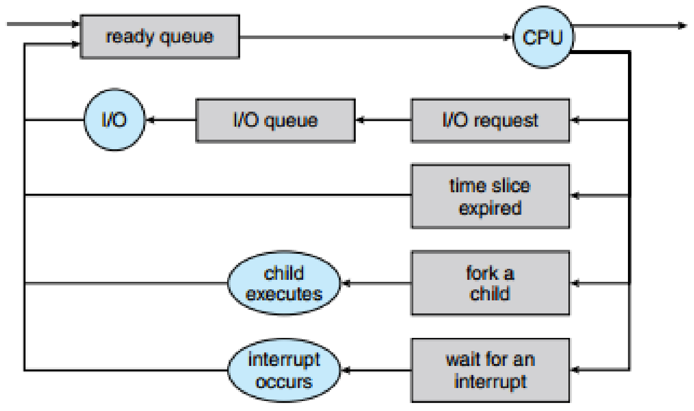
1 Process Scheduling

The objective of multiple programming is to have a process running at all times to maximize CPU utilization. Context switches are made when the processor is idle. After a process runs on the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.. The process scheduler is responsible to select an available process for execution on CPU.

Scheduling Queues

The queues are generally stored as a linked list containing **pointers to PCBs**. The header points to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the queue.

- **Job Queue:** All the processes in the system including disk.
- **Ready queue:** All the process that are residing in memory and are ready to execute.
- **Device queue:** All the processes waiting for a particular I/O device.



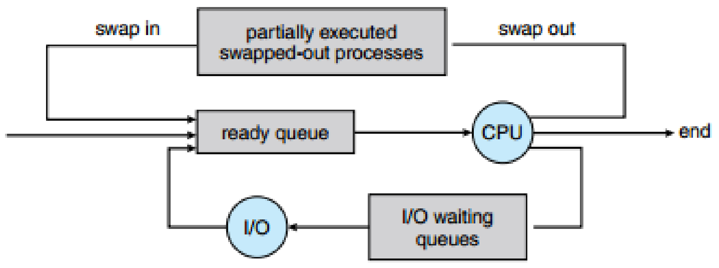
Schedulers

The OS is responsible for choosing a process to run on the CPU. This selection is carried out by a **scheduler**. In a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (disk) for later execution.

- The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.
- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The CPU scheduler must select a new process frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Memory is fast and can afford switches rapidly to give a perception of multiple processes running at the same time.

The job scheduler controls the **degree of multi-programming** (the number of processes in memory). If the degree of multi-programming is stable, then the average rate of process creation must be equal to the average departure rate of processes. It should select a good **process mix** of I/O-bound and CPU-bound processes.



The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multi-programming. The process is swapped out, and is later swapped in. This is called **swapping**. Swapping may be necessary to improve the process mix or because a change in memory requirements has **over-committed available memory**, requiring memory to be freed up.

Dispatcher

When the scheduler completes its job of selecting a process, it is the dispatcher which takes that process to the desired state/queue. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. **Dispatch latency** is the time it takes for the dispatcher to stop one process and start running another.

- Switching context
- Switching to user mode
- Jumping to the proper location of user program to restart the program.

2 Scheduling Criteria

Our goal is to improve the average of the above parameters and minimize the variance.

- Maximize **CPU Utilization**
- Maximize **throughput**.
- Minimize the **Waiting time** which is the sum of the periods spent waiting in the ready queue.

- Minimize **turnaround time** which is the interval from time of submission to time of completion. It is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on CPU and doing I/O.
- Minimize **Response time** which is the time from the submission of a request until the first response is produced. The turnaround time is limited by speed of the output device.

3 Scheduling Algorithms

Preemptive scheduling is used when a process switches from running state or waiting state to ready state. CPU is allocated for the limited time and the process is placed back in the ready queue if not completed. Preemptive Scheduling incurs the cost of maintaining the integrity of shared data.

First Come First Serve

- The process that requests the CPU first is allocated the CPU first. It is implemented by using the FIFO queue. FCFS is a **non-preemptive** scheduling algorithm. It has **high average waiting time**.
- This can lead to **convoy effect**. Suppose there is a CPU-intensive process and several others with less burst time but are I/O bound (need I/O frequently). While CPU intensive process is being executed, the I/O bound processes complete I/O and wait in the ready queue leading to **I/O devices being idle**. All these processes enter the device queue where the CPU intensive process makes other processes wait again leading to **CPU being idle**.

Shortest Job First

- Process with the shortest burst time is scheduled first. FCFS is used to break a tie. It has **optimal average waiting time**. Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling.
- With short-term scheduling, there is no way to know the length of the next CPU burst. We have to predict processes; can use **exponential average** of previous burst time or a static feature like process type or process size.
- A preemptive SJF(**shortest-remaining-time-first scheduling**) algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Priority Scheduling

- Processes are scheduled according to their priorities. FCFS is a **non-preemptive** scheduling algorithm.
- **Aging** is a technique of gradually increasing the priority of processes that wait in the system for a long time to deal with starvation.

Round Robin Scheduling

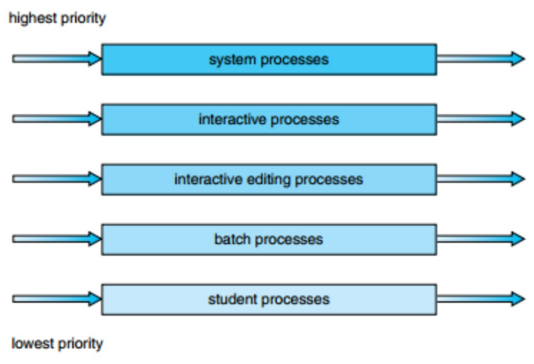
- Each process is assigned a fixed time quantum in a cyclic way. It is designed especially for the **time-sharing system**. To implement Round Robin scheduling, we maintain a FIFO queue of processes. The CPU scheduler picks the first process and sets a timer to interrupt after 1-time quantum, and dispatches the process at the expiry of the timer and the next process is executed. A context switch will be executed if the process has not completed yet, and it will be put at the tail of the ready queue.
- If the time quantum is extremely large, the RR policy is the same as the FCFS policy. If the it is extremely small, the RR approach is called **processor sharing** and (in theory) creates the appearance that each of n processes has its own processor running at 1/n the speed of the real processor. A rule of thumb is that **80%** of the CPU bursts should be shorter than the time quantum.

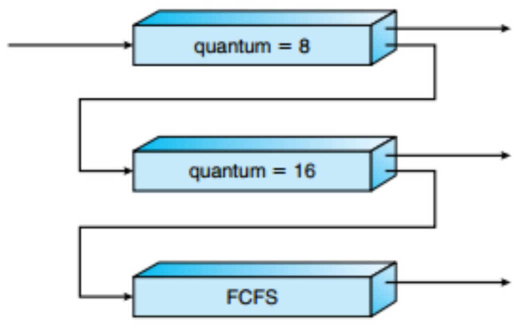
Multilevel Queue Scheduling

- According to the priority of process, processes are permanently assigned to a queue when they enter the system. Generally high priority process are placed in the top level queue. Only after completion of processes from top level queue, lower level queued processes are scheduled. It can suffer from starvation. A RR approach could be used to allocate CPU time to the queues.

Multilevel Feedback Queue Scheduling

- This allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.





4 Thread Scheduling

It is the kernel-level threads, not processes, that are scheduled by the OS. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP) instead.

- On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP, a scheme known as **process-contention scope** (PCS), since competition for the CPU takes place among threads belonging to the same process.
- To decide which kernel thread to schedule onto a CPU, the kernel uses **system-contention scope** (SCS). Competition for the CPU with SCS scheduling takes place among all threads in the system.
- Systems using the one to-one model, such as Windows, Solaris, and Linux, schedule threads using only SCS. Typically, PCS is done according to priority which are set by the programmer. Typically, this is preemptive in nature.

5 Multiple-Processor Scheduling

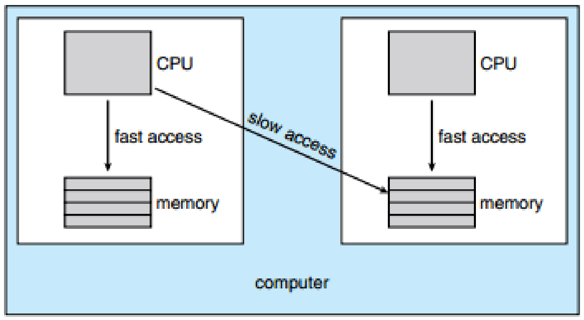
- **Asymmetric multiprocessing** has a "master" processor that handles all schedules decisions, I/O processing and other system activities. Only one processor accesses system data structure reducing the need for data sharing.
- In **Symmetric multiprocessing**, each processor is self-scheduling. All processors may have a common ready queue or a private queue for each processor.

Processor Affinity

Migration of process from one processor to another incurs the high cost of invalidating and repopulating caches. Thus a process has an affinity for the processor it is running on.

When OS does not guarantee it but attempts to keep running a process on the same processor, it is **soft affinity**. If it is not allowed to migrate, it is **hard affinity**. Solaris allows processes to be assigned to **processor sets**.

The main-memory architecture of a system can affect processor affinity issues. In non-uniform memory access (NUMA), a CPU has faster access to some parts of main memory than to other parts. If CPU Scheduler and memory placement algorithms work together, a process with an affinity to a processor can be assigned affinity to the memory than it can access quickly.



Load Balancing

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. It is trivial if there is a common run queue.

- **Push Migration**: A specific task periodically checks the load on each processor and distributes the load by pushing processes from overloaded to less-busy processors.
- **Pull Migration**: An idle processor pulls a waiting task from a busy processor.

Linux runs push migration every 200 ms and pull migration whenever a processor is idle. Load balancing invalidates the benefit of Processor Affinity.

Multi-core Processors

Memory stall causes processor to wait for up to 50% of its time for data to become available. In multi-core processors, multiple hardware threads are assigned to each core in multi-threaded processor cores. The core can switch to another thread if a thread stalls.

- With **coarse-grained multi-threading**, a thread executes until a long-latency event (memory stall) occurs. The cost of switching between threads is high, as the instruction pipeline must be flushed before the other thread can begin execution on the processor core. Once this new thread begins execution, it begins filling the pipeline with its instructions.
- **Fine-grained (or interleaved) multi-threading** switches between threads at a much finer level of granularity—typically at the boundary of an instruction cycle. The architectural design of fine-grained systems includes logic for thread switching. As a result, the cost of switching between threads is small.