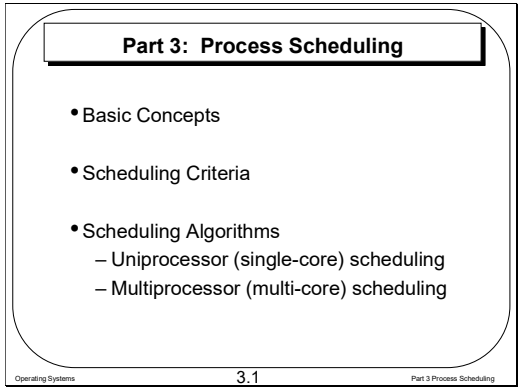
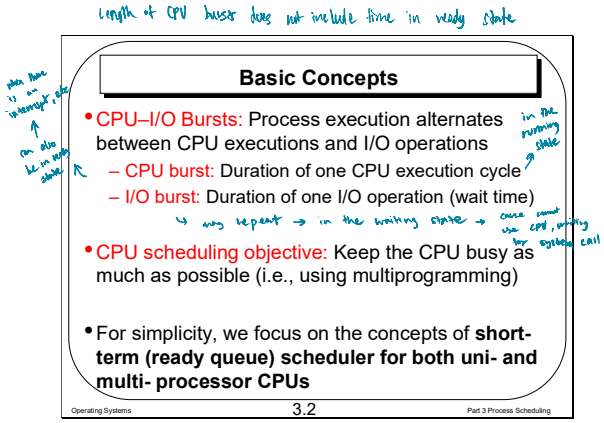
Slide 1



Review of last lecture:
We start with talking about single processs operations, including process creation and termination.
You can use some system calls to create/terminate a process.
Then, we talk about IPC, inter-process communication.
We have two IPC models: shared memory and message passing. Message passing is more widely used.
Then, we talk about two message passing methods: direct communication and indirect communication.
Recall they are message passing, not shared memory.

Multithreading not included in this part of mod

Slide 2



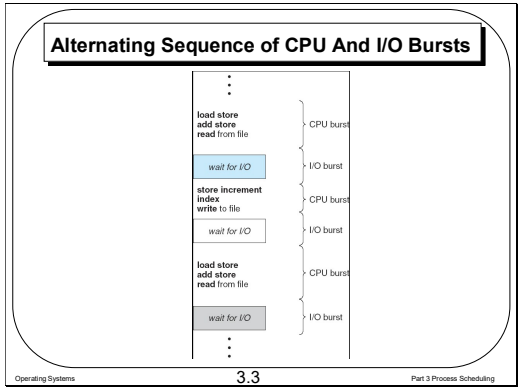
When a process has to wait for I/O or other events, the operating system performs a context switch, and gives the CPU core to another process. That means, a normal process execution has CPU execution, then I/O and then CPU execution, and so on. CPU execution and I/O wait form a cycle.

Why do we need to learn about CPU-I/O burst cycle? Single process cannot fully utilize the CPU. When a process is in an I/O burst, we need to seek another process to execute on the CPU. That is multiprogramming.

Maximum CPU utilization is obtained with multiprogramming. Multiprogramming means that there are multiple programs kept in the memory at one time and CPU is multiplexed among the processes. CPU scheduling or process scheduling is to keep the CPU busy. If the CPU comprises more than one processing core (multiprocessor system), then the scheduler needs to select multiple processes to execute in parallel, one per core.

For simplicity, we focus on the concepts of short-term, ready queue, scheduler. The scheduling algorithms for long-term scheduler and medium-term scheduler will be discussed in the second half of the course.

Slide 3

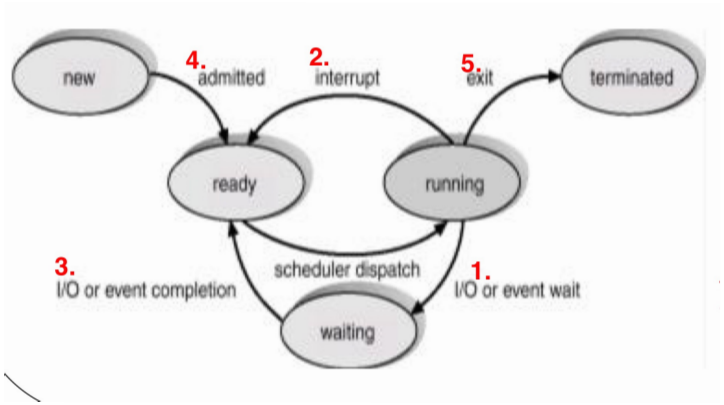


Let's look at one example of alternativing sequence of CPU and IO bursts. As you can see from this figure, the process execution usually starts with a CPU burst, and then followed by an IO burst by reading from file. Then, CPU and IO bursts appear alternatively, and they form a cycle.

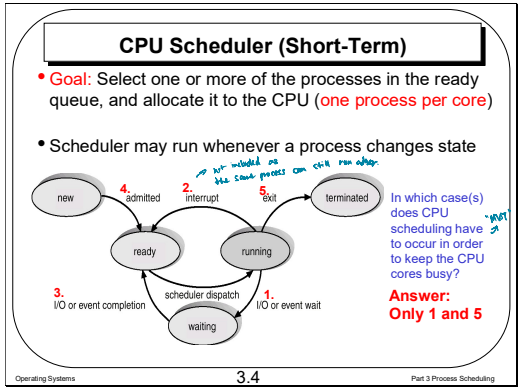
Turnaround time : Transition 4 to 5
→ loops through 1, 2, 3 many times

waiting time: time spent in running state

response time : Transition 4 to first time going into running state



Slide 4



When one or more of the CPU cores become idle, the CPU scheduler is responsible for selecting processes from the ready queue and allocating them to the cores (one per core).

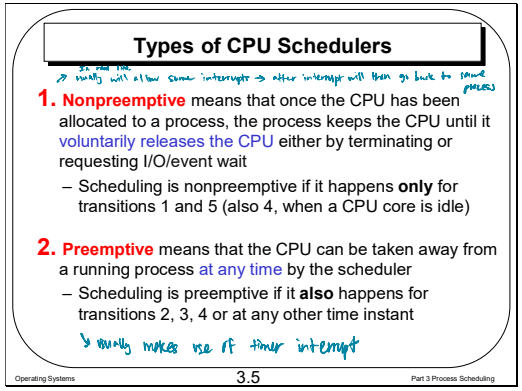
CPU scheduling decisions can take place when a process changes the state. Let's see the process state transitions. There are five transitions when a scheduling decision may be made. **Scheduler dispatch** is the decision made by the CPU scheduler.

For which cases does scheduling have to occur, in order to keep the CPU busy? When I say "have to occur", I mean: if scheduling does not take place, one or more CPU cores will become idle.

Let's look at Case 4: admitted. In this case, we simply add a process to ready queue. If there are running processes they can continue to run; the newly admitted process can run if some core is idle. Hence scheduling is not always required to keep the CPU busy.

You can see that for the first and the fifth cases, we have no choice but to execute the short-term scheduler in order to keep the CPU cores busy. This is because, the active process on a CPU core cannot execute anymore.

Slide 5

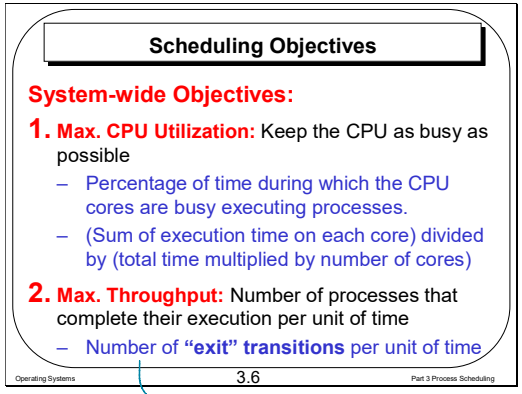


Based on when a scheduler chooses a new process, we can categorize the scheduler to be nonpreemptive or preemptive.

A scheduler is nonpreemptive, if once the CPU has been allocated to a process, the process keeps the CPU until it voluntarily releases it either by terminating or requesting I/O/event wait. That means, scheduling is nonpreemptive if it happens only for transitions 1 and 5 (also 4, when a CPU core is idle).

A scheduler is preemptive if the CPU can be taken away from the running process at any time by the OS. Scheduling is preemptive if it also happens for transitions 2, 3, 4 or at any other time. We should pay attention to “any time” here. Any time includes all the five cases and also any other time instant, depending on the scheduling algorithm. As we will see later, preemptive scheduling considers newly arriving processes.

Slide 6

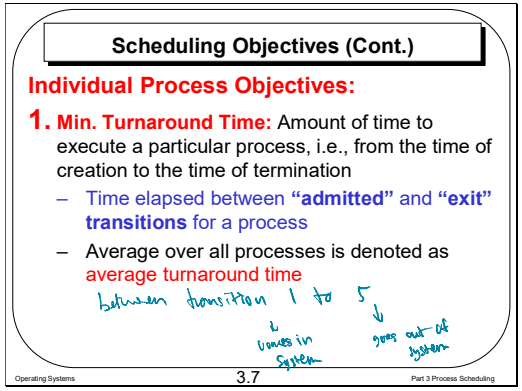


In operating system, there are multiple scheduling criteria. In this lecture, we introduce five of them. Two are system-wide criteria, whereas three are process specific criteria.

The first system-wide criteria is CPU utilization. The goal is to keep the CPU as busy as possible. It is the ratio of CPU core execution time over total time (can also be denoted as a percentage of total time) divided by the number of CPU cores.

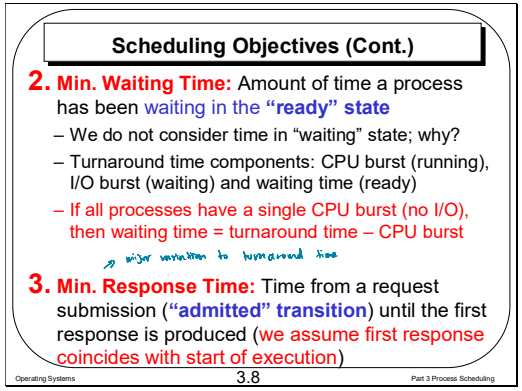
Throughput refers to the number of processes that complete their execution per time unit. For example, the number of processes per second.

Slide 7



The third criteria is turnaround time, which is a process specific criteria. It denotes the amount of time to execute a particular process, from the time of creation to the time of termination.

Slide 8



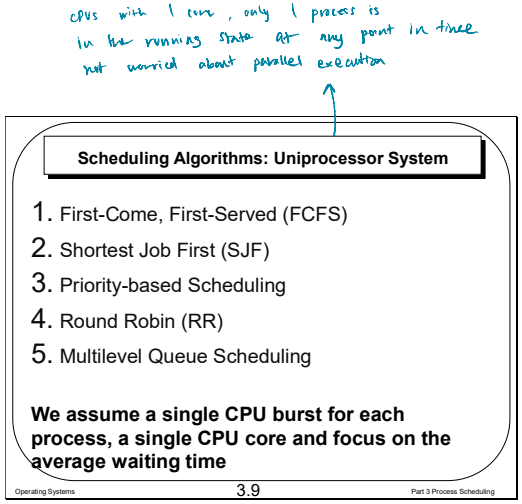
Waiting time is the amount of time a process has been waiting in the ready queue. Please note, CPU scheduling algorithm does not affect the total amount of time during which a process executes or does I/O (waits in waiting state). It only affects the waiting time in the ready state.

Consider the total turnaround time. There are three parts: CPU execution time (that is the CPU burst length), I/O burst length (in the waiting state), and waiting time (in the ready state). See the process state transition diagram.
If all processes have a single CPU burst, then waiting time = turnaround time – CPU burst time. Why? Since a process has a single CPU burst, that means, the process has only two cases: either executing on the CPU or in the ready queue.

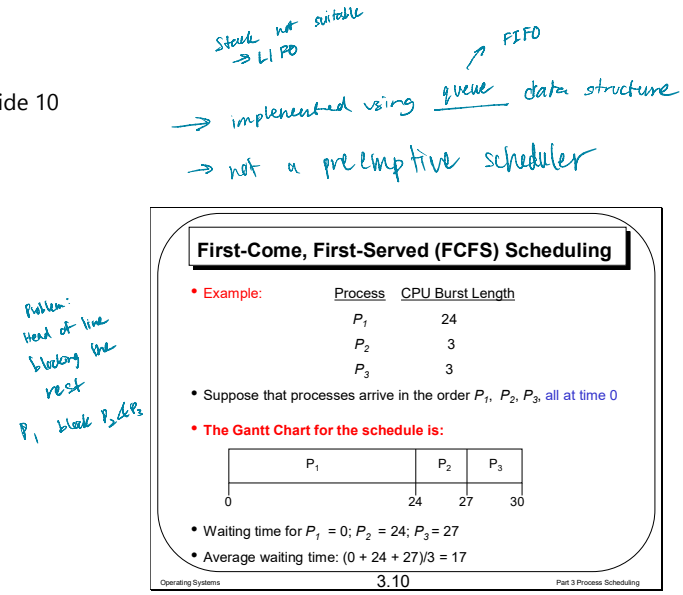
Response time is the amount of time it takes from when a request was submitted until the first response is produced. Please note, a process can produce some output fairly early, and response time is no larger than the turnaround time. Consider one example. When you run a command on a console, what is the time for getting the first output message? Is it response time or turnaround time? It is the response time. What is the time for getting the last output message? It is the turnaround time.

In the remainder of this lecture, we assume that each process has only one CPU burst for simplicity. This can be extended to multiple bursts easily.
Our measure is the average waiting time. In the tutorial, you will have some questions on multiple CPU bursts as well as on calculating the other measures

Slide 9



Slide 10



The simplest algorithm is first come first served, FCFS algorithm. The process that comes first is scheduled on the CPU core first. This algorithm can be implemented using a FIFO queue.

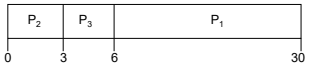
Here is one example. Suppose these three processes arrive at time 0. We use a Gantt chart to represent the schedule. *In the exam, I encourage you to draw the Gantt chart to help your calculation.*

Note, once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. So is FCFS preemptive or nonpreemptive? Nonpreemptive.

Slide 11

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order P_2, P_3, P_1, again all at time 0
- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = **3**
  - Much better than the previous case

The average waiting time of FCFS algorithm is affected by the arrival order. Let's consider another arrival order for the same example.

The average waiting time is much better than the previous case.

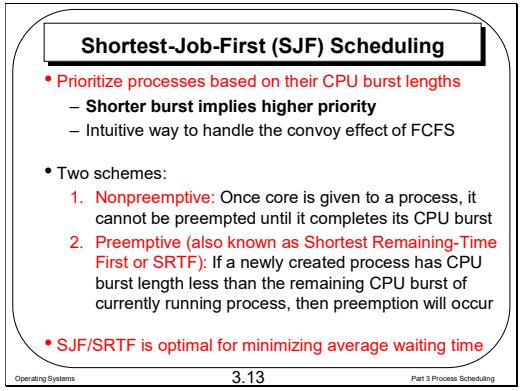
Slide 12

FCFS Scheduling: Properties

- Is FCFS preemptive or nonpreemptive?
  - Nonpreemptive, because processes have to voluntarily release the CPU once allocated
- Main problem with FCFS is the ***convoy effect***
  - Short processes suffer increased waiting times due to an earlier arrived long process
    - *We* saw this effect in the previous example - P1 increased the waiting times of P2 and P3

Different arrival order causes different waiting times. This reveals the problem of FCFS. It is called the convoy effect.
In FCFS, the job arrival order matters!

Slide 13



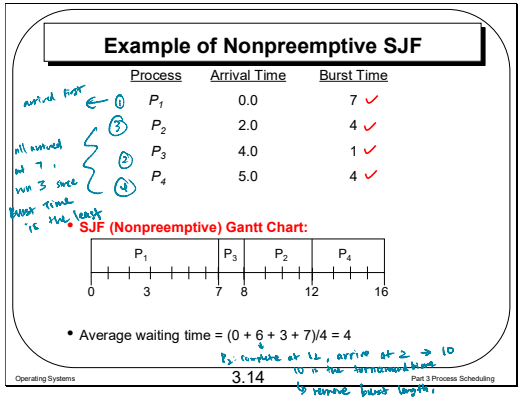
The convoy effect of FCFS problem shows that the short job should have a higher priority. Thus, we have the shortest job first scheduling. Priority is given to the job with the shortest time.

Depending on whether the scheduling is preemptive or not, we have two variants of SJF algorithm:

- Nonpreemptive: Once the CPU is given to the process, it cannot be preempted until the CPU burst is completed.
- Preemptive: If a new process arrives with CPU burst length less than the remaining time of the current executing process, then preempt. That means, the CPU should be given to the new process.

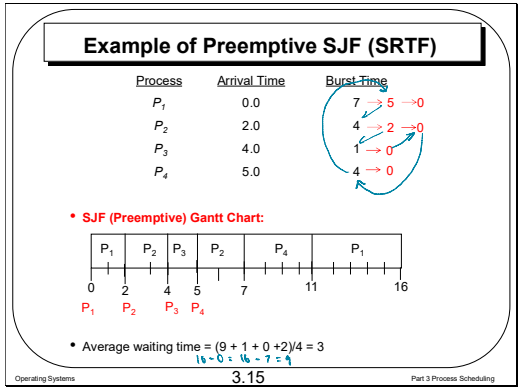
SJF is optimal, in the sense that it gives the minimum average waiting time for a given set of processes. The proof can be found in the textbook.

Slide 14



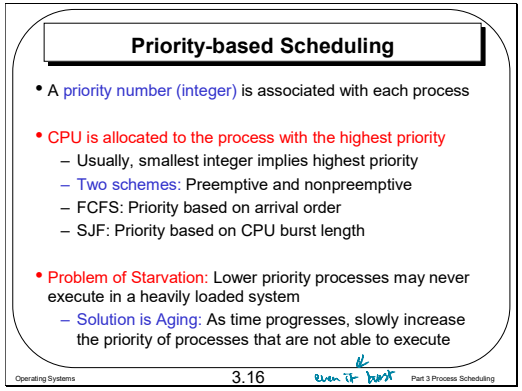
P2 and P4 have the same CPU burst time. How to break the tie? We choose the one with the earliest arrival time when two processes have the same CPU burst length. However, note tie can broken using any strategy. The optimality of SJF w.r.t average waiting time will still hold, irrespective of the tie breaking strategy.

Slide 15



We need to consider scheduling when a new process comes, or when a process terminates.

Slide 16



In FCFS, a higher priority is given to a process that arrives earlier. In SJF, a higher prirority is given to a process with shorter CPU burst length. In general, we can assign an arbitrary priority to each process and schedule based on that priority. This is what happens in priority-based scheduling.

The priority number can be an integer. For example, in Linux, the priority number can be [-20, 20]. -20 means the highest priority; 20 means the lowest priority. Similar to SJF, priority scheduling can be preemptive or nonpreemptive.

One of the problems with priority-based scheduling is that of starvation. That is, lower priority processes may never execute. In a heavily loaded system, a low priority process may never get the CPU. In UC Berkeley, they found a process that had starved for over 30 years.

A common solution for avoiding starvation is to introduce aging. The basic idea of aging is that, as time progresses, the system gradually increases the priority of the process. Eventually, the process has a high priority to execute.

Slide 17

Round Robin (RR) Scheduling

- Fixed time quantum for scheduling (q): A Process is allocated CPU for q time units, preempted thereafter and inserted at the end of the ready queue
  - Processes are scheduled cyclically based on q
  - Usually q is small, around 10-100 milliseconds
- Performance:
  - *n processes in ready queue implies* waiting time is no more than (n-1)q time units
  - **Large q:** Degenerates to FCFS
  - **Small q:** *Many context switches; high overhead*

Now we look at a fair scheduling algorithm. This algorithm is called round robin. In RR, each process gets a small unit of CPU time (called time quantum). The time quantum is usually 10-100 ms. After this time has elapsed, the process is preempted and added to the tail of the ready queue. We give the CPU to the process that is at the head of the ready queue. This is the timer interrupt in the process state transition diagram.

Question: is RR preemptive or nonpreemptive? Preemptive. The CPU is forcibly taken away from the process when the time quantum is finished. We actually use RR to implement the scheduling of time sharing systems.

We can also see, if the time quantum q is very large, say, larger than the maximum length of the CPU bursts, RR becomes FCFS. When a process comes, it will be added to the end of the ready queue. Each process can finish the execution in a time quantum.

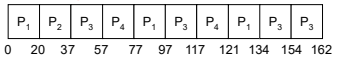
If q is too small, context switch overhead is very high. So, q must be large with respect to the context switch overhead. There will be a tutorial question on this issue.

Slide 18

Example: RR with Time Quantum = 20

Process	Burst Time
P_1	53 → 33
P_2	17 → 0
P_3	68 → 48
P_4	24 → 4

- All processes come at time 0 in the order of P1, P2, P3, P4
- **RR scheduling Gantt Chart:**

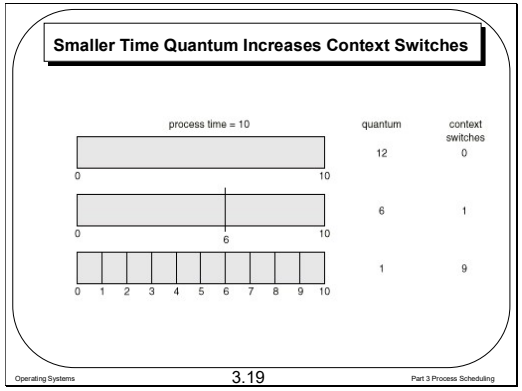


- Typically, higher average waiting/turnaround time than SJF, but better *response time*

RR has a higher average turnaround time than SJF, but better response time.

What is the response time for P1? 0 P2? 20....

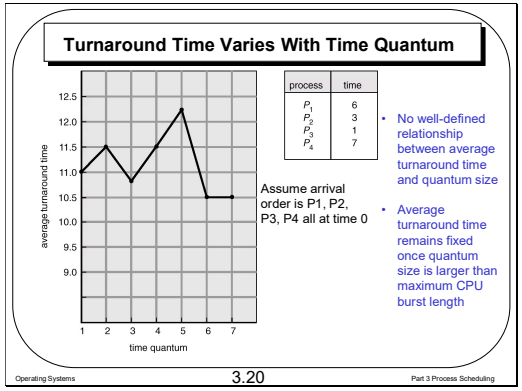
Slide 19



So, now let's see the effect of the time quantum size on the number of context switches.

A process with a CPU burst of ten units of time. If the quantum is 12 units of time, the number of context switches *during* the process execution is zero. If the quantum is 6, the number of context switches during the process execution is 1. If the quantum becomes 1, the number of context switches is 9. So you can see, a small time quantum will dramatically increase the context switch overhead. There is a similar question in the tutorial.

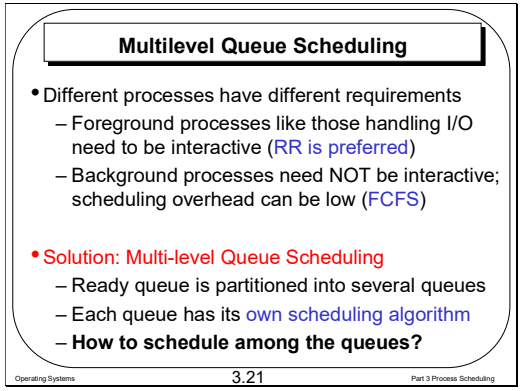
Slide 20



FCFS = (6+9+10+17)/4=10.5, assuming arrival order of P1, P2, P3 and P4.

In this example we see how the time quantum size affects the turnaround time. We can make two observations. First, there is no clear relationship between the average turnaround time and the time quantum size. As the time quantum size increases, the average turnaround time may not always increase. Second, if the time quantum is larger than the largest CPU burst length, the average turnaround time becomes stable. Why? Because RR becomes FCFS.

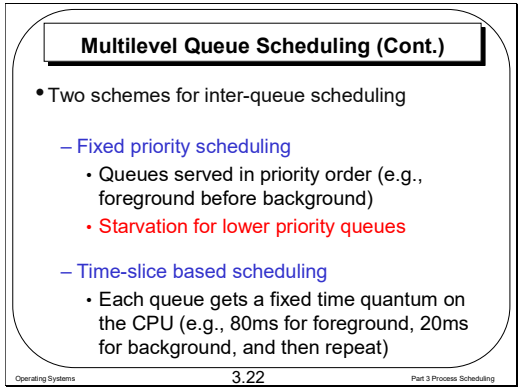
Slide 21



So far, we focused on a single ready queue. In practice, different processes have different requirements on performance, and thus they may require different scheduling algorithms. For example, foreground processes are more interactive like the key stroke, mouse or when you are playing a game. We prefer RR for such processes. Background processes are more about batch processing such as virus scanning. We prefer FCFS for such processes. Moreover, foreground processes may have a higher priority than background processes.

Multi-level queue scheduling can be used to handle such varying requirements. The ready queue is partitioned into separate queues, and each queue can have its own scheduling algorithm.

Slide 22



We call it multi-level queue, because the scheduling is not only performed within each queue, but also among the queues.

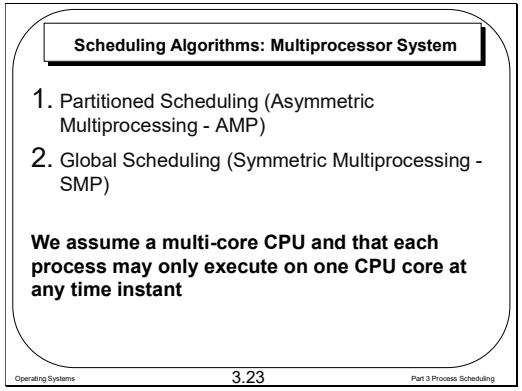
There are basically two ways of inter-queue scheduling.

The first one is fixed priority scheduling: we first serve all the processes in the foreground queue. After serving all the processes in the foreground queue, we consider the background queue. That may cause starvation for the background process. So, how to solve this problem? Aging.

The other method is time slice. Each queue gets a certain amount of CPU time which it can use to schedule among the processes. For example, 80% of the CPU time is given to the foreground queue, and the other 20% is given to the background queue. In each queue, we perform scheduling according to the chosen algorithm for that queue.

Slide 23

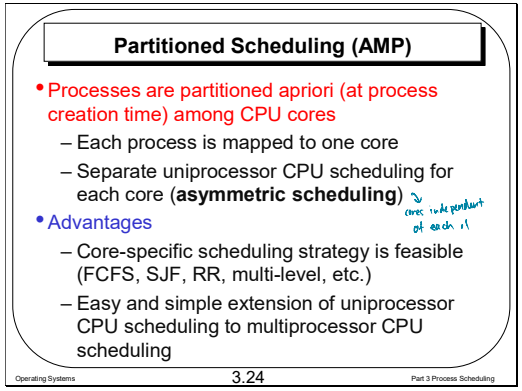
→ each CPU has 1 CPU burst
→ cannot run same process in parallel (in multithreading you can)
→ a process cannot run in 2 cores at the same time (only at different times)



We now look at the multiprocessor scheduling problem in which processes need to be allocated to two or more CPU cores. We will look at two types of scheduling strategies, partitioned and global scheduling.

For this section, we assume that a single process cannot execute in parallel on two CPU cores at the same time. However, it is possible that it executes on different CPU cores at different times. For example, it may start executing on core 1 and at a later time be context switched to execute on core 2.

Slide 24

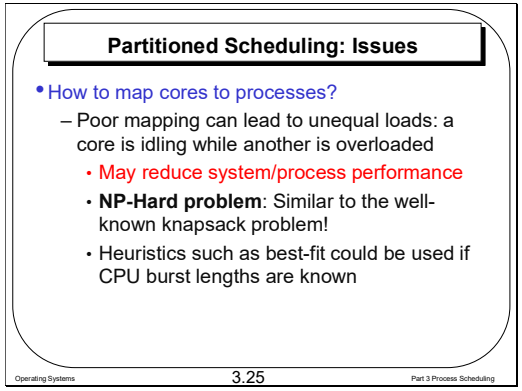


This is the simplest mechanism to extend single-core CPU scheduling to multi-core systems.

There are one or more dedicated ready queues per CPU core (queues are specific to cores), and different uniprocessor scheduling algorithms can be used for different cores. Mapping of processes to cores is done at process creation time.

What would be the main challenge in such a simple extension of uniprocessor scheduling to multiprocessors?

Slide 25

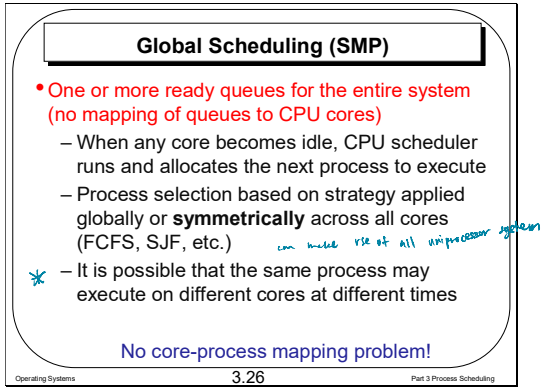


Allocating processes to cores is a challenging problem. Ideally, some sort of load balancing strategy must be employed so that all cores are equally utilized. This will improve overall system and process performance (turnaround time, throughput, etc.). An unbalanced distribution may lead to scenarios in which one core is idling while the other core is overloaded with several processes waiting/ready to execute. This can worsen the performance metrics on the overloaded core.

However, optimal allocation of processes to core is not feasible, unless P=NP. This is a NP-Hard problem very similar to the well-known knapsack problem, in which items with weights (processes with CPU bursts) need to be allocated to a set of knapsacks of a certain fixed capacity (cores operating at a fixed frequency).

Heuristics could be used to perform this allocation provided CPU burst lengths are known in advance. For example, best-fit could try to allocate a process to a core that is least backlogged (total pending execution time on the core) among all the cores.

Slide 26

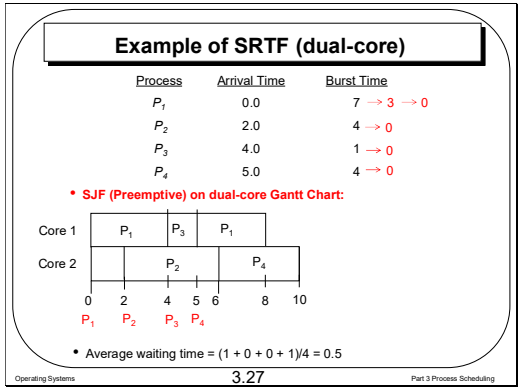


A global set of ready queues are maintained. Queues are not assigned to specific CPU cores. When a core becomes idle, a process is selected from one of the queues and allocated to the core. The selection strategy is applied globally across all the cores.

For example, if we use FCFS on a dual-core system, then there will be a single global ready queue. At any point in time, the two earliest arrived processes that are ready to execute will be executing on the two CPU cores.

Since processes could have multiple CPU bursts or they could be preempted, it is possible that the same process is executing on different CPU cores at different times. However, our assumption is that the same process cannot be executing on two different CPU cores at the same time.

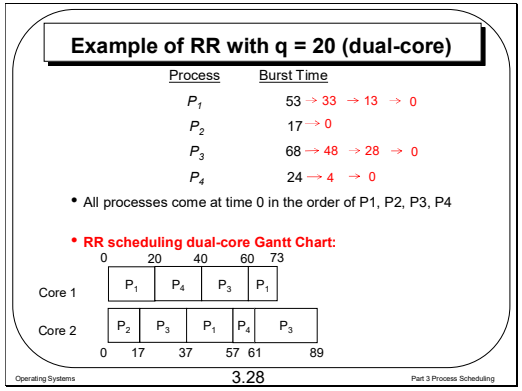
Slide 27



Preemptive shortest job first applied to the case when we have two CPU cores. The example is the same as the one used for uniprocessor scheduling.

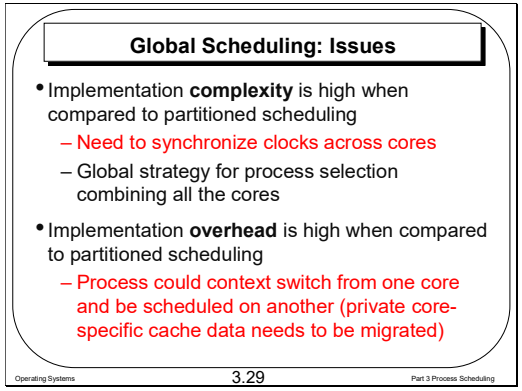
Compare the waiting times obtained here with the ones for the uniprocessor system.

Slide 28



Round robin scheduling applied to the case when we have two CPU cores. The example is the same as the one used for uniprocessor scheduling.

Compare the waiting times obtained here with the ones for the uniprocessor system.



The scheduler must use a single clock so that the cores can be synchronized. Notion of time is global across all cores.

Selection of next process to run may involve going through several processes in the ready queue, particularly when there are more than one idle cores.

When a process context switches from one core to another, it loses access to the core's private cache. This means that data must now be migrated to the other core's private cache, which can be time consuming. Or the process has to start with a cold private cache in the other core, which can significantly increase its execution time. Either way, migration has overhead, which is not applicable in partitioned scheduling.