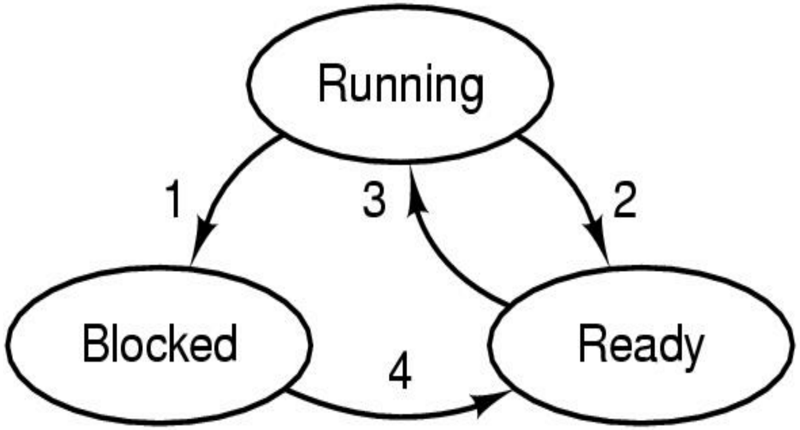
CPU Scheduling

Section 2.4: Tanenbaum’s book
Chapter 5: Silberschatz’s book

Kartik Gopalan

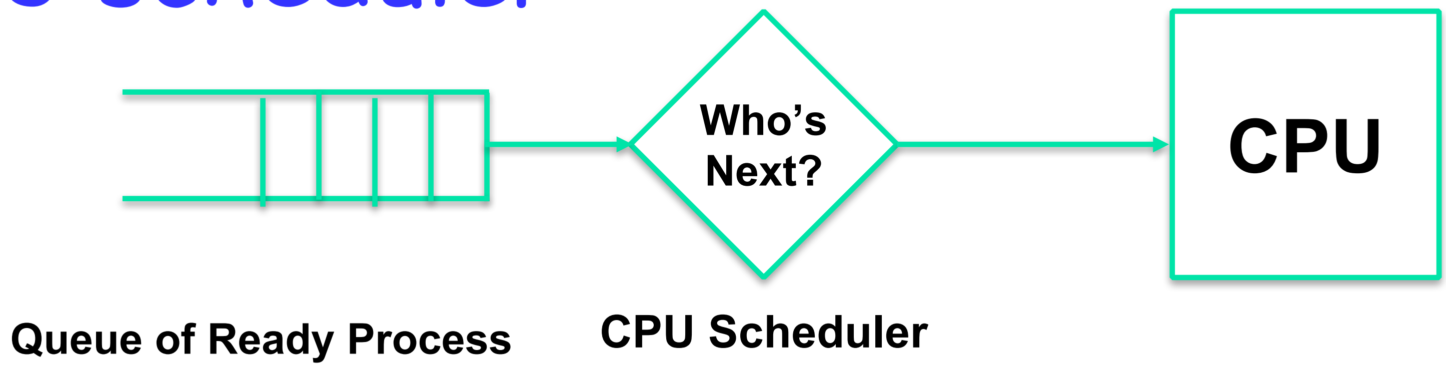
Process Lifecycle Revisited



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Ready
  - Process is ready to execute, but not yet executing
  - Its waiting in the scheduling queue for the CPU scheduler to pick it up.
- Running
  - Process is executing on the CPU
- Blocked
  - Process is waiting (sleeping) for some event to occur.
  - Once the event occurs, process will be woken up, and placed on the scheduling queue.

CPU scheduler



- Selects the next process to run on the CPU from among the processes that are ready to execute

- CPU scheduling decision may take place when any process:
  1. Switches from running to waiting state
  2. Switches from running to ready state (pre-emptive scheduling)
  3. Switches from waiting to ready
  4. Terminates

Dispatcher

- Not the same as scheduler.

- Dispatcher gives control of the CPU to the process selected by the scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- *Dispatch latency*
  - Time it takes for the dispatcher to stop one process and start another running

Linux CPU Scheduling Code

- To browse, go to http://lxr.free-electrons.com

- To download, modify, and compile, go to
  - http://ftp.kernel.org/

- Scheduling code is located under
  - kernel/sched/core.c

- Three ways to invoke kernel code
  - System calls
  - Hardware interrups
  - Exceptions

- For x86, all entry points into the 4.2 kernel are located in assembly code in
  - arch/x86/entry/entry_32.S
  - OR arch/x86/entry/entry_64.S

When does the scheduler execute?

A. Timer interrupt fires OR
B. Some process blocks (or gives up CPU)

- Look at the CPU scheduler code in Linux
    - schedule() function in kernel/sched/core.c
    - http://lxr.free-electrons.com/source/kernel/sched/core.c#L2988
- Entry point to the kernel (in entry_*.S) invokes schedule() function which then
    - Figures out the next process to schedule and
    - Swaps the prev and next process via context_switch() function, which in turn
    - Switches memory state using switch_mm() and
    - Switches register+stack state using switch_to()

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – Number of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process, from submission to termination.
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced.

Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Different systems have different scheduling goals

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

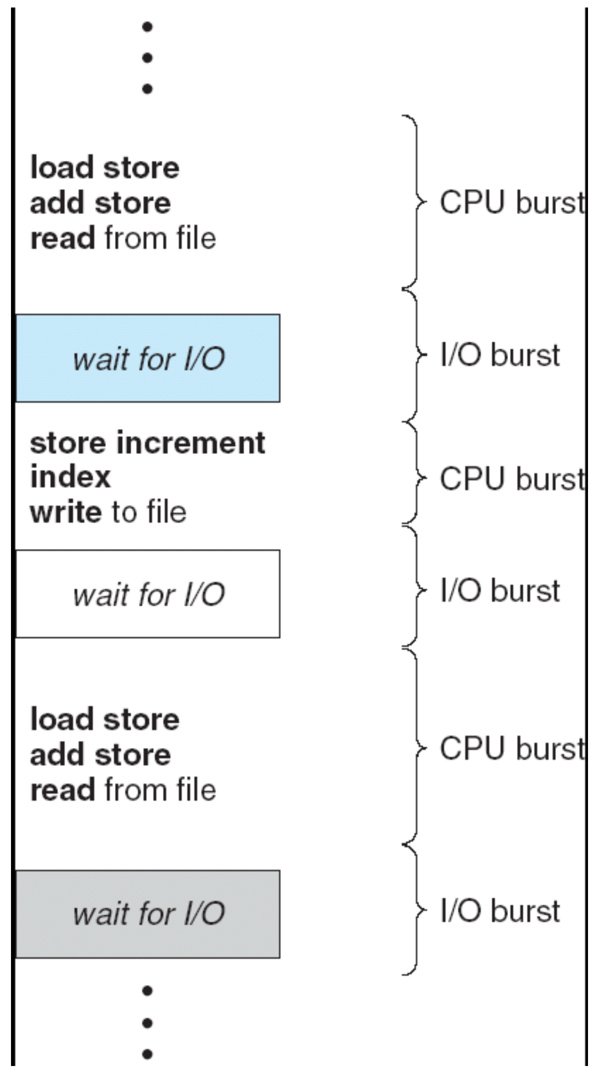
Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

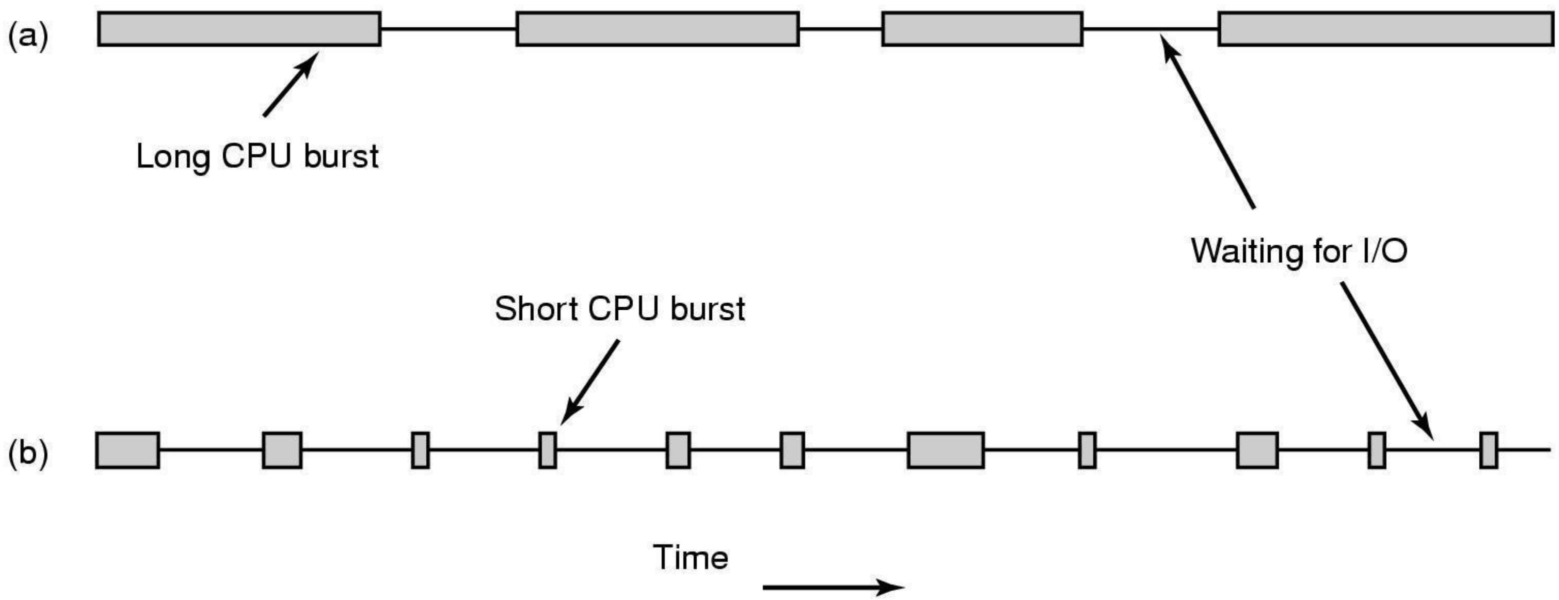
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Alternating Sequence of CPU And I/O Bursts in a Typical Program



Alternating Sequence of CPU And I/O Bursts (contd)

- A CPU-bound process



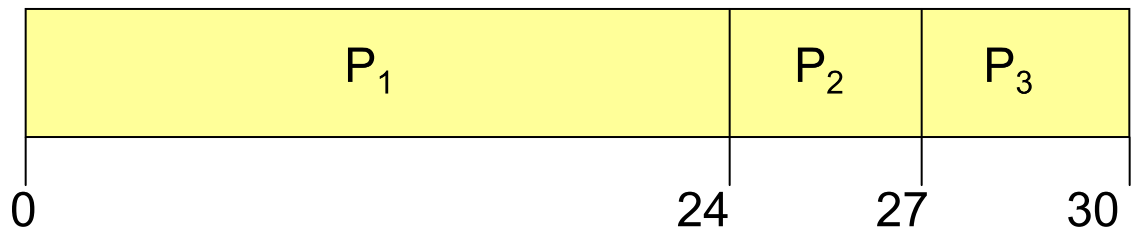
- An I/O-bound process

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

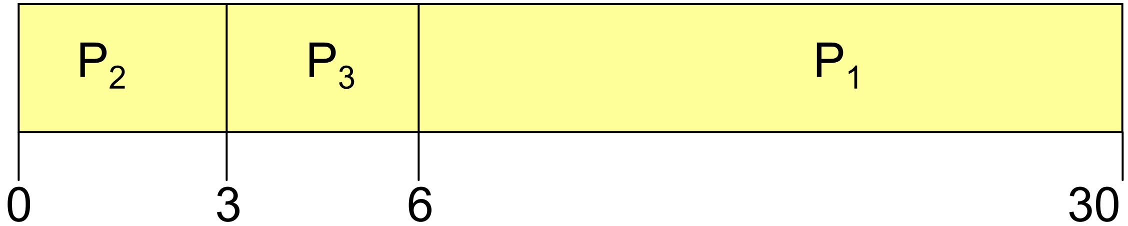


FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- 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 previous case
- *Convoy effect* : short process behind long process

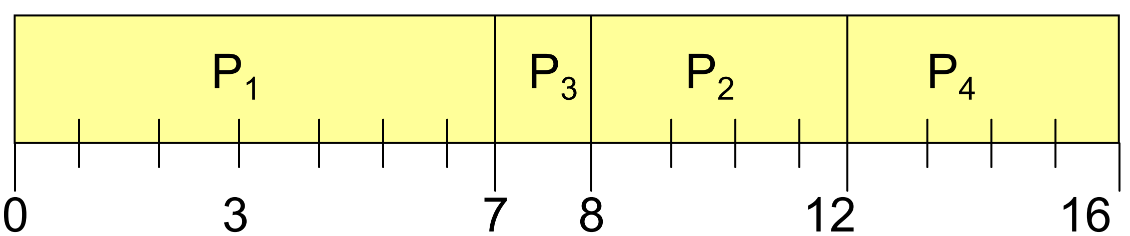
Shortest-Job-First (SJF) Scheduling

- Consider the length of its next CPU burst for each process.
- Schedule the process having the shortest next CPU burst.
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of currently executing process, preempt the current process. This scheme is know as the Shortest Remaining Time First (SRTF)
- Claim: SJF algorithm is optimal
  - SJF Gives a schedule with least average waiting time compared to any possible scheduling algorithm.

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

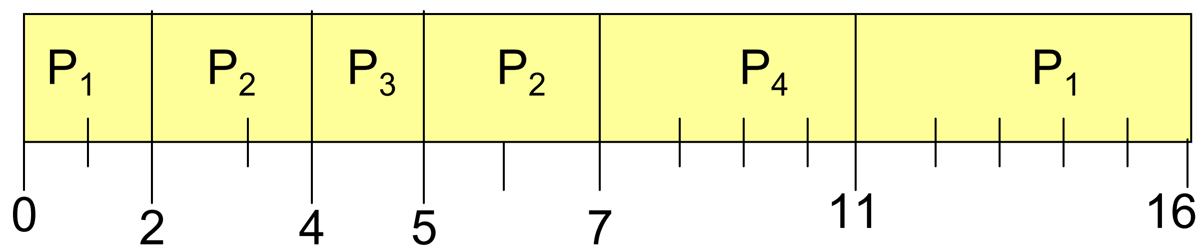


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

SJF is Optimal w.r.t average wait time

- Meaning that no other algorithm can achieve a lower wait time than SJF

  Proof:
- Assume that there was an algorithm X that gave better average wait time than SJF for a set of N processes.
- Since X is not the same as SJF, it means that there must be at least two processes P1 and P2 in the schedule generated by X such that
  a. P1 executes before P2 does and
  b. The CPU execution time of P1 is longer than that of P2 and
  c. The average wait time of the schedule is smaller than that given by SJF
- BUT, is you swap the positions of P1 and P2 in the schedule generated by X, then the average wait time goes down!
- So keep swapping all such process pairs that satisfy conditions (a) and (b) above. Each swap will reduce the average wait time.
- Finally you will end up with a schedule generated by SJF, whose wait time cannot be reduced any further. Hence SJF is Optimal.

Exponential Averaging:
Determining the Length of Next CPU Burst

- Not easy. Can only *guess* the length of next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

t_n = actual length of the n^{th} CPU burst

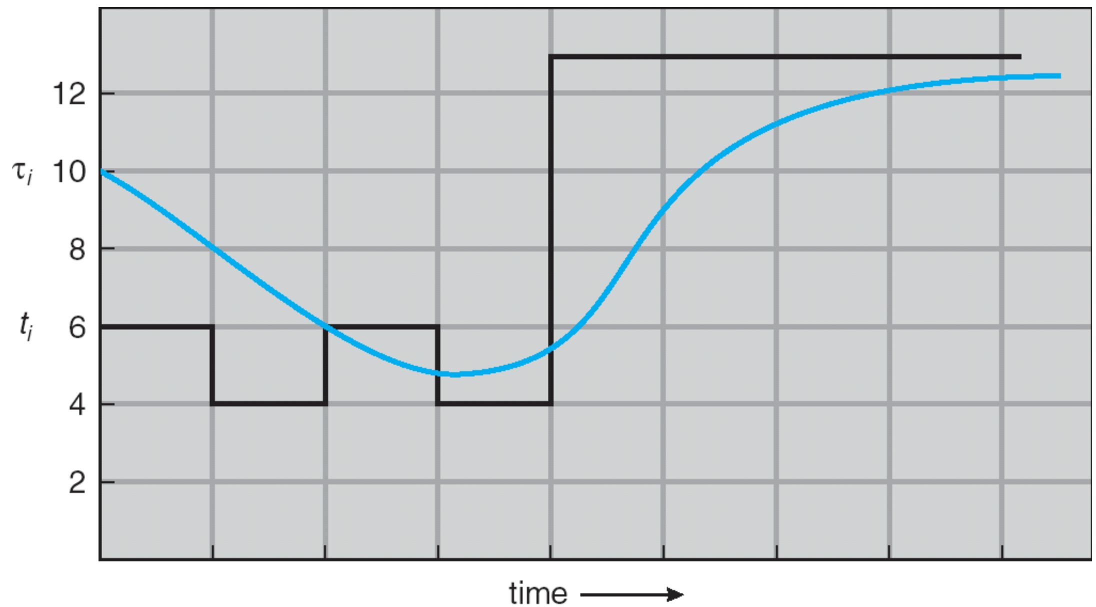
τ_n = predicted value of the n^{th} CPU burst

α, $0 <= \alpha <= 1$

Define:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha) * \tau_n$$

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha) * \tau_n$$

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - CPU burst history does not count

- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts

- If we expand the formula, we get:

$$\begin{aligned}\tau_{n+1} = \alpha\, t_n &+ (1-\alpha)\alpha\, t_{n-1} + \ldots \\ &+ (1-\alpha)^j \alpha\, t_{n-j} + \ldots \\ &+ (1-\alpha)^{n+1} \tau_0\end{aligned}$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Round Robin (RR)

- Each process gets a fixed unit of CPU burst time (*time quantum*)
  - usually 10 to 100 milliseconds.

- After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in bursts of at most q time units at once. No process waits more than $(n-1)q$ time units.

- Performance
  - q large $\Rightarrow$ FIFO, because processes rarely get pre-empted
  - q small $\Rightarrow$ Smaller response times, but too much context switching overhead. q must be large compared to context switch time, otherwise overhead is too high

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

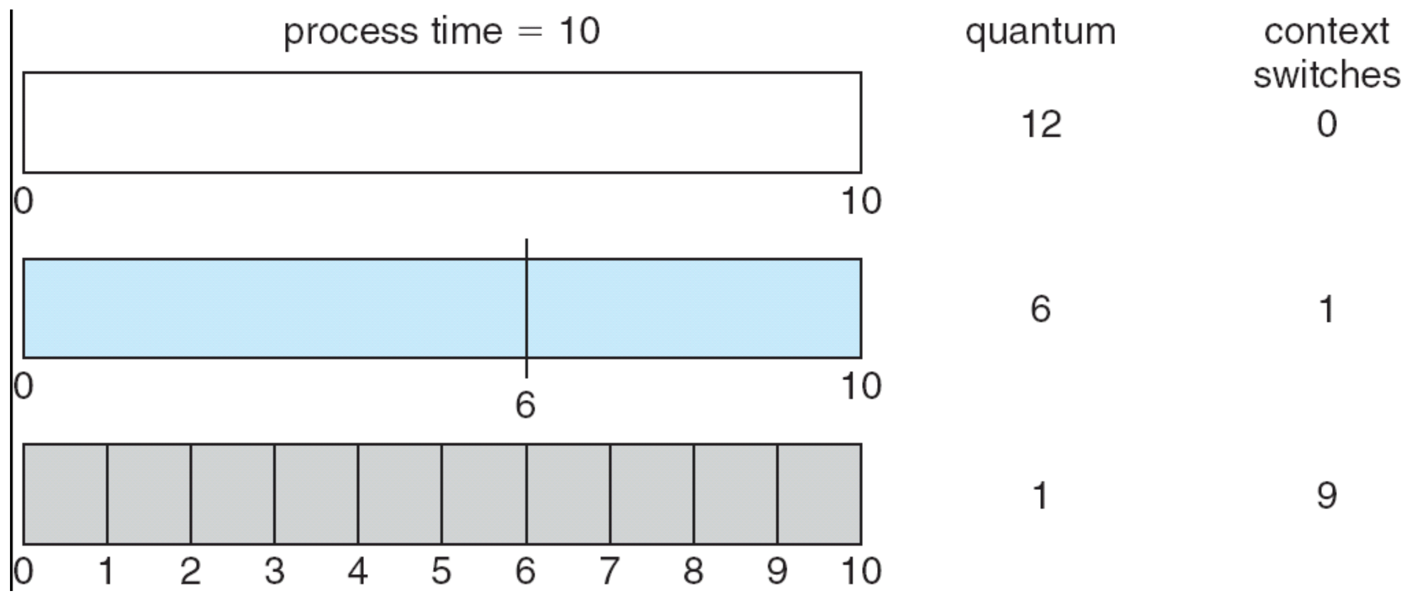
- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but
  - Better *response time*
  - No Starvation

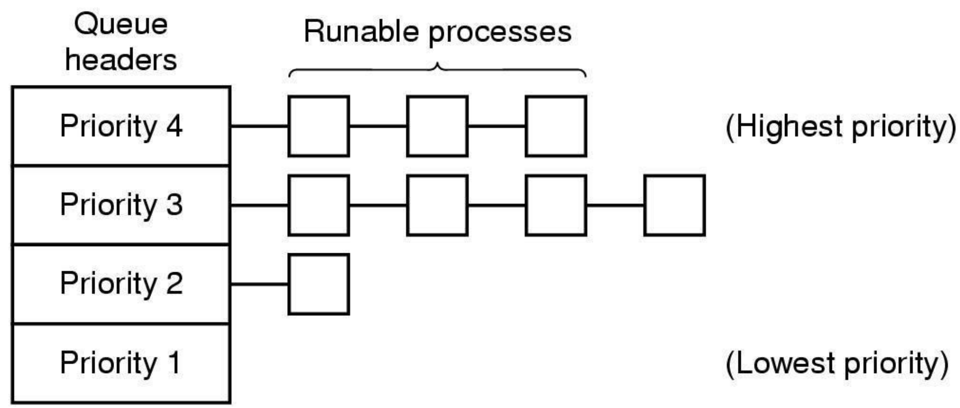
Time Quantum and Context Switch Time



- Smaller the time quantum, more the number of context switches.
- Larger the time quantum, larger the response time.
- Scheduling algorithm needs to find a balance between context switch overhead and response time.

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
- Again two types
  - Preemptive
  - nonpreemptive

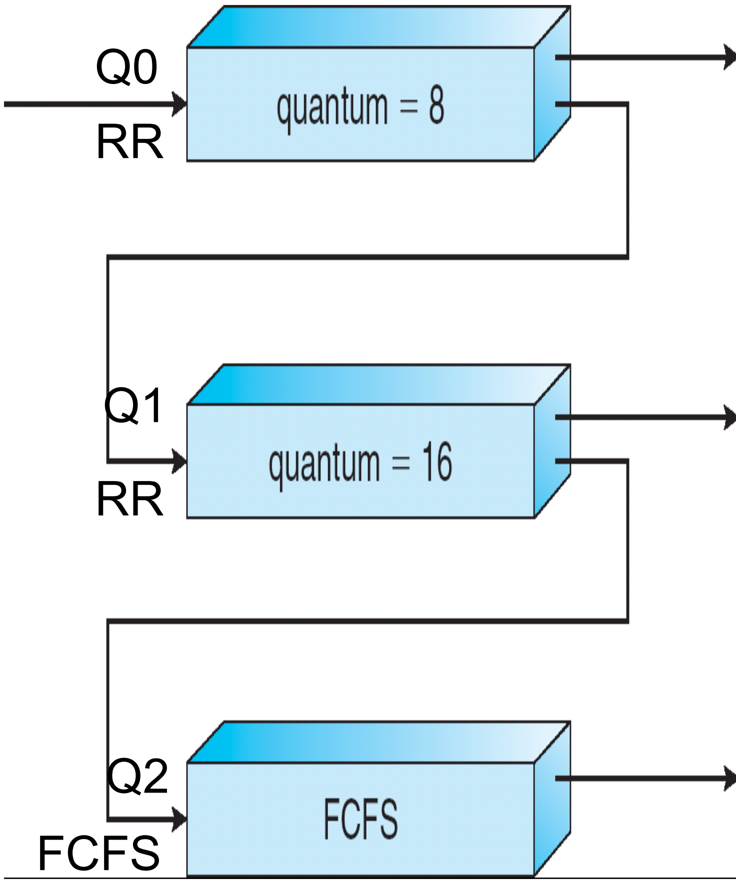


Example with four priority classes

- SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** ➔ low priority processes may never execute
- Solution ≡ **Aging** ➔ as time progresses increase the priority of a lower priority process that is not receiving CPU time.

Multilevel Feedback Queue (MFQ)

- Ready queue is partitioned into separate queues
- Each queue has its own scheduling algorithm
- A process can move between the various queues;
- MFQ scheduler defined by :
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade or demote a process
- Example of MFQ that gives higher priority to interactive jobs
  - A new job enters queue Q_0 which is served RR.
  - When it gains CPU, it receives 8 ms. If it doesn't finish in 8 ms, job is moved to Q_1.
  - At Q_1 job is again served RR; receives 16 ms.
  - If it still doesn't complete, it is preempted and moved to queue Q_2 where it is served FCFS.



Real-Time Scheduling

- *When each task needs to be completed before a given deadline*
- *Hard real-time* systems
  - Required to complete a critical task before its deadline
  - For example, in a flight control system, or nuclear reactor
- *Soft real-time* systems
  - Meeting deadlines desirable, but not essential
  - For example, video or audio
- Schedulability criteria
  - Given *m* periodic events, where event *i* occurs within period P_i and requires C_i computation time each period
  - Then the load can be handled only if

$$\text{sum}_i \, (C_i/P_i) \le 1$$

  - The above condition is *necessary* but *NOT sufficient.*

Fair Scheduling

- Notion of “fairness” does not necessarily mean equal CPU share for all processes.
- Say you have N processes
- Each process P_i is assigned a weight w_i
- The the CPU time will be divided among processes in proportion to their weights.
- Let's say some process does not use its assigned CPU time.
    - The the “spare” CPU time is divided among the remaining ready processed according to the ratio of their weights.

Work-conserving versus non-work-conserving

- Work-conserving scheduler
  - CPU will not remain idle if there are processes in the ready queue

- Non-work-conserving scheduler
  - Under some conditions, scheduler may decide to "waste" CPU time even though there may be processes sitting in the ready queue
  - E.g. Sometimes real-time process cannot be started before a given time (release time).

Linux CPU Scheduling Code

To browse, go to http://lxr.linux.no/

To download, modify, and compile, go to

http://ftp.kernel.org/

Scheduling code is located under

kernel/sched.c

Entry points into the kernel are located at

arch/x86/entry_32.S

OR arch/x86/entry_64.S