

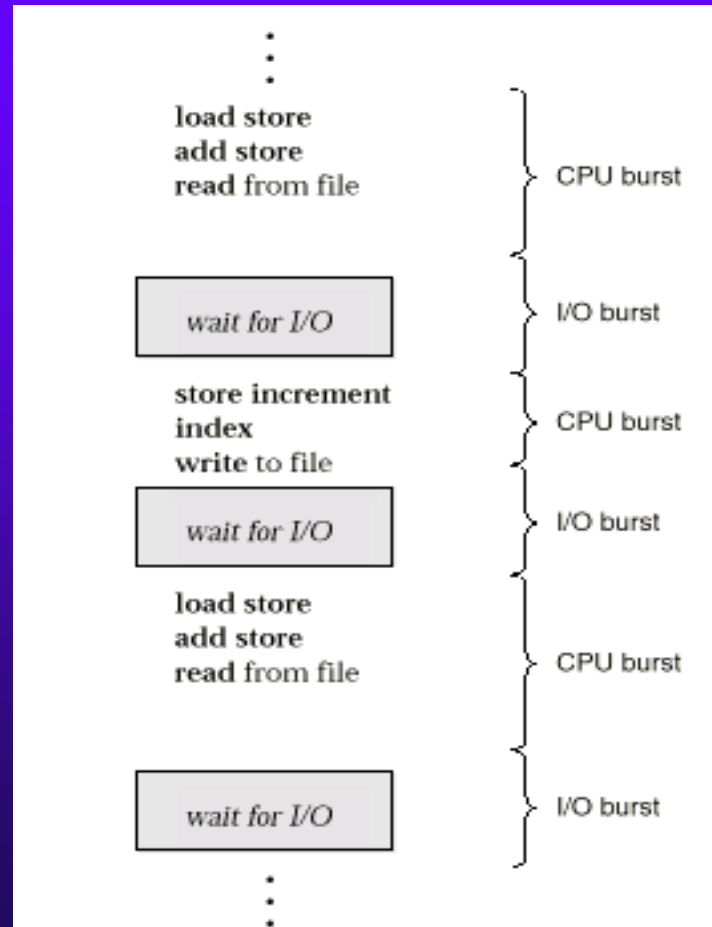


CPU Scheduling

Date

Basic concepts

- ◆ Maximize CPU utilization with multi programming.
- ◆ CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.



CPU Scheduler

- ◆ Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- ◆ CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state.
 2. Switches from running to ready state.
 3. Switches from waiting to ready.
 4. Terminates.
- ◆ Scheduling under 1 and 4 is *non preemptive*.
- ◆ All other scheduling is *preemptive*.
 - It is expensive and need extra hardware.



Dispatcher

- ◆ Dispatcher module gives control of the CPU to the process selected by the short-term 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

Scheduling criteria

- CPU utilization – keep the CPU as busy as possible. CPU utilization vary from 0 to 100. In real s/m it varies from 40 (lightly loaded) to 90 (heavily loaded) s/m.
- Throughput – Number of processes that complete their execution per time unit.
- Turnaround time – amount of time to execute a particular process (interval from time of submission to time of completion of process).
- Waiting time – amount of time a process has been waiting in the ready queue (sum of the periods spend waiting in the ready queue).
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)
- Optimization criteria
 - Max CPU utilization, Max throughput, Min turnaround time, Min waiting time, Min response time

Scheduling Algorithms

◆ First – Come, First – Served Scheduling

– Example: 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:



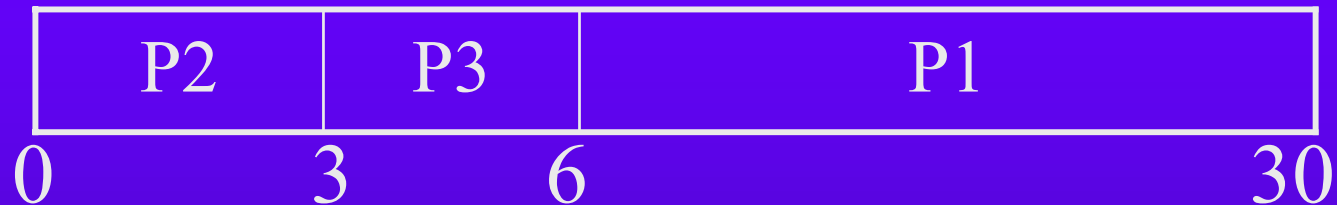
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$



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

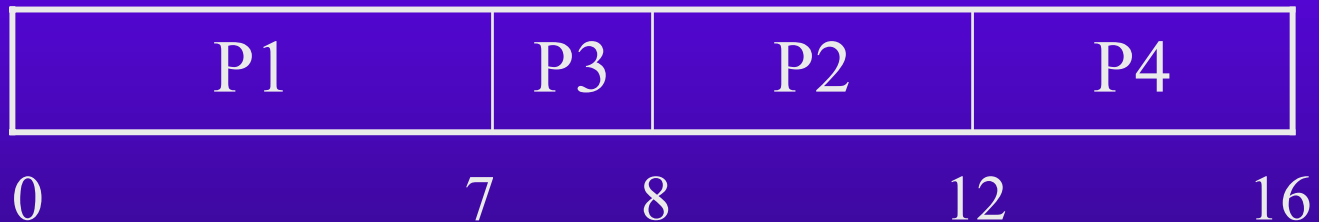
Shortest-Job-First (SJR) Scheduling

- ◆ Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- ◆ Two schemes:
 - Non preemptive – 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 current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- ◆ SJF is optimal – gives minimum average waiting time for a given set of processes.



<u>Process</u>	<u>Arrival Time</u>		<u>Burst Time</u>
P_1	0.0	7	
P_2	2.0	4	
P_3	4.0	1	
P_4	5.0	4	

◆ SJF (non-preemptive)

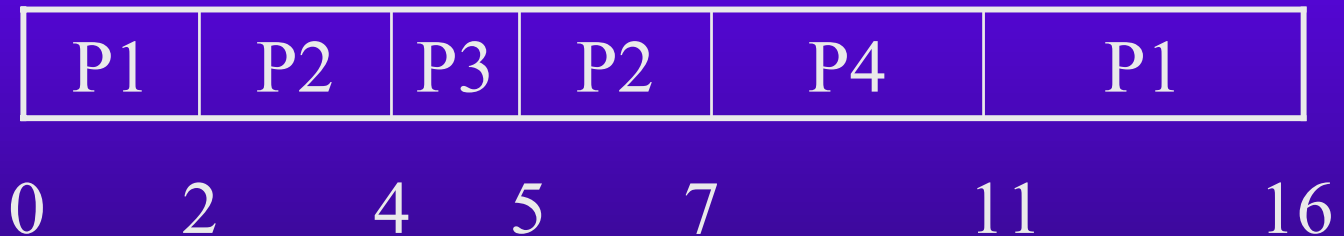


◆ Average waiting time = $(0 + 6 + 3 + 7)/4 - 4$



<u>Process</u>	<u>Arrival Time</u>		<u>Burst Time</u>
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 = $(9 + 1 + 0 + 2)/4 - 3$

Determining Length of Next CPU Burst

- ◆ Can only estimate the length.
- ◆ Can be done by using the length of previous CPU bursts, using exponential averaging.

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define:

- ◆ $\alpha = 0$
 $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
 – $\tau_{n+1} = \tau_n$, Recent history does not count.
- ◆ $\alpha = 1$
 – $\tau_{n+1} = t_n$ Only the actual last CPU burst counts.

- ◆ If we expand the formula, we get:

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

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

Priority Scheduling

- ◆ A priority number (integer) is associated with each process
- ◆ The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority).
 - Preemptive
 - Non preemptive
- ◆ SJF is a priority scheduling where priority is the predicted next CPU burst time.
- ◆ Problem \equiv Starvation – low priority processes may never execute.
- ◆ Solution \equiv Aging – as time progresses increase the priority of the process.

Round Robin (RR)

- ◆ Each process gets a small unit of CPU time (*time quantum*), usually 10-100 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 chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- ◆ Performance
 - q large \Rightarrow FIFO
 - q small $\Rightarrow q$ must be large with respect to context switch, otherwise overhead is too high.



Process Burst Time

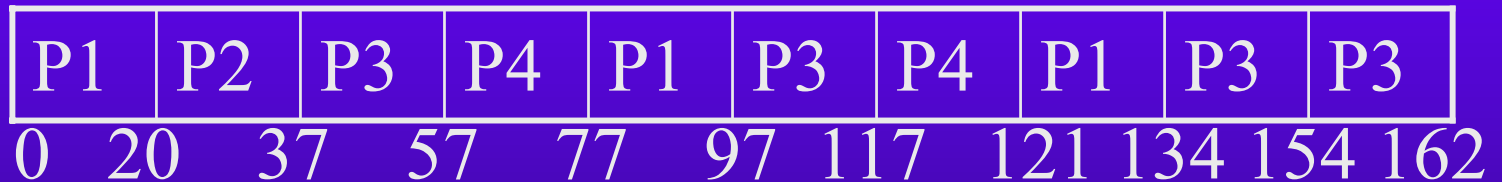
P_1 53

P_2 17

P_3 68

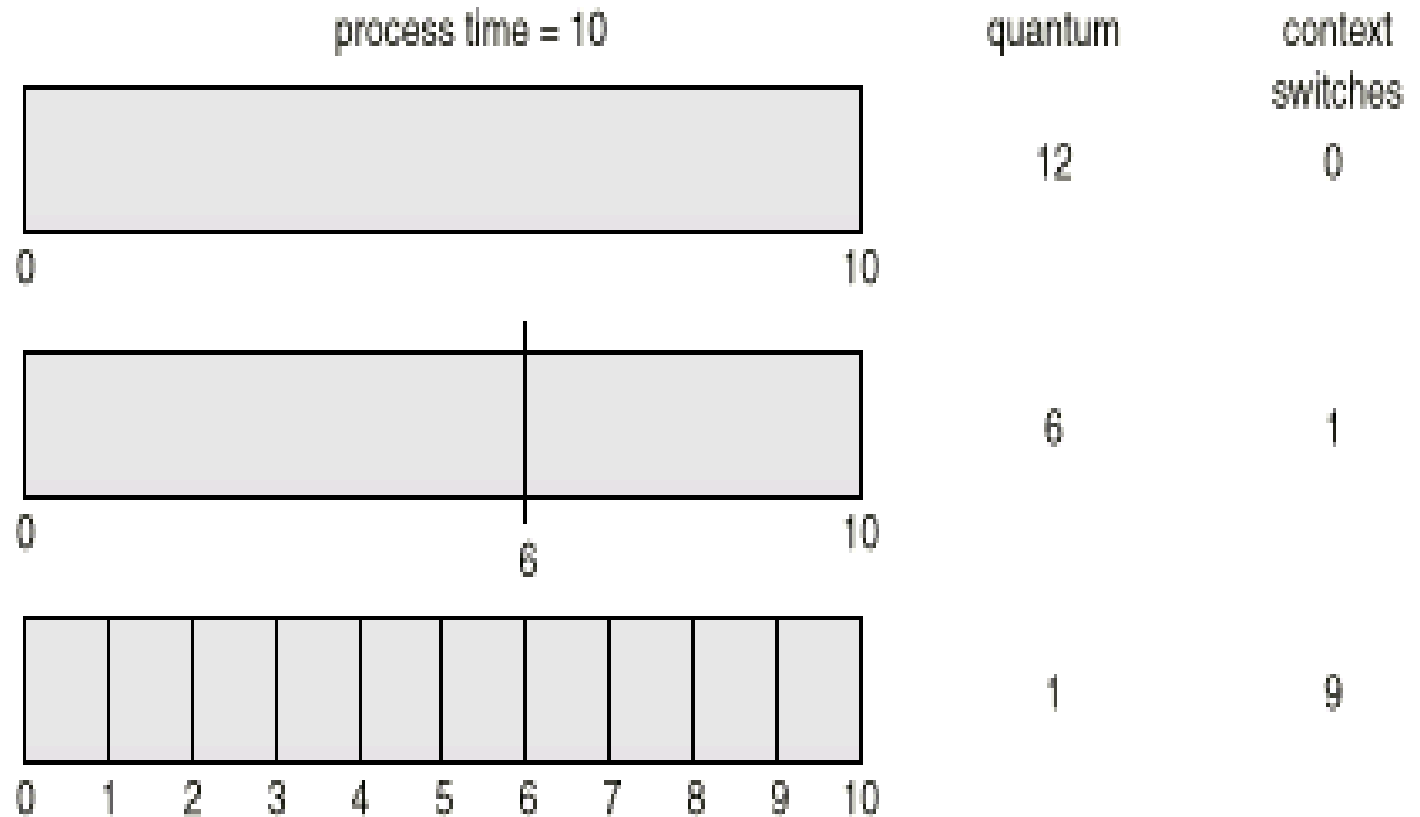
P_4 24

◆ The Gantt chart is:

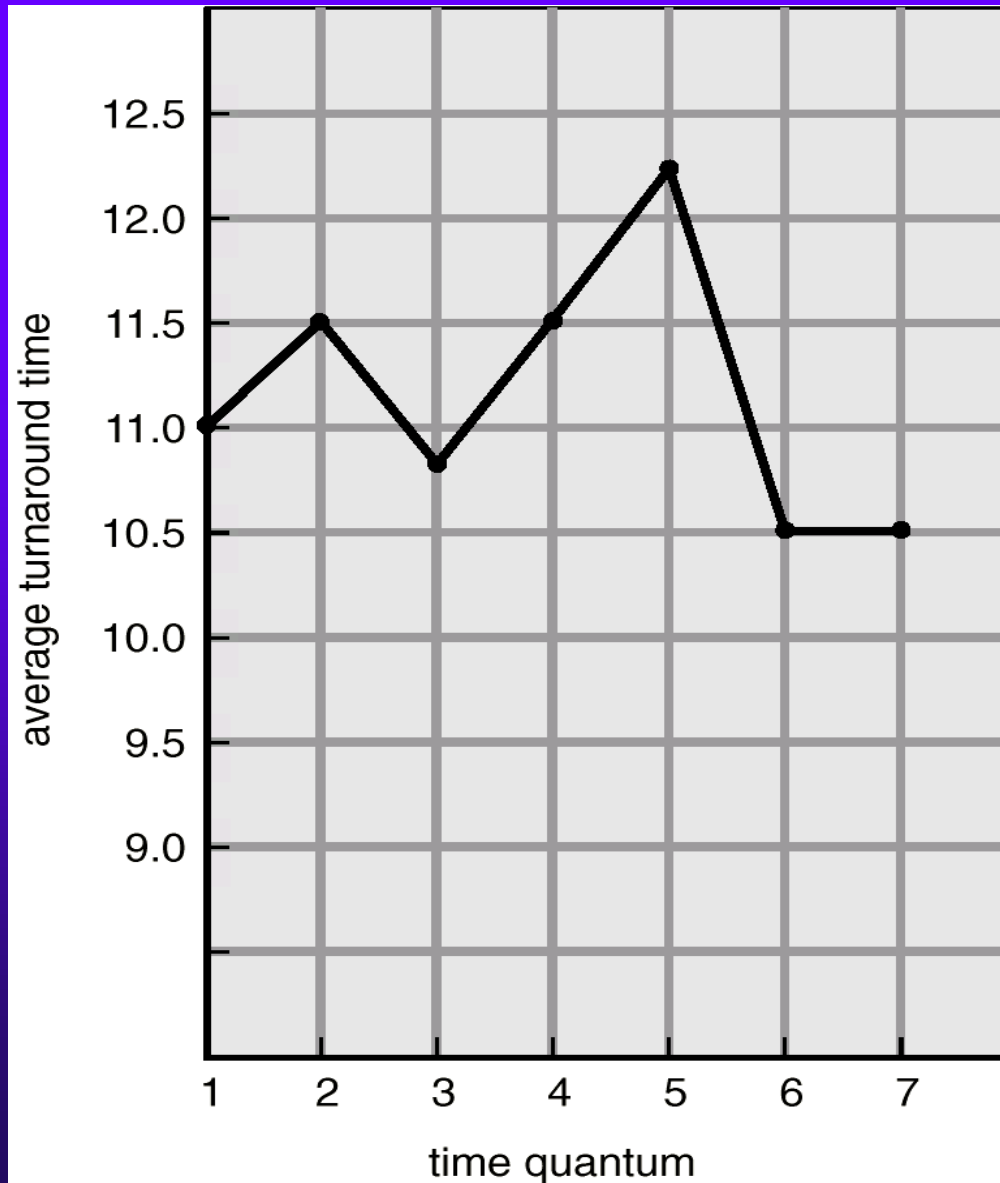


◆ Typically, higher average turnaround than SJF, but better *response*.

How a Smaller Time Quantum Increases Context Switches

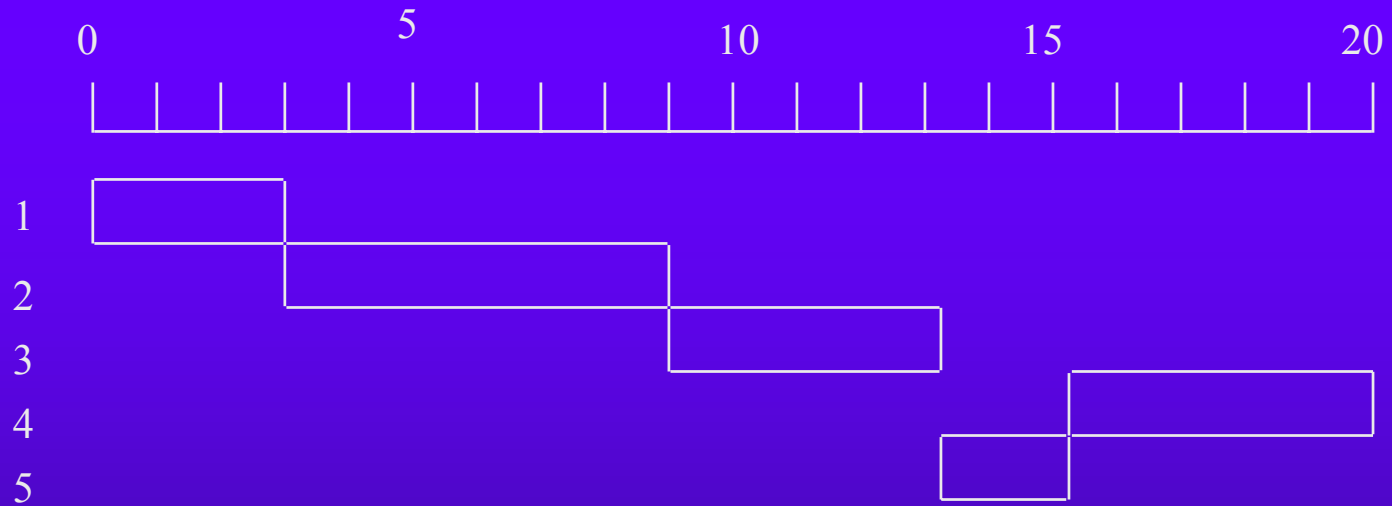


Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Highest Response Ratio Next (HRRN)



- ◆ Choose next process with the lowest ratio

$$\frac{\text{time spent waiting} + \text{expected service time}}{\text{expected service time}}$$



Fair-Share Scheduling

- ◆ User's application runs as a collection of processes (threads)
- ◆ User is concerned about the performance of the application
- ◆ Need to make scheduling decisions based on process sets

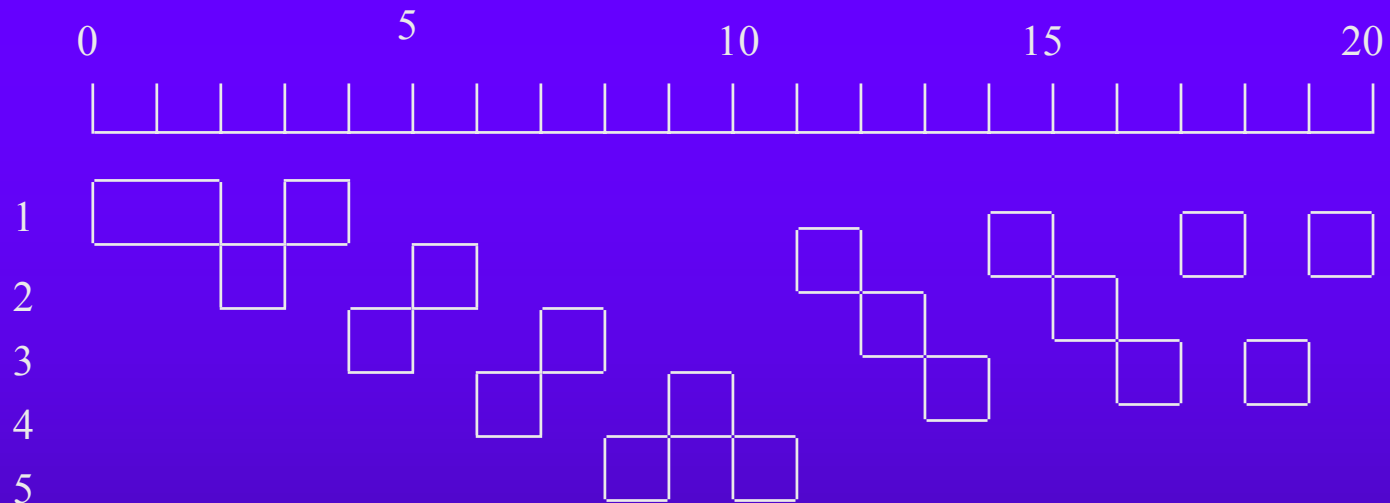


Time	Process A			Process B			Process C		
	Priority	Process	Group	Priority	Process	Group	Priority	Process	Group
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		.	.						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					.	.			.
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		.	.						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						.		.	.
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		.	.						
		78	78						
5	98	39	39	70	3	18	76	15	18

Shaded rectangle represents executing process

Figure 9.16 Example of Fair Share Scheduler Three Processes, Two Groups

Feedback



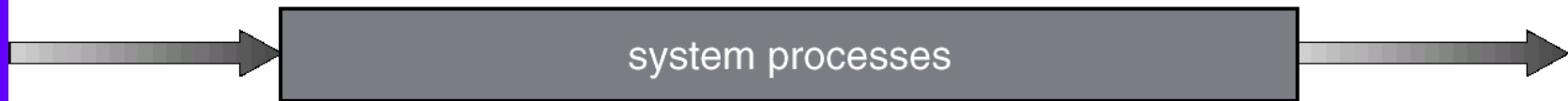
- ◆ Penalize jobs that have been running longer
- ◆ Don't know remaining time process needs to execute

Multilevel Queue

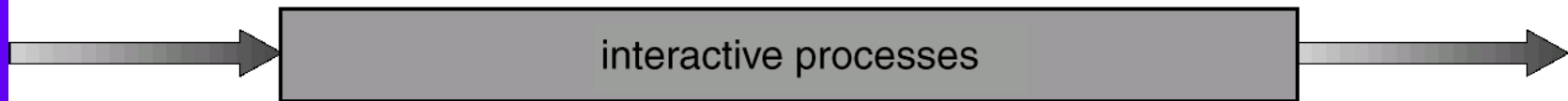
- ◆ Ready queue is partitioned into separate queues:
 - foreground (interactive)
 - background (batch)
- ◆ Each queue has its own scheduling algorithm,
 - foreground – RR
 - background – FCFS
- ◆ Scheduling must be done between the queues.
 - Fixed priority scheduling; i.e., serve all from foreground then from background. Possibility of starvation.
 - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
 - 20% to background in FCFS



highest priority



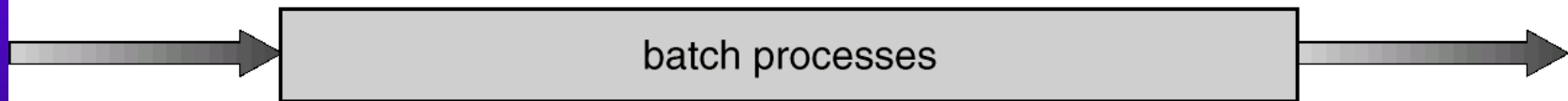
system processes



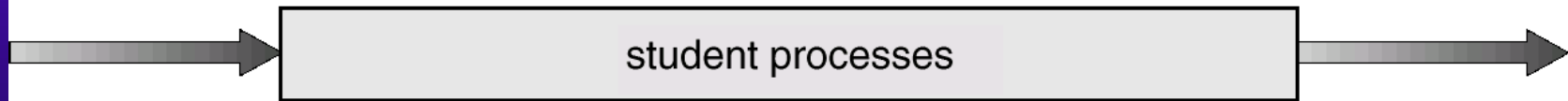
interactive processes



interactive editing processes



batch processes



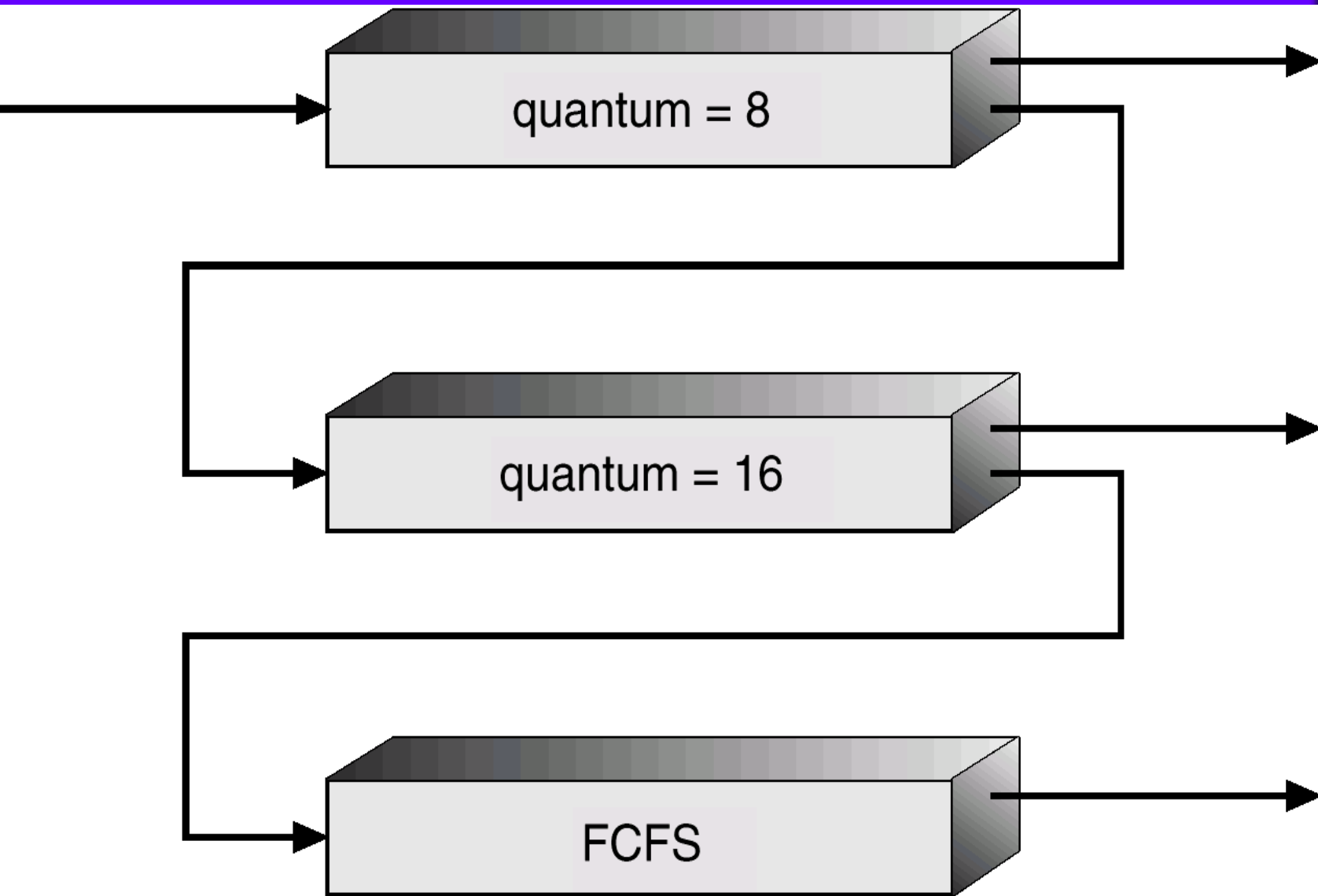
student processes



lowest priority

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service
- Three queues:
 - Q_0 – time quantum 8 milliseconds Q_1 – time quantum 16 milliseconds Q_2 – FCFS
- Scheduling
 - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
 - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .



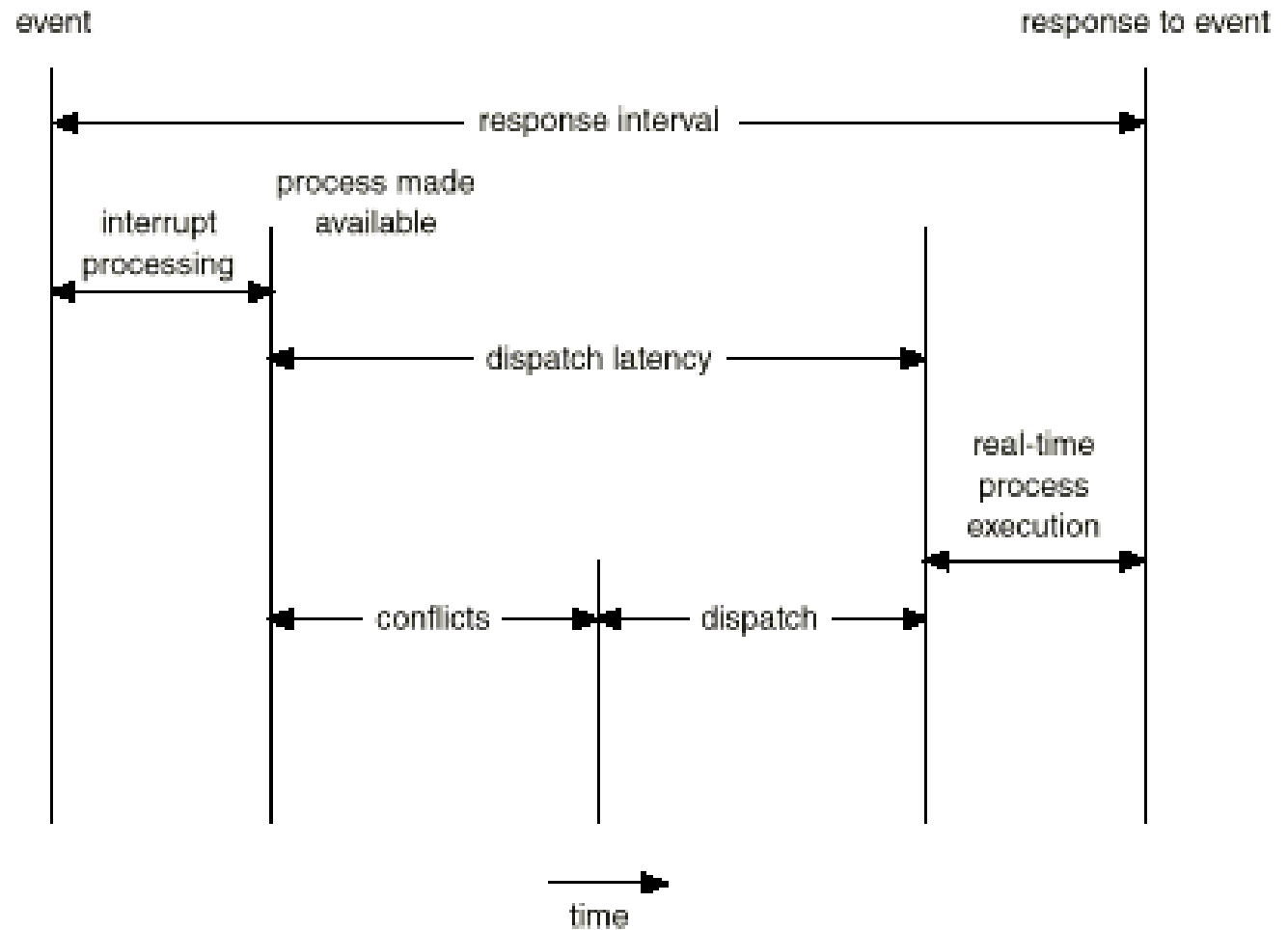
Multiple-Processor Scheduling

- ◆ CPU scheduling more complex when multiple CPUs are available.
- ◆ *Homogeneous processors* within a multiprocessor.
- ◆ *Load sharing*
- ◆ *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.

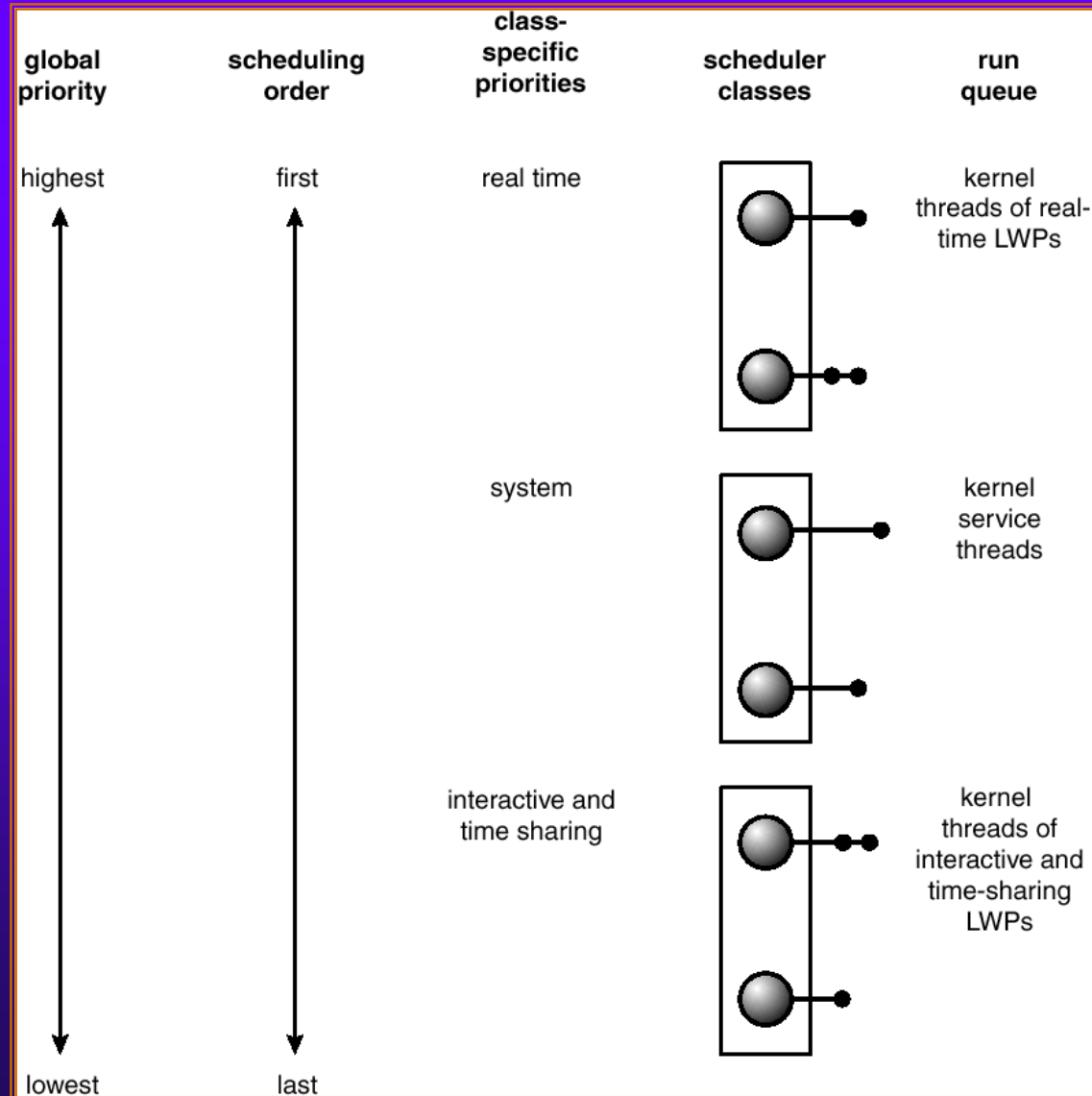
Real-Time Scheduling

- ◆ *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time.
- ◆ *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones.

Dispatch Latency



Solaris 2 Scheduling



Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1



Traditional UNIX Scheduling

- ◆ Multilevel feedback using round robin within each of the priority queues
- ◆ Priorities are recomputed once per second
- ◆ Base priority divides all processes into fixed bands of priority levels
- ◆ Adjustment factor used to keep process in its assigned band



Bands

- ◆ Decreasing order of priority
 - Swapper
 - Block I/O device control
 - File manipulation
 - Character I/O device control
 - User processes



Time	Process A		Process B		Process C	
	Priority	CPU Count	Priority	CPU Count	Priority	CPU Count
0	60	0 1 2 . . 60	60	0	60	0
1	75	30	60	0 1 2 . . 60	60	0
2	67	15	75	30	60	0 1 2 . . 60
3	63	7 8 9 . . 67	67	15	75	30
4	76	33	63	7 8 9 . . 67	67	15
5	68	16	76	33	63	7

Shaded rectangle represents executing process

Figure 9.17 Example of Traditional UNIX Process Scheduling

Linux Scheduling

◆ Two algorithms: time-sharing and real-time

◆ Time-sharing

- Prioritized credit-based – process with most credits is scheduled next
- Credit subtracted when timer interrupt occurs
- When credit = 0, another process chosen
- When all processes have credit = 0, recrediting occurs
 - Based on factors including priority and history

◆ Real-time

- Soft real-time
- Posix.1b compliant – two classes
 - FCFS and RR
 - Highest priority process always runs first

Thread Scheduling

- ◆ Local Scheduling – How the threads library decides which thread to put onto an available LWP
- ◆ Global Scheduling – How the kernel decides which kernel thread to run next