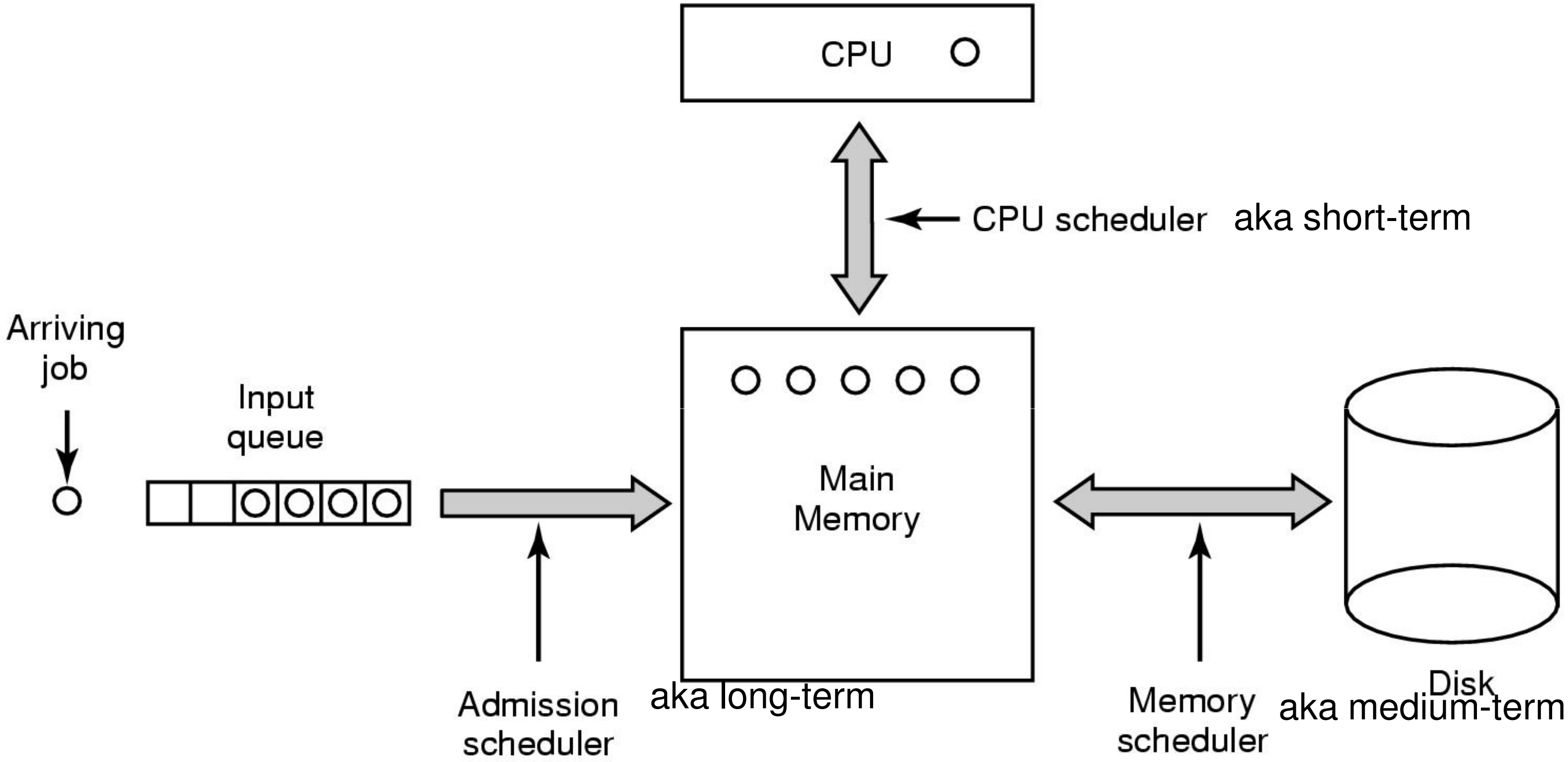
Uniprocessor Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms

Three level scheduling



Types of Scheduling

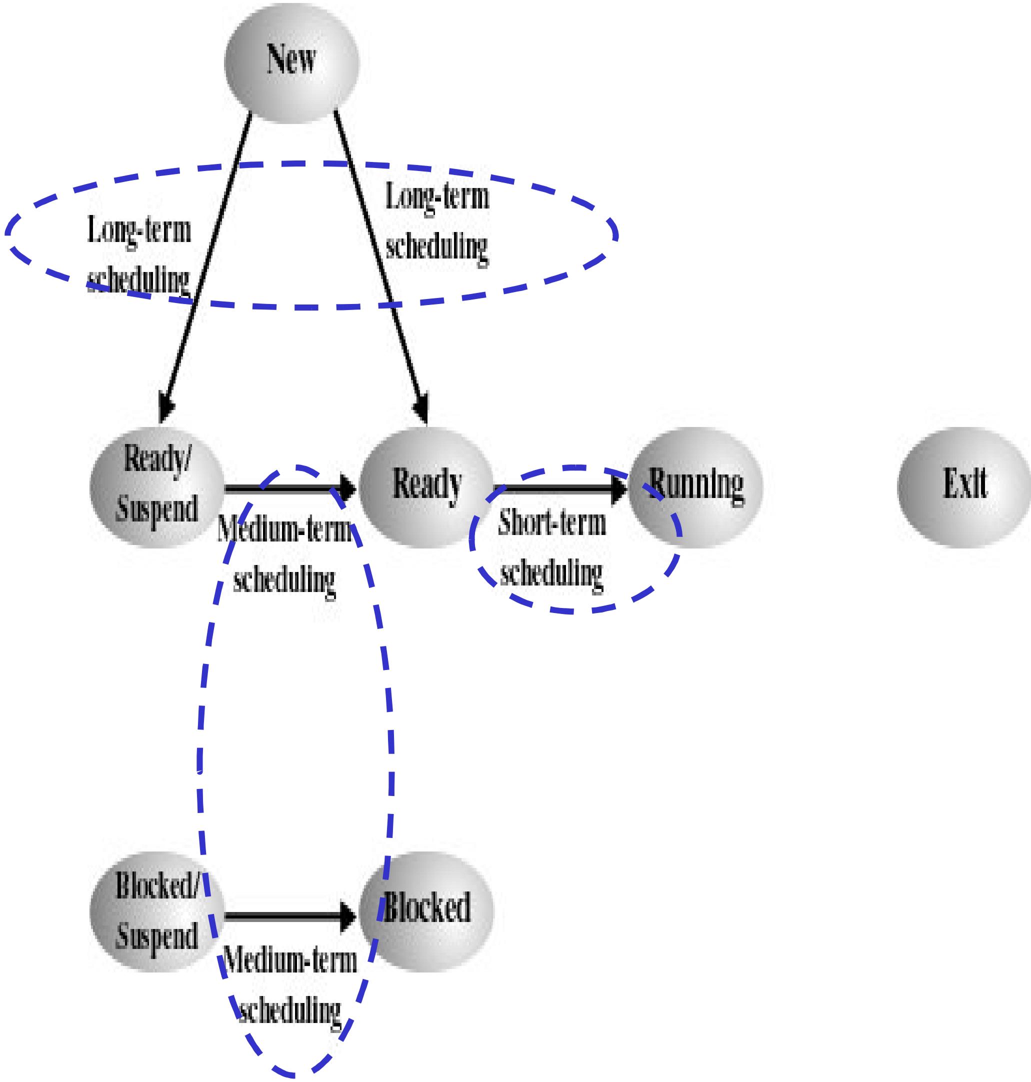


Figure 9.1 Scheduling and Process State Transitions

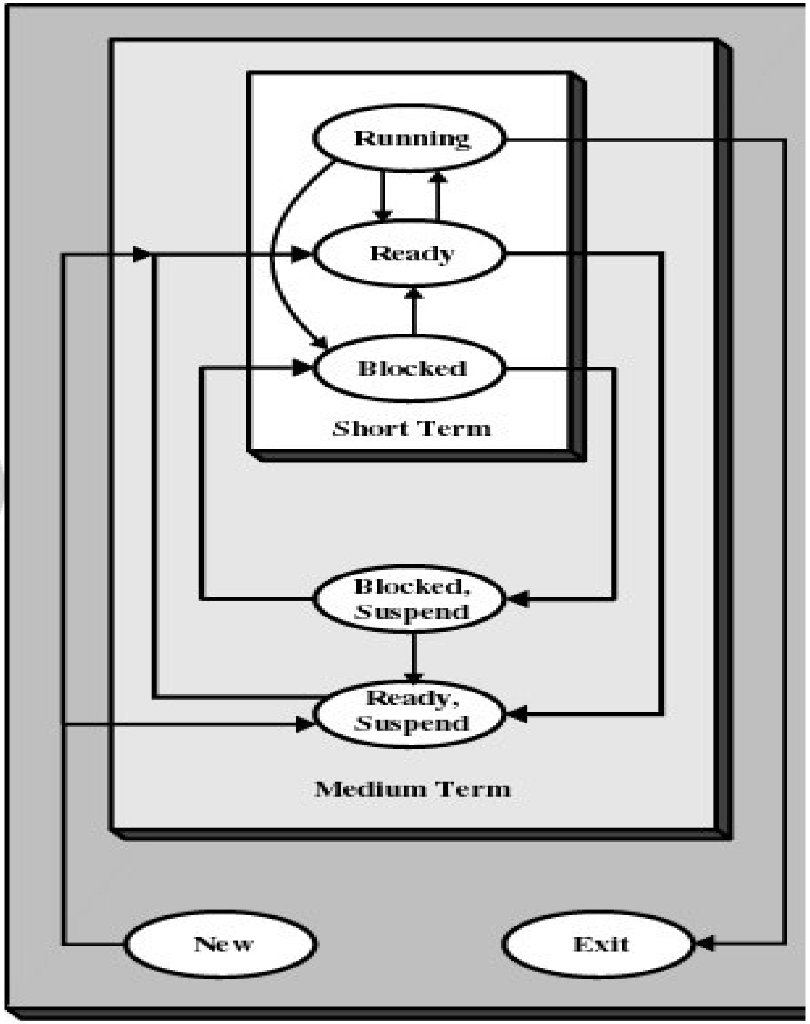


Figure 9.2 Levels of Scheduling

Long- and Medium-Term Schedulers

Long-term scheduler

- Determines which programs are admitted to the system (ie to become processes)
- requests can be denied if e.g. thrashing or overload

Medium-term scheduler

- decides when/which processes to suspend/resume

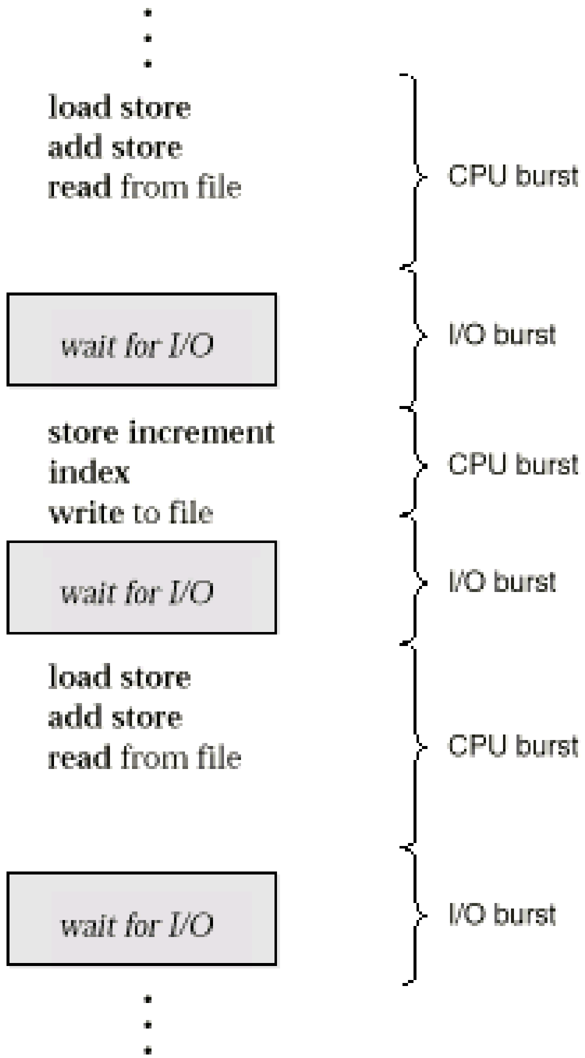
- Both control the degree of multiprogramming
  - More processes, smaller percentage of time each process is executed

Short-Term Scheduler (this is our focus here)

- Decides which process will be *dispatched*; invoked upon
  - Interrupts, Operating system calls, Signals, ..
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running; the dominating factors involve:
  - switching context
  - selecting the new process to dispatch

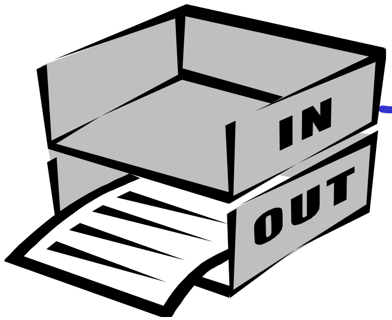
CPU-I/O Burst Cycle

- Process execution consists of a *cycle* of
  - CPU execution and
  - I/O wait.
- A process may be
  - CPU-bound
  - IO-bound



Scheduling Criteria- Optimization goals

CPU utilization – keep CPU busy (when there is work to do)



Throughput – # of processes that complete per time unit

Response time – time between a request submission until the response (execution + **waiting** time)

Fairness - watch priorities, avoid starvation, ...

Overhead e.g. context switching, computing priorities, …

Decision Mode

Nonpreemptive

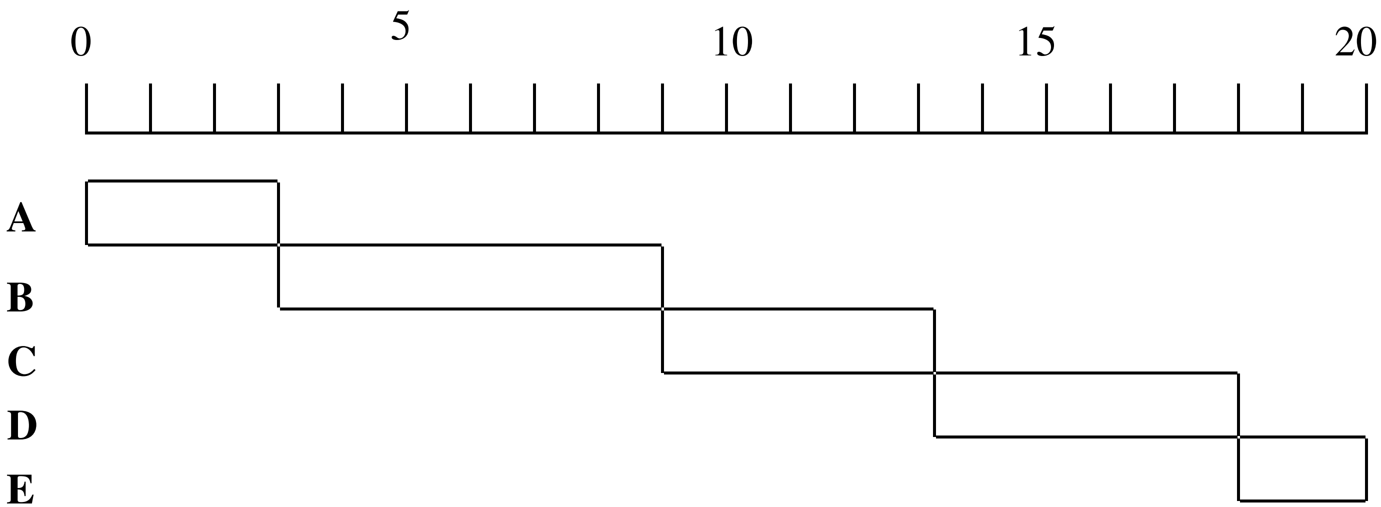
- Once a process is in the running state, it will continue until it terminates or blocks itself for I/O

Preemptive

- Currently running process may be interrupted and moved to the Ready state by the operating system
- Allows for better interactive service since any one process cannot monopolize the processor for very long

Algorithms/methods for scheduling - single processor

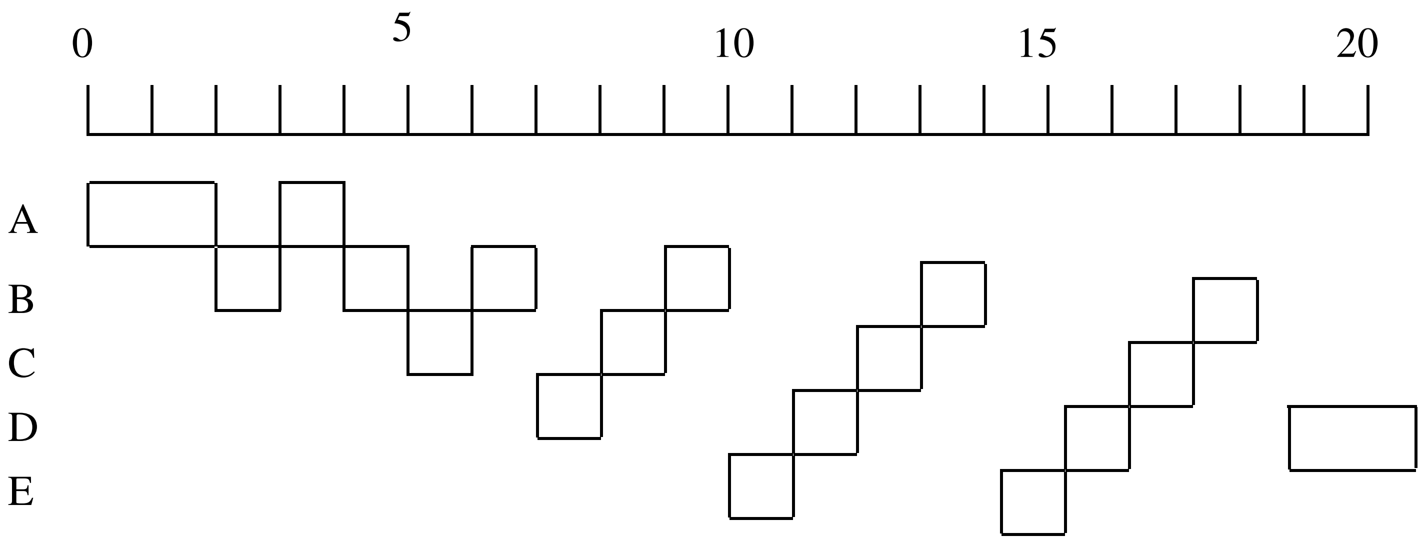
First-Come-First-Served (FCFS)



Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- non-preemptive
- Favors CPU-bound processes
- A short process may have to wait very long before it can execute (**convoy effect**)

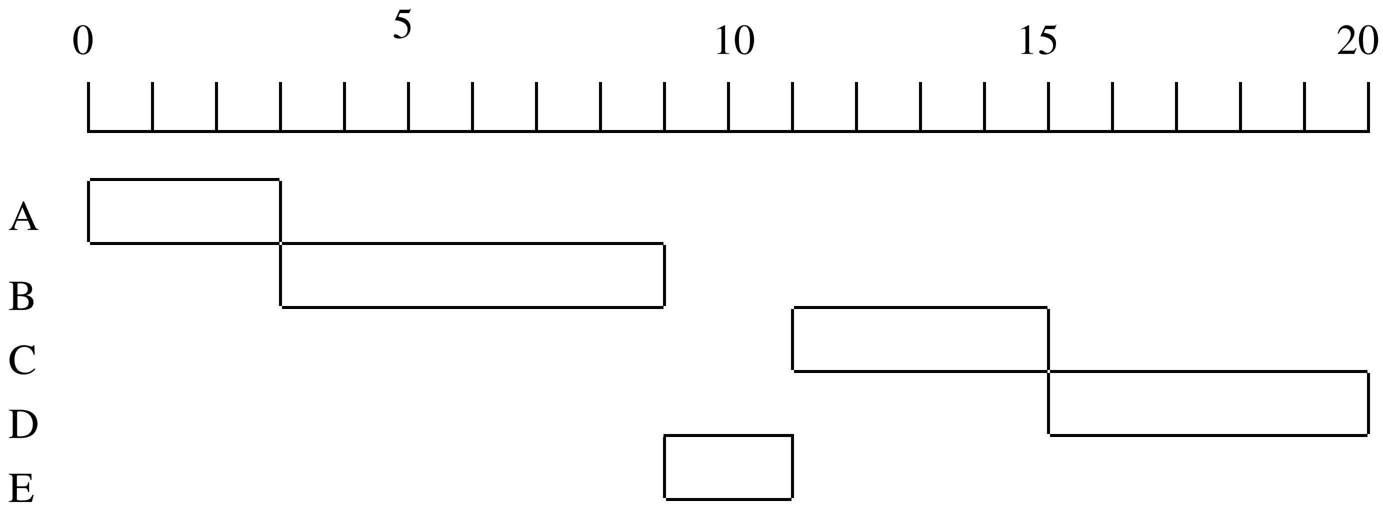
Round-Robin



Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- preemption based on clock (interrupts on *time slice* or *quantum* -q- usually 10-100 msec)
- fairness: for *n* processes, each gets 1/*n* of the CPU time in chunks of at most *q* time units
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *overhead can be high due to context switches*

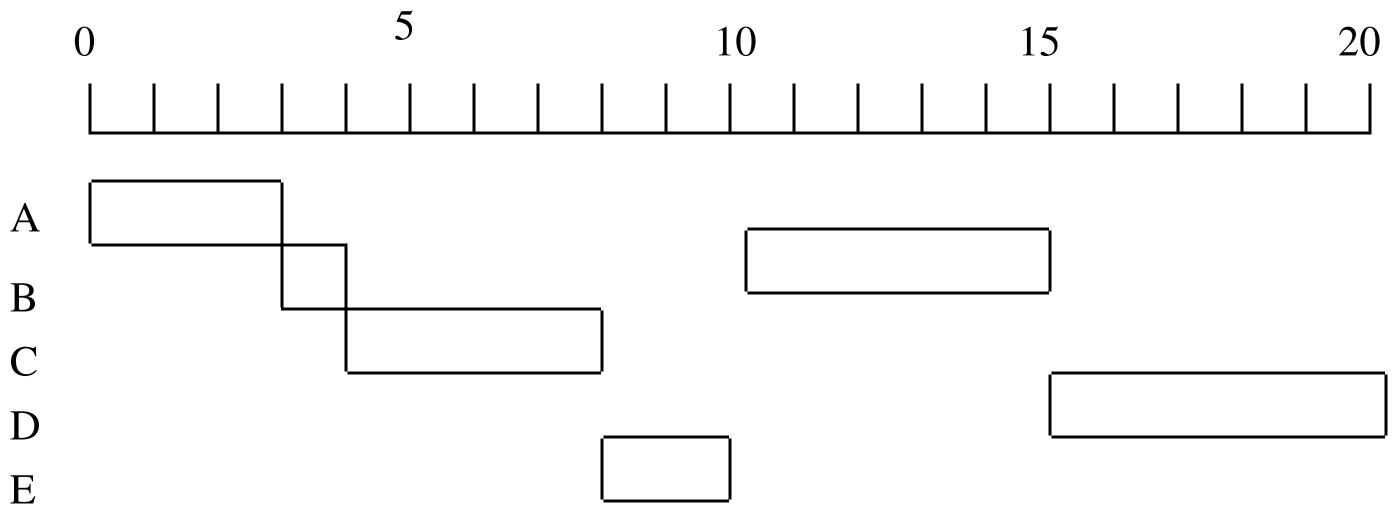
Shortest Process First



Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

- Non-preemptive
- Short process jumps ahead of longer processes
- Avoid convoy effect

Preemptive SPF: Shortest Remaining Time First

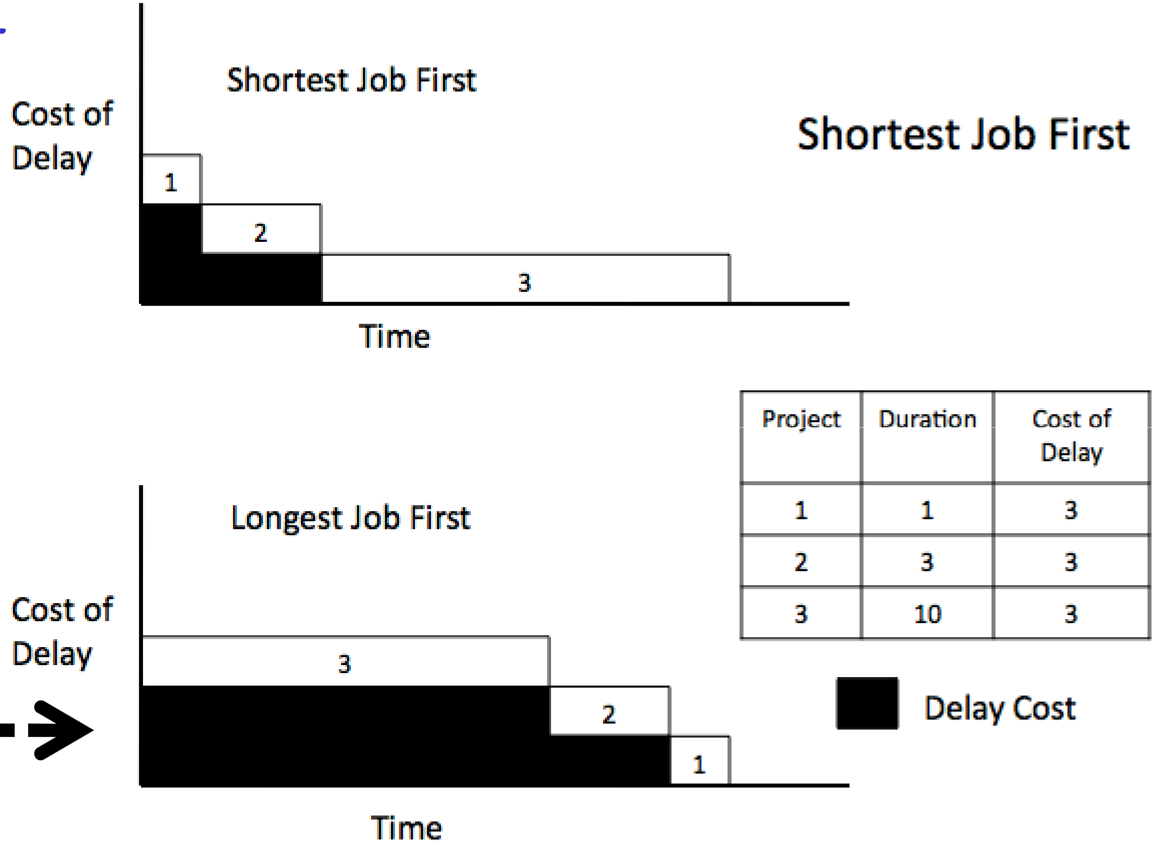


Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Preemptive (at arrival) version of shortest process next

On SPF Scheduling

- gives *high throughput*
- gives *minimum* (optimal) sum (also *average) response (waiting) time* for a given set of processes
  - Proof (non-preemptive): analyze the summation giving the waiting time
- Intuition: ------→



From "The Principles of Product Development Flow," by Donald G. Reinertsen. Celeritas Publishing: 2009. Copyright 2009, Donald G. Reinertsen

But: possibility of *starvation* for longer processes

On SPF Scheduling (cont)

- must estimate processing time (next cpu burst)
  - Can be done automatically (exponential averaging)
  - If estimated time for process (given by the user in a batch system) not correct, the operating system may abort it

Determining Length of Next CPU Burst

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

1. $t_n =$ actual lenght of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. $\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

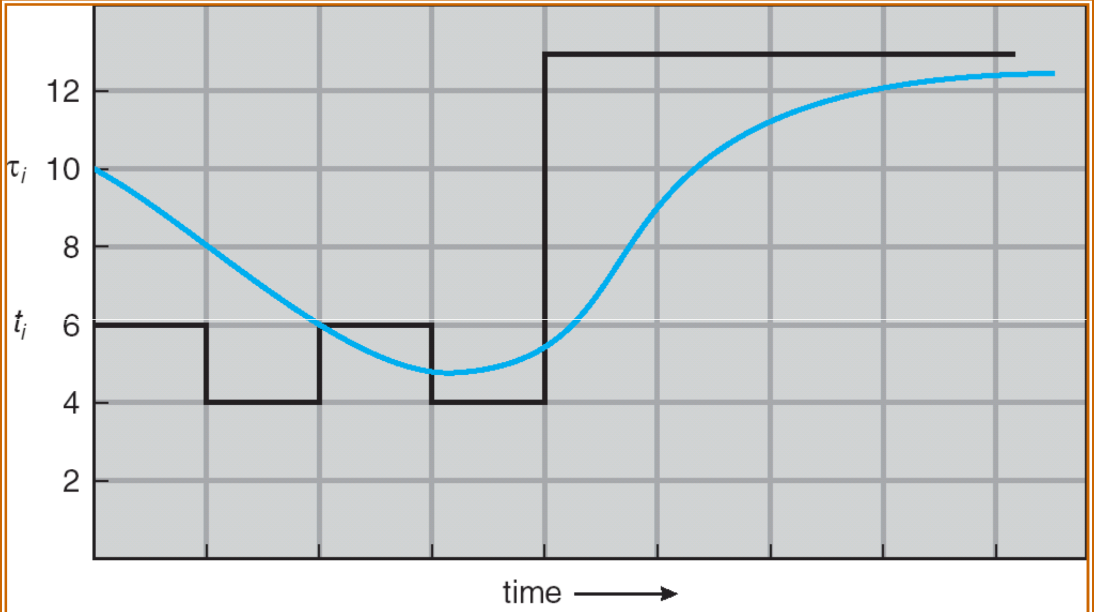
On Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - history does not count, only initial estimation counts
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts.
- If we expand the formula, we get:

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

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

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 . . .

Priority Scheduling: General Rules

- Scheduler can choose a process of higher priority over one of lower priority
  - can be preemptive or non-preemptive
  - can have multiple ready queues to represent multiple level of priority
- **Example Priority Scheduling**: SPF, where priority is the predicted next CPU burst time.
- **Problem** ≡ Starvation – low priority processes may never execute.
- ***A* solution** ≡ *Aging* – as time progresses increase the priority of the process.

Priority Scheduling Cont. : Multilevel Queue

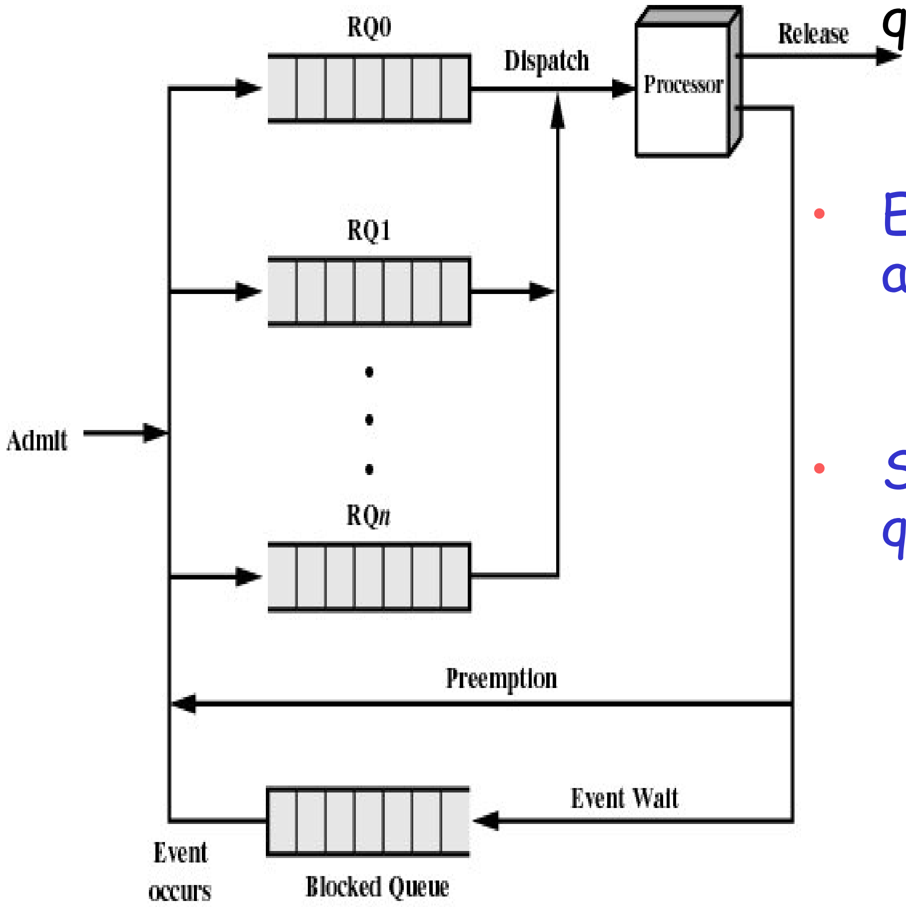
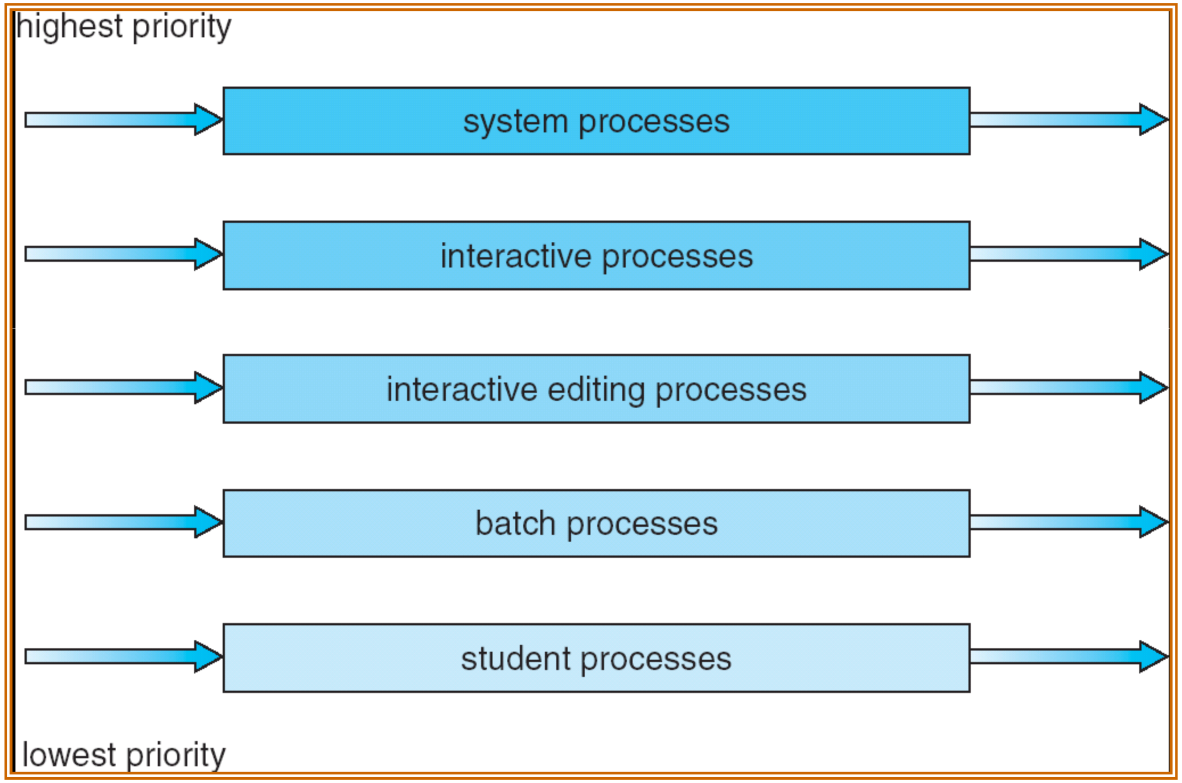


Figure 9.4 Priority Queuing

- Ready queue is partitioned into separate queues, eg
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm, eg
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues.
  - **Fixed option** eg., serve all from foreground then from background. Possible **starvation**.
  - **Alternative**: Time slice – each queue gets a fraction of CPU time to divide amongst its processes, eg.
    - 80% to foreground in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue

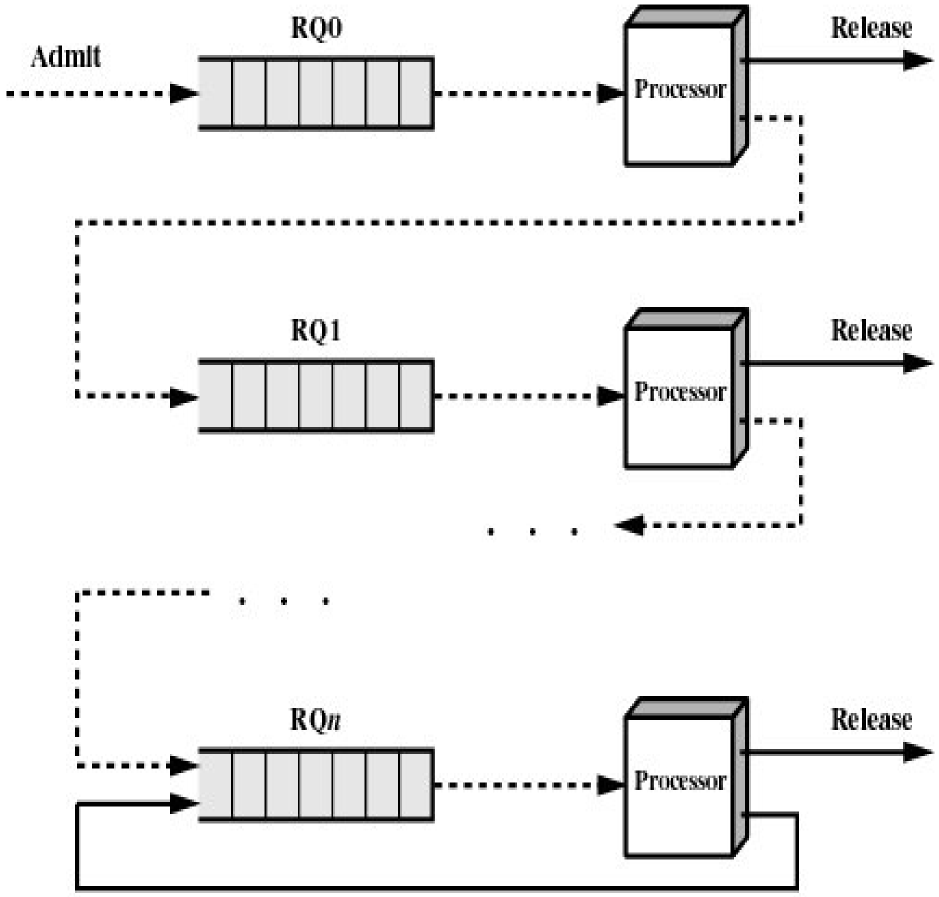
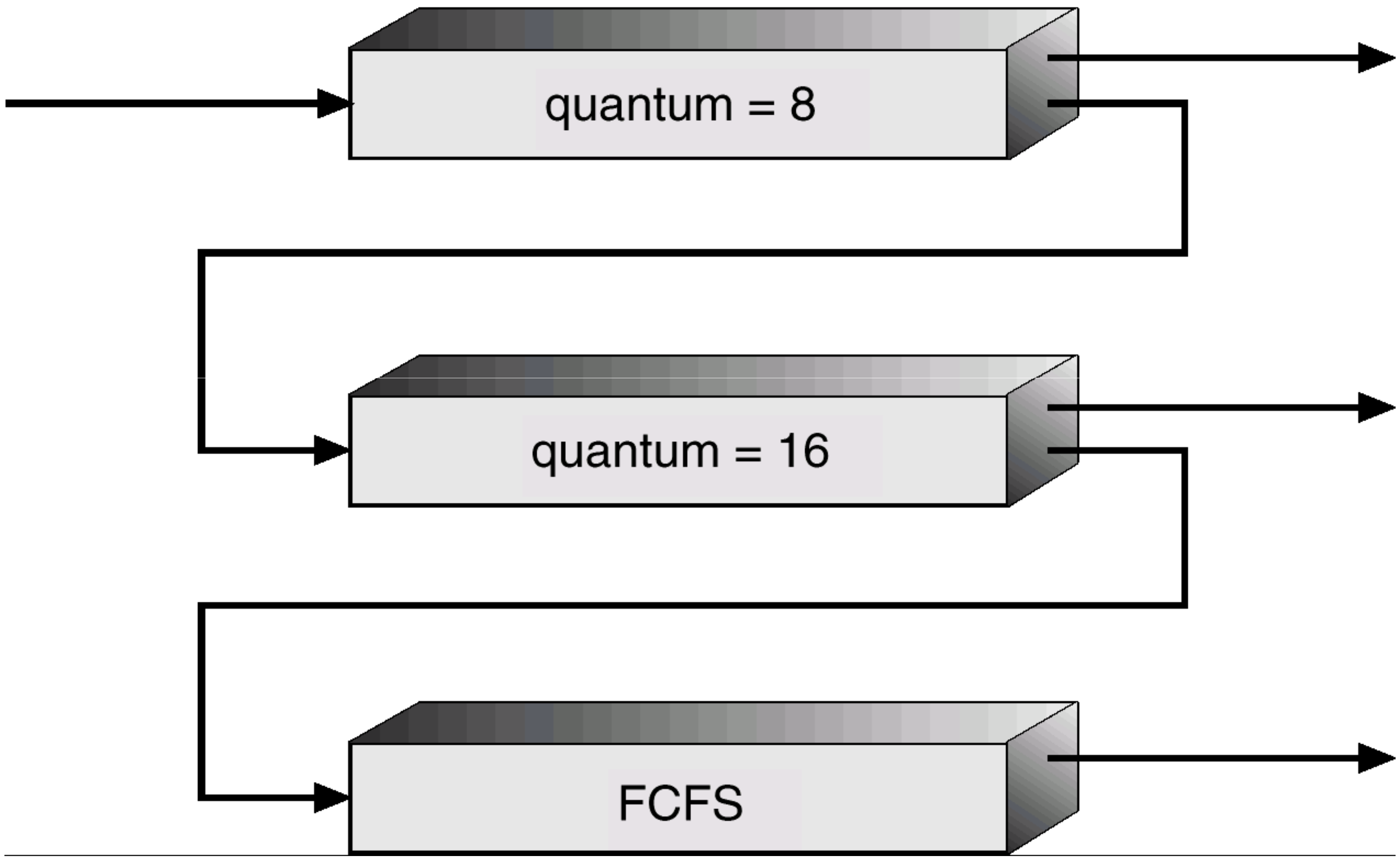


Figure 9.10 Feedback Scheduling

- A process can move between the various queues; **aging** can be implemented this way.
- scheduler parameters:
  - number of queues
  - scheduling algorithm for each queue
  - method to upgrade a process
  - method to demote a process
  - method to determine which queue a process will enter first

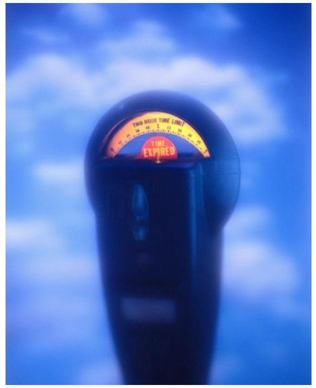
Multilevel Feedback Queues



Real-Time Scheduling

Real-Time Systems

- Tasks or processes attempt to interact with outside-world events , which occur in "real time"; process must be able to keep up, e.g.
  - Control of laboratory experiments, Robotics, Air traffic control, Drive-by-wire systems, Tele/Data-communications, Military command and control systems
- Correctness of the RT system depends not only on the logical result of the computation but also on the time at which the results are produced



i.e. Tasks or processes come with a **deadline** (for starting or completion)

Requirements may be **hard** or **soft**

Periodic Real-TimeTasks: Timing Diagram

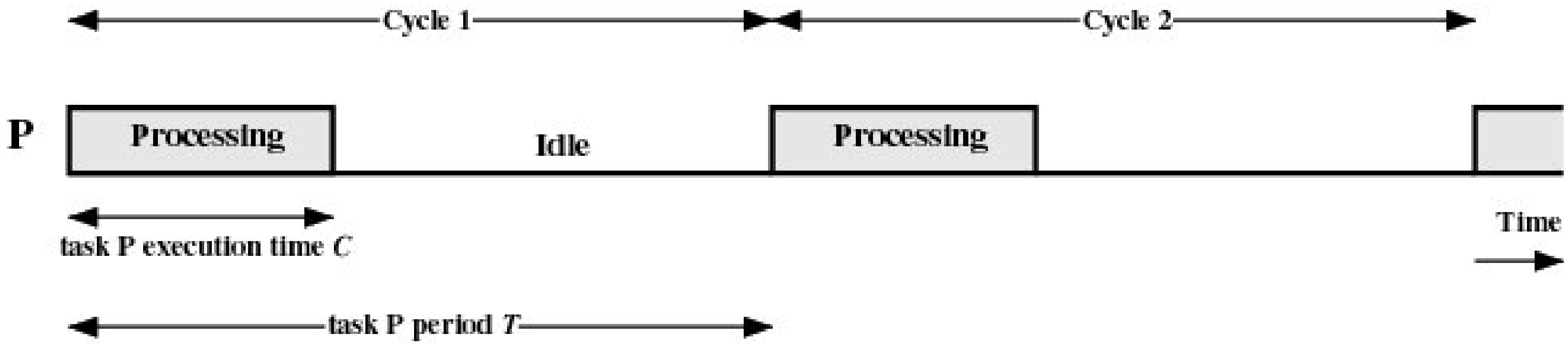
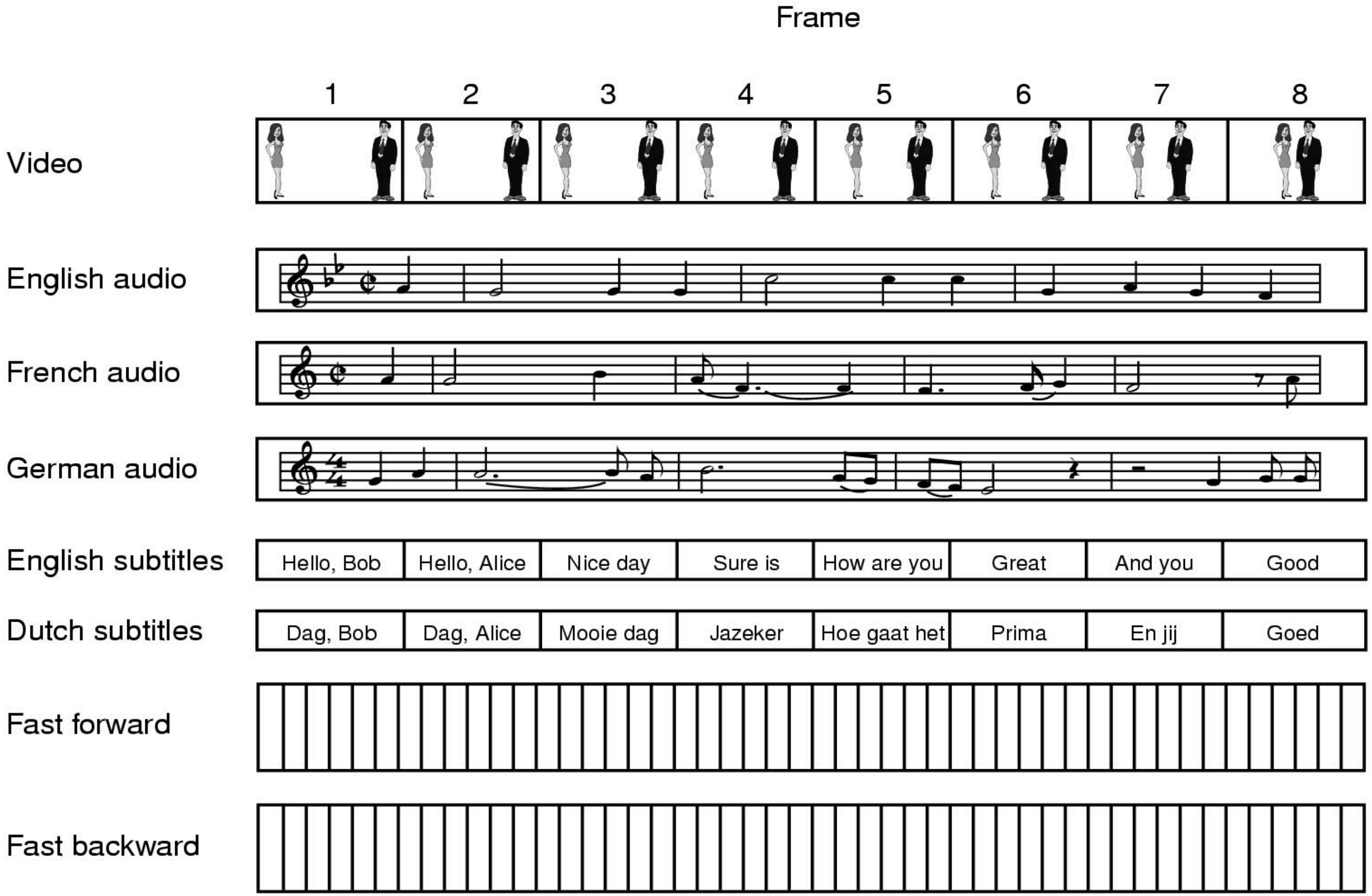


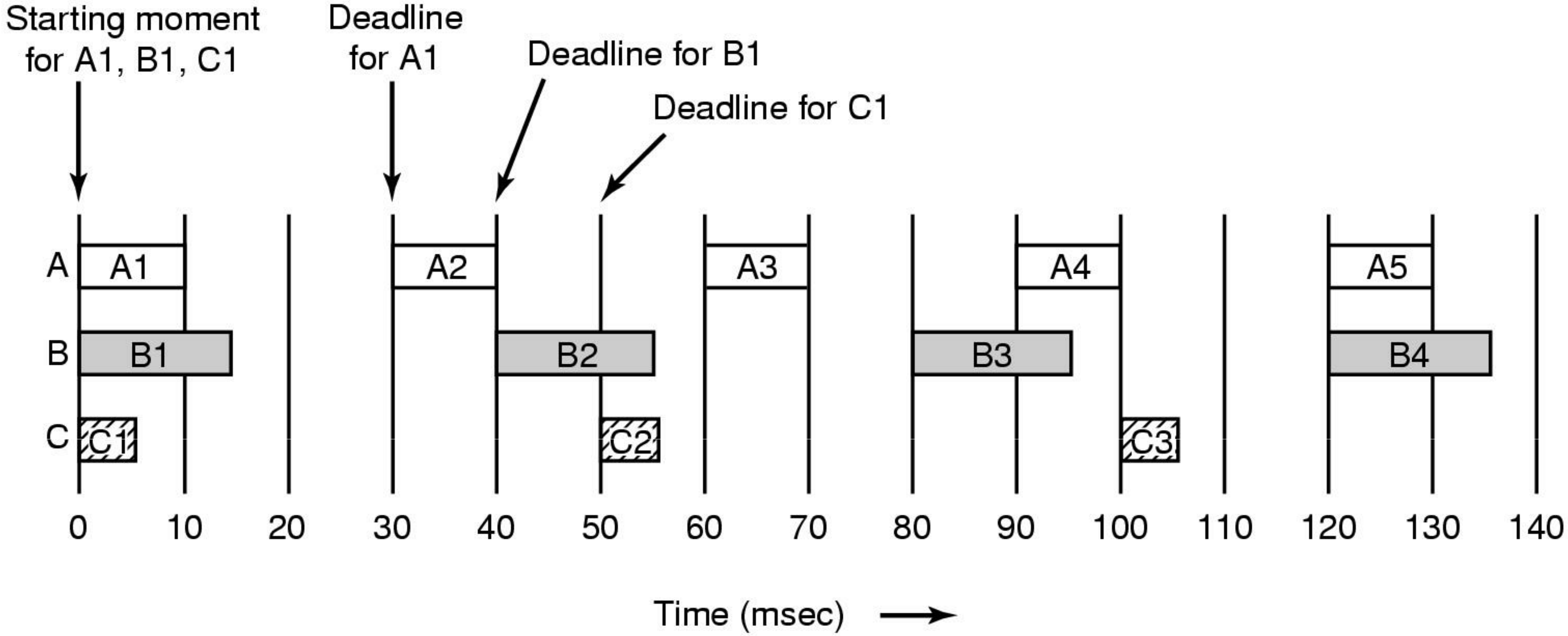
Figure 10.7 Periodic Task Timing Diagram

E.g. Multimedia Process Scheduling



A movie may consist of several files

E.g. Multimedia Process Scheduling (cont)



- Periodic processes displaying a movie
- Frame rates and processing requirements may be different for each movie (or other process that requires time guarantees)

Scheduling in Real-Time Systems

Schedulable real-time system

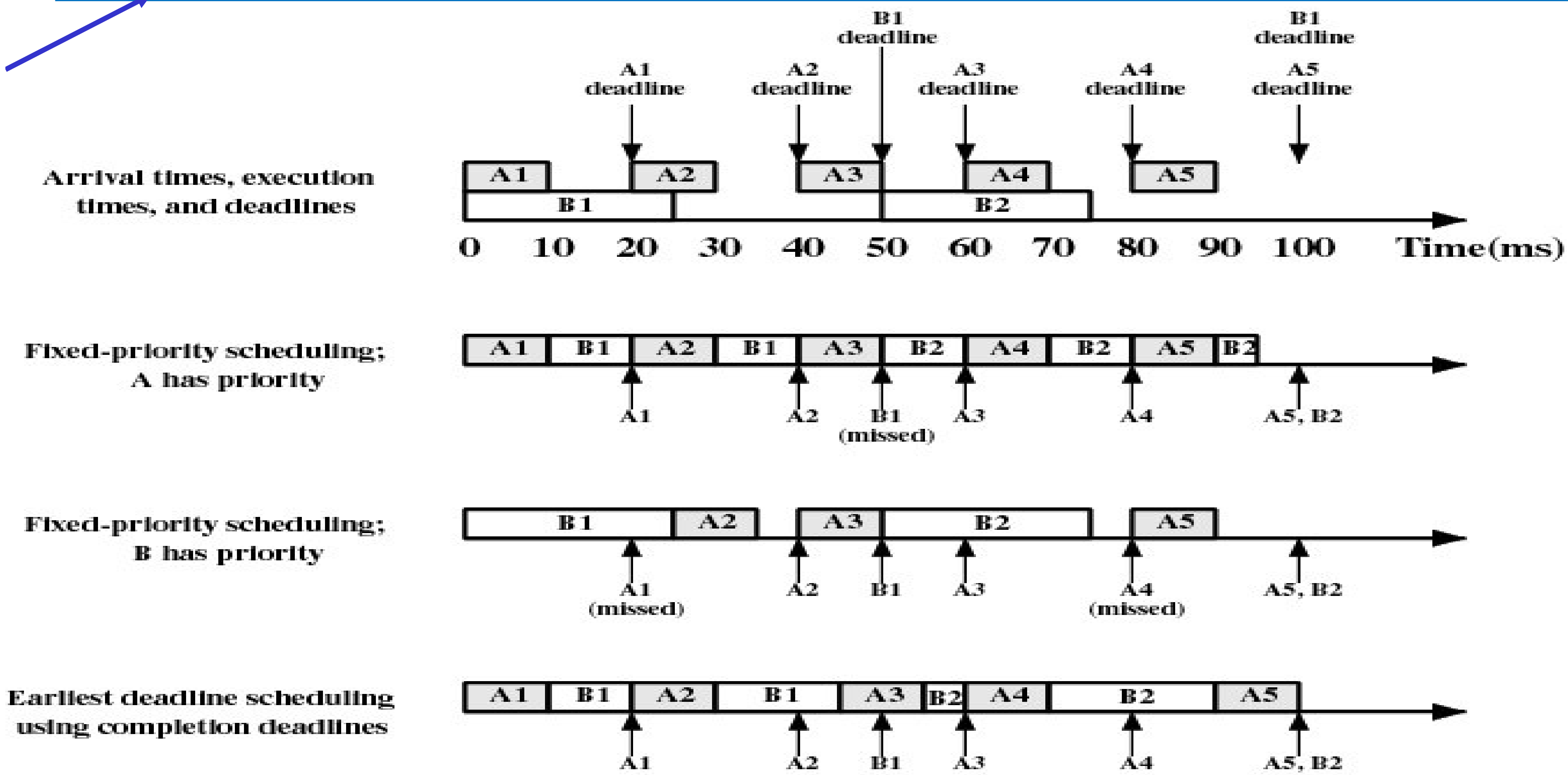
- Given
  - *m* periodic events
  - event *i* occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

$$\textbf{Utilization} = \sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Scheduling with deadlines: Earliest Deadline First

Set of tasks with deadlines is schedulable (can be executed s.t. no process misses its deadline) **iff** EDF is a schedulable (feasible) sequence. (why?)

Example sequences:



Rate Monotonic Scheduling

- Assigns priorities to tasks on the basis of their periods
- Highest-priority task is the one with the shortest period

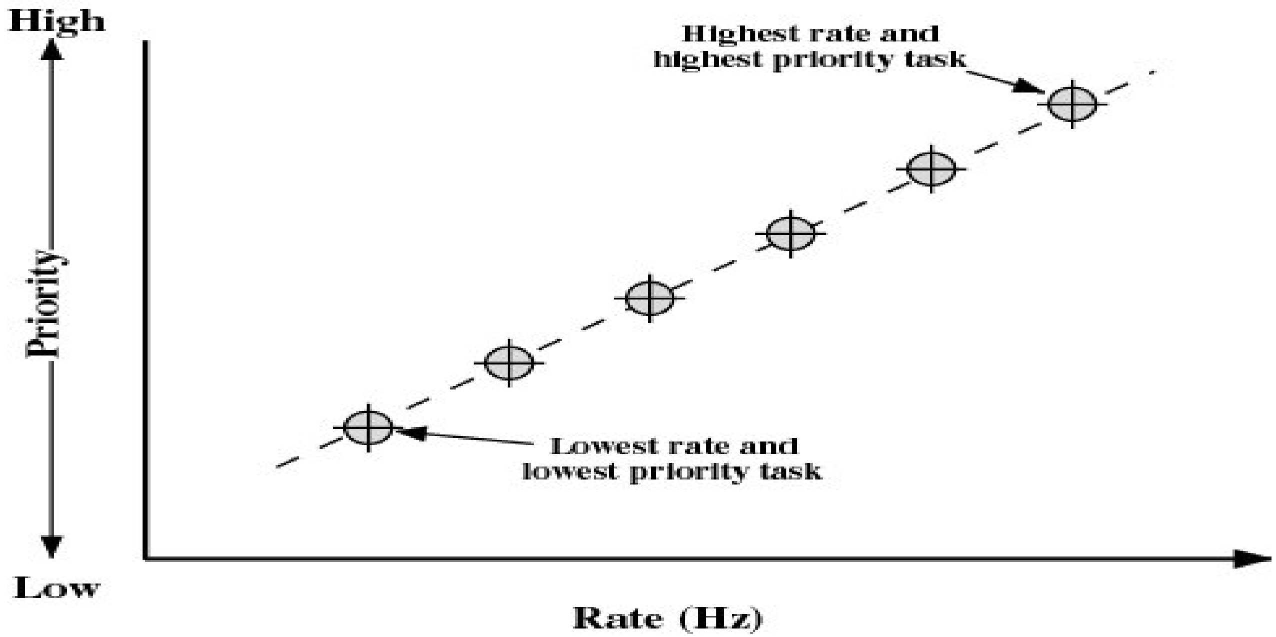
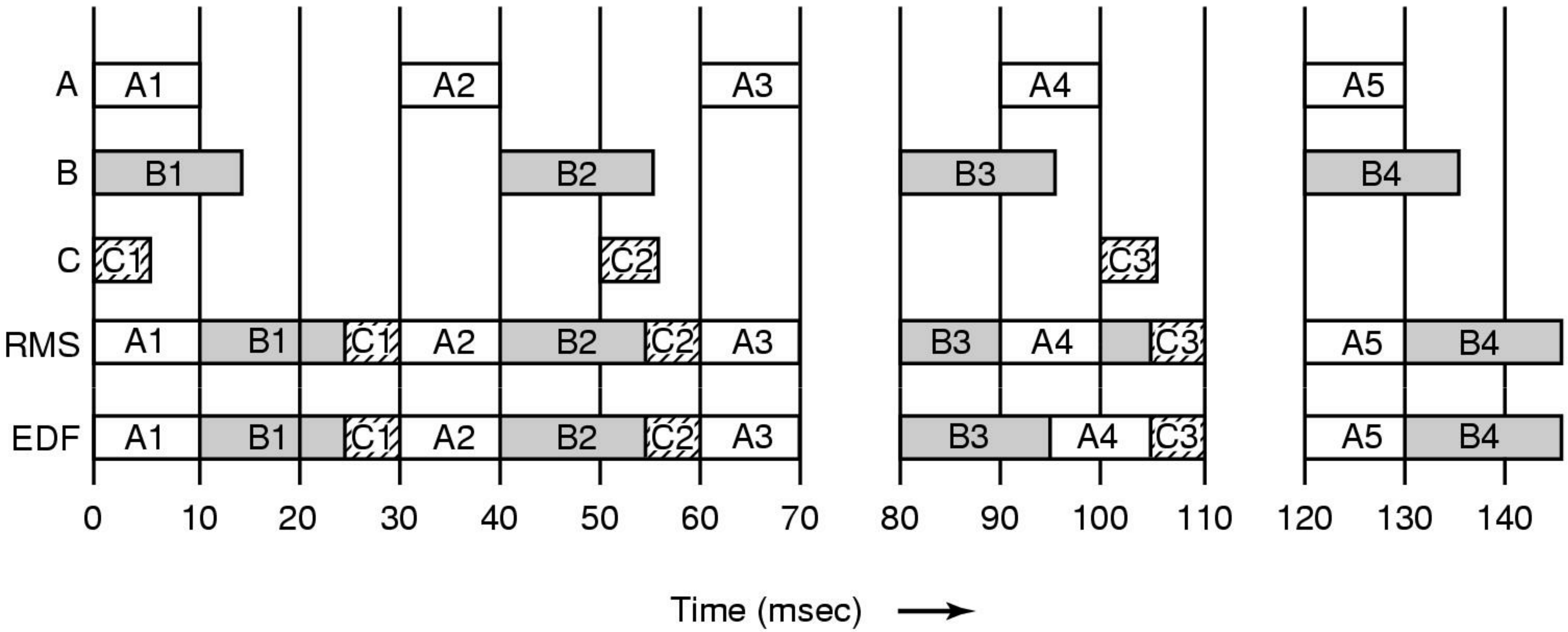


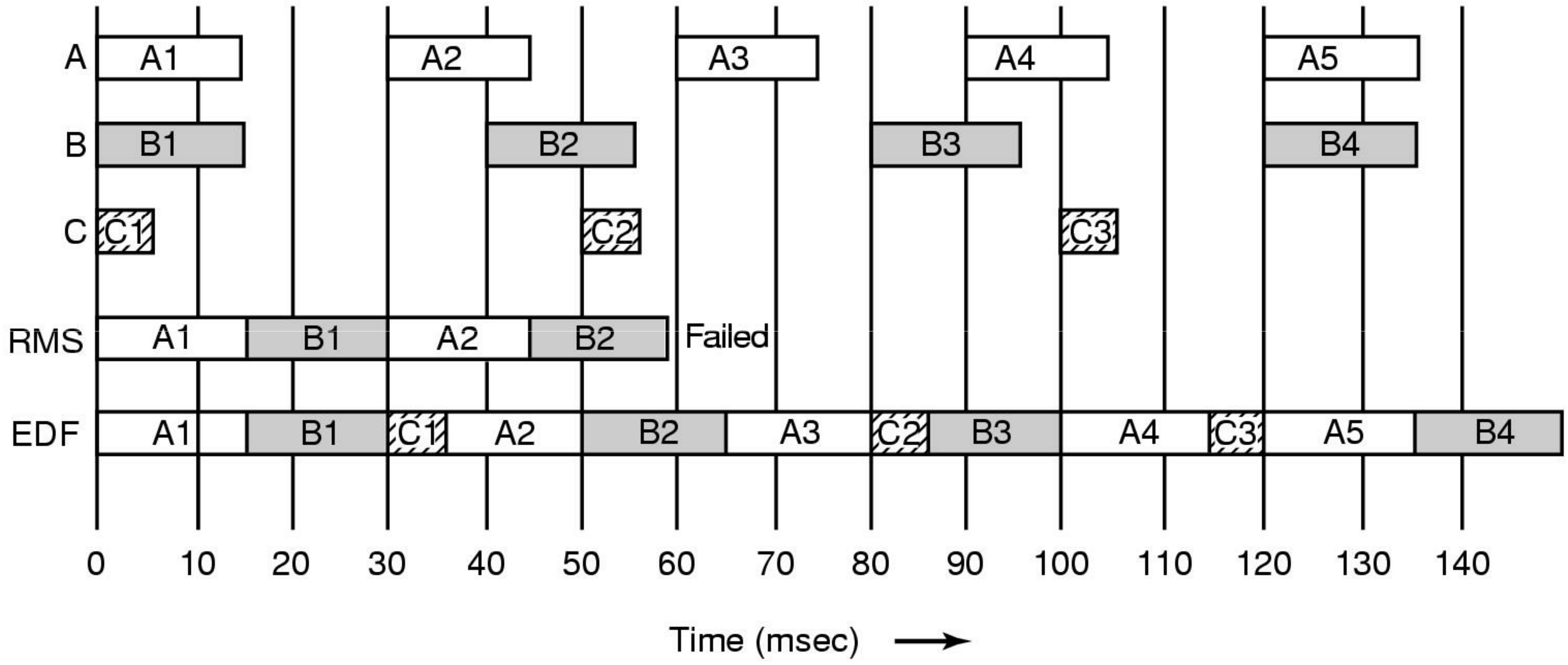
Figure 10.8 A Task Set with RMS [WARR91]

EDF or RMS? (1)



A
B
C
RMS
EDF
A1
A2
A3
A4
A5
B1
B2
B3
B4
C1
C2
C3
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
Time (msec)

EDF or RMS? (2)



Another example of real-time scheduling with RMS and EDF

EDF or RMS? (3)

- RMS “accomodates” task set with less utilization

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 0.7$$

  - (recall: for EDF that is up to 1)

- RMS is often used in practice;
  - main reason: stability is easier to meet with RMS; priorities are static, hence, under transient period with deadline-misses, critical tasks can be “saved” by being assigned higher (static) priorities
  - it is ok for combinations of hard and soft RT tasks

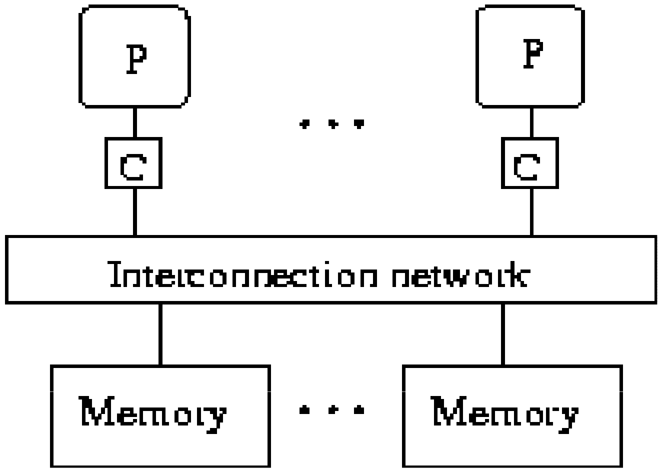
Multiprocessor Systems

Scheduling

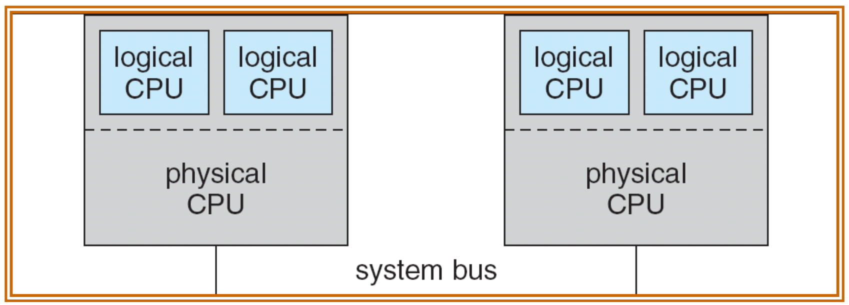
Multiprocessors

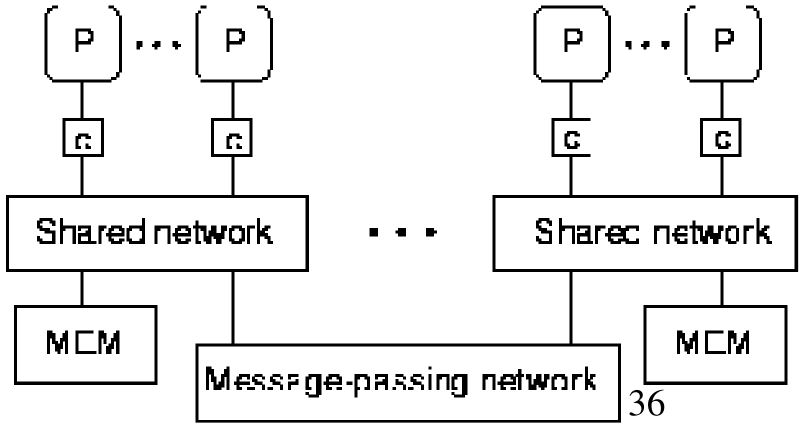
Recall from overview:

- Memory interconnection; uniform/non-uniform access



- Hyperthreading



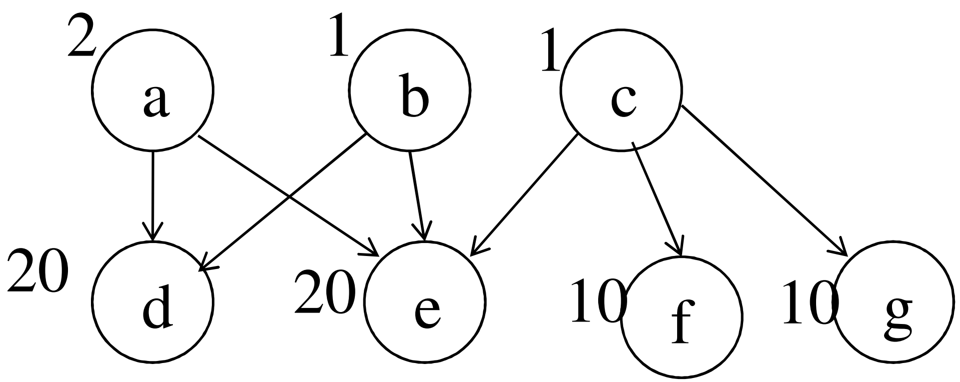


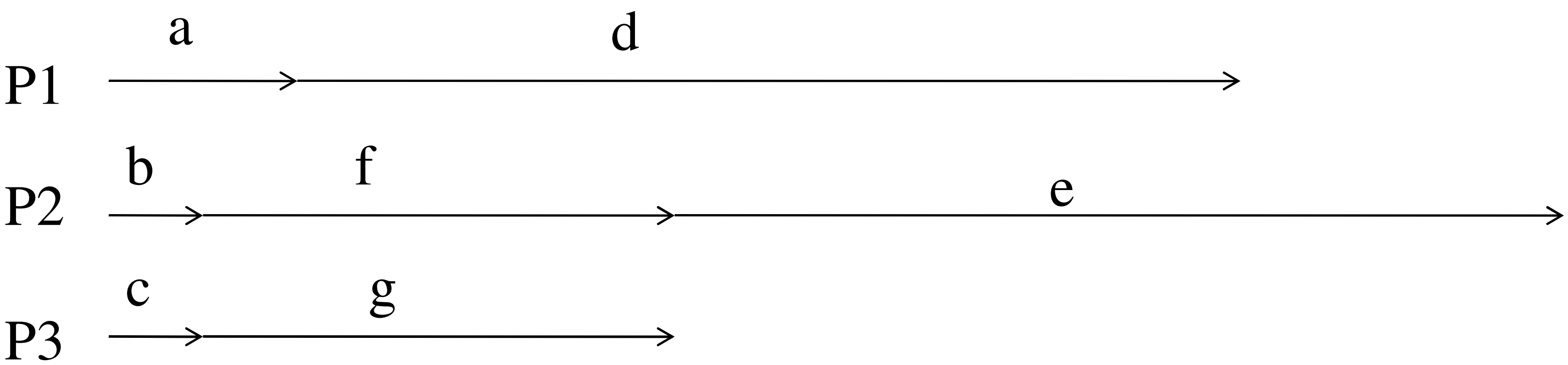
Scheduling in Multiprocessors

Different degrees of parallelism

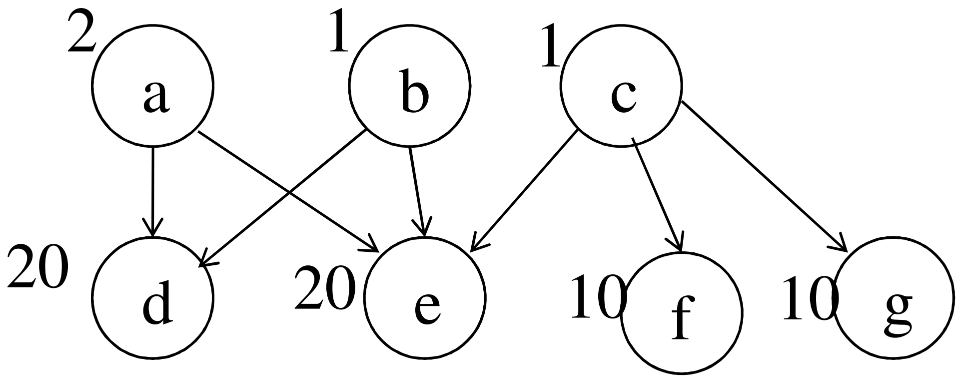
- Independent and Coarse-Grained Parallelism
  - no or very limited synchronization
  - can by supported on a multiprocessor with little change (and a bit of salt ☺)
- *Medium-Grained Parallelism*
  - *collection of threads; usually interact frequently*
- Fine-Grained Parallelism
  - Highly parallel applications; specialized and fragmented area

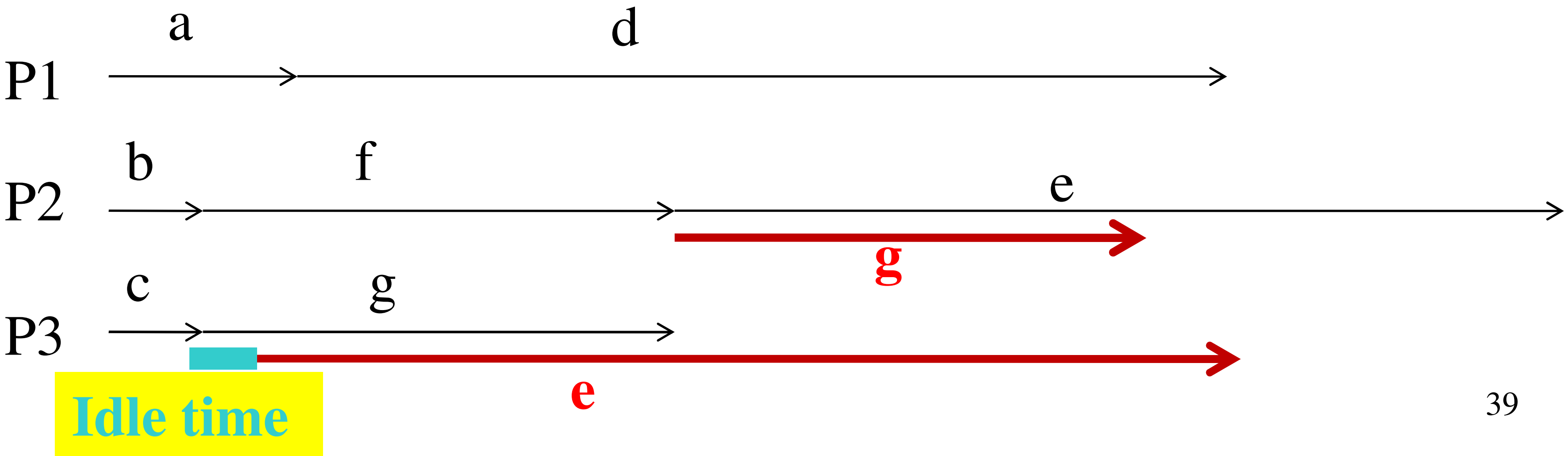
Introducing idle time can improve utilization and finishing times …





Introducing idle time can improve utilization and finishing times …

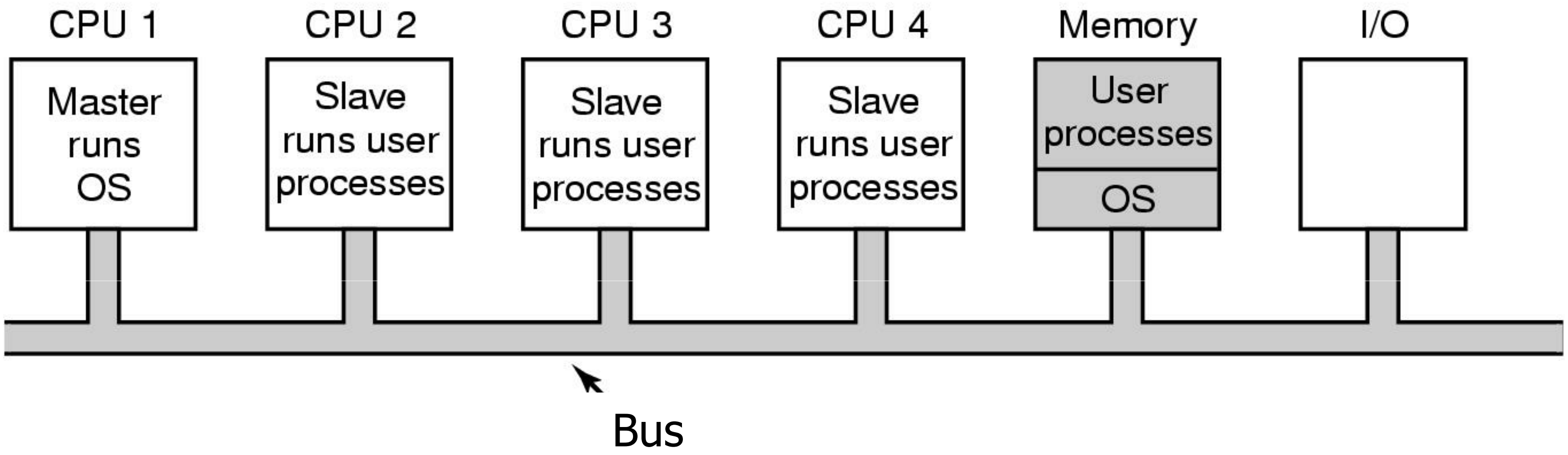




OS Design issues (1): Who executes the OS/scheduler(s)?

- Master/slave architecture: Key kernel functions always run on a particular processor
- Peer architecture: Operating system can execute on any processor
  - Each processor does self-scheduling
  - New issues for the operating system
    - Make sure two processors do not choose the same process

Master-Slave multiprocessor OS

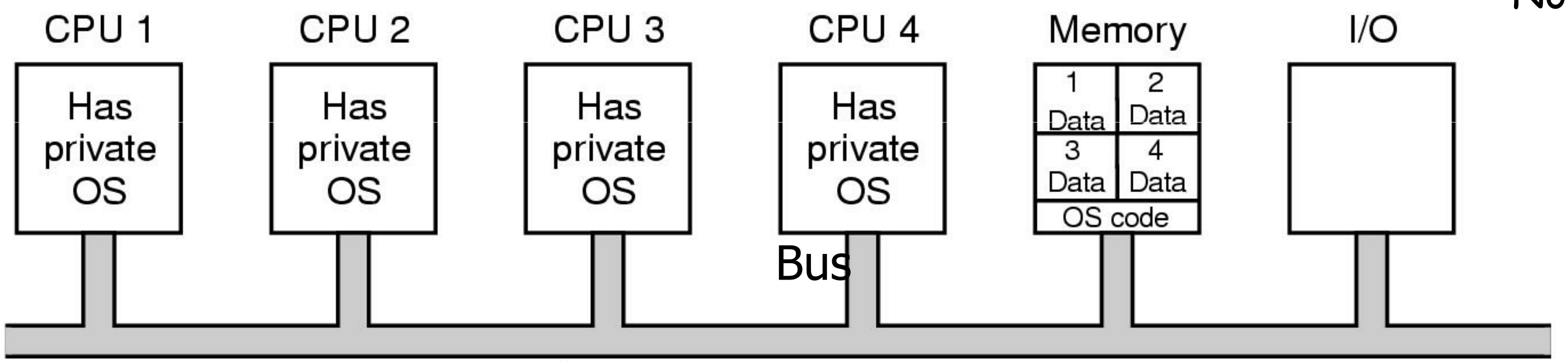


- Master/slave architecture: Key kernel functions always run on a particular processor
  - Master is responsible for scheduling; slave sends service request to the master
  - Disadvantages
    - Failure of master brings down whole system
    - Master can become a performance bottleneck

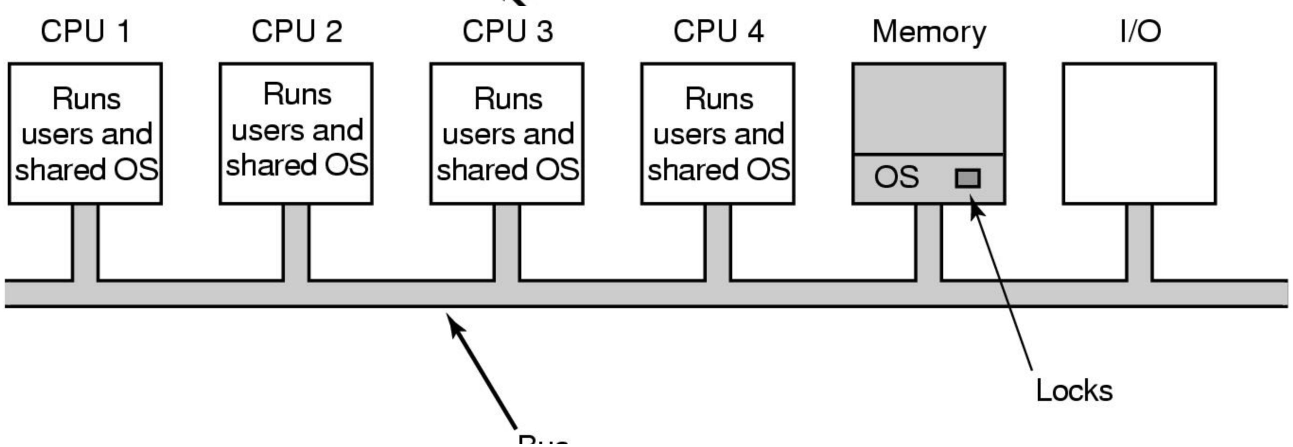
Peer Multiprocessor OS

Operating system can execute on any processor

- Each processor does self-scheduling
- operating system: Makes sure two processors do not choose the same process



Non-symmetric
:Each CPU has its own operating system

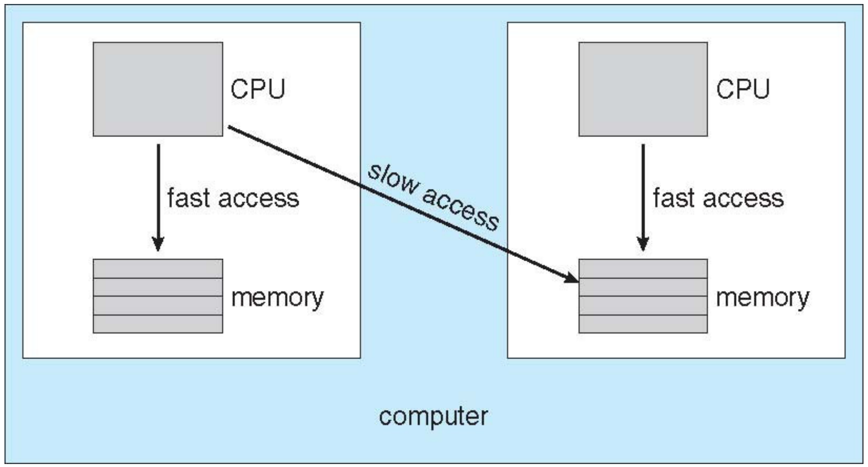


Symmetric
- SMP multiprocessor model

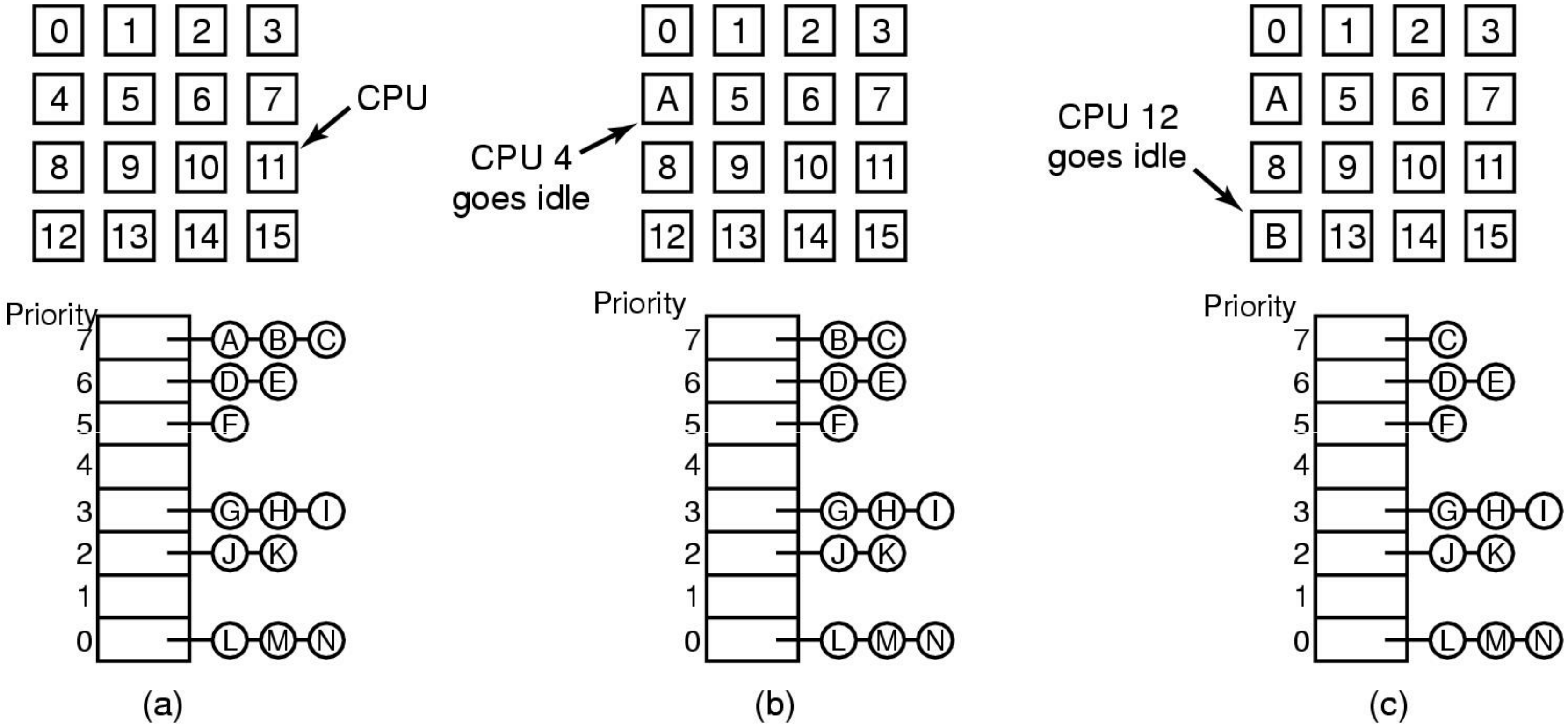
Design issues 2:
Assignment of Processes to Processors

Per-processor ready-queues vs global ready-queue

- Permanently assign process to a processor;
  - Less overhead
  - A processor could be idle while another processor has a backlog
- Have a global ready queue and schedule to any available processor
  - can become a bottleneck
  - Task migration not cheap (cf. NUMA and scheduling)

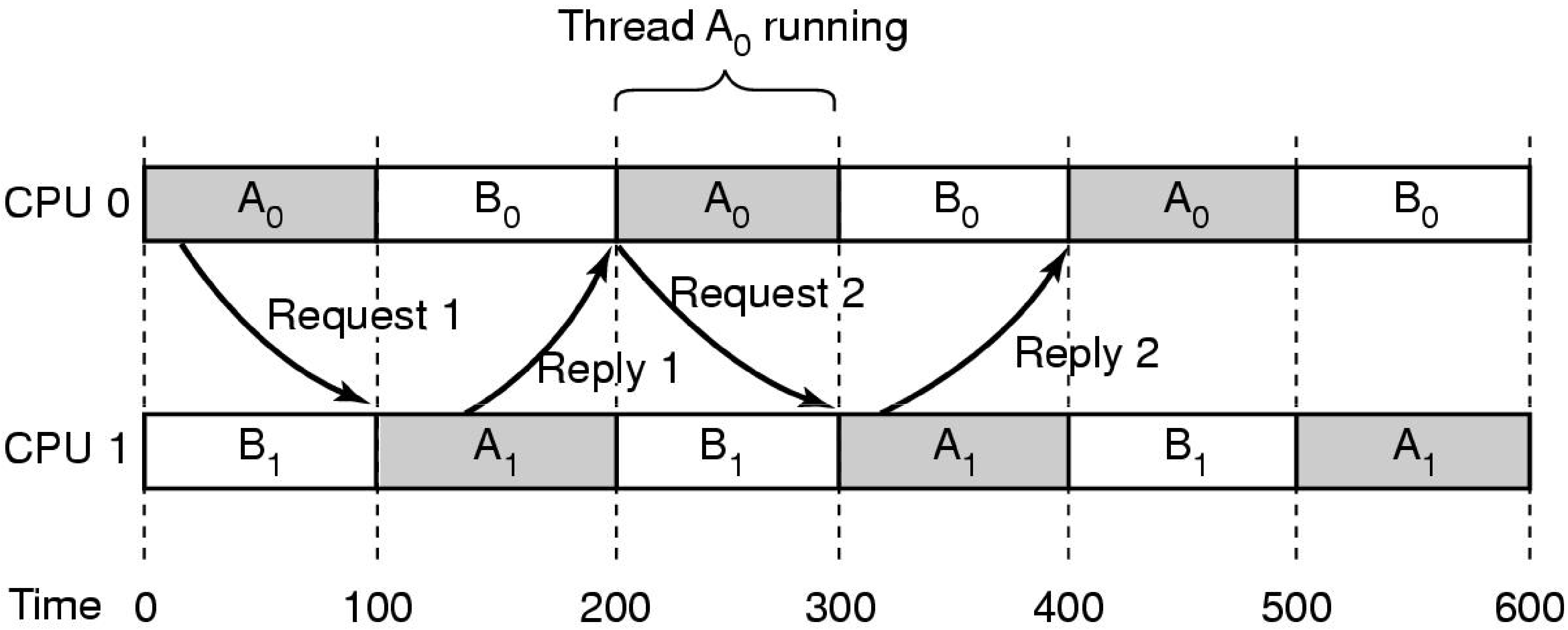


Multiprocessor Scheduling: Load sharing / Global ready queue



- (sharing time) note use of single data structure for scheduling

Multiprocessor Scheduling
Load Sharing: another problem



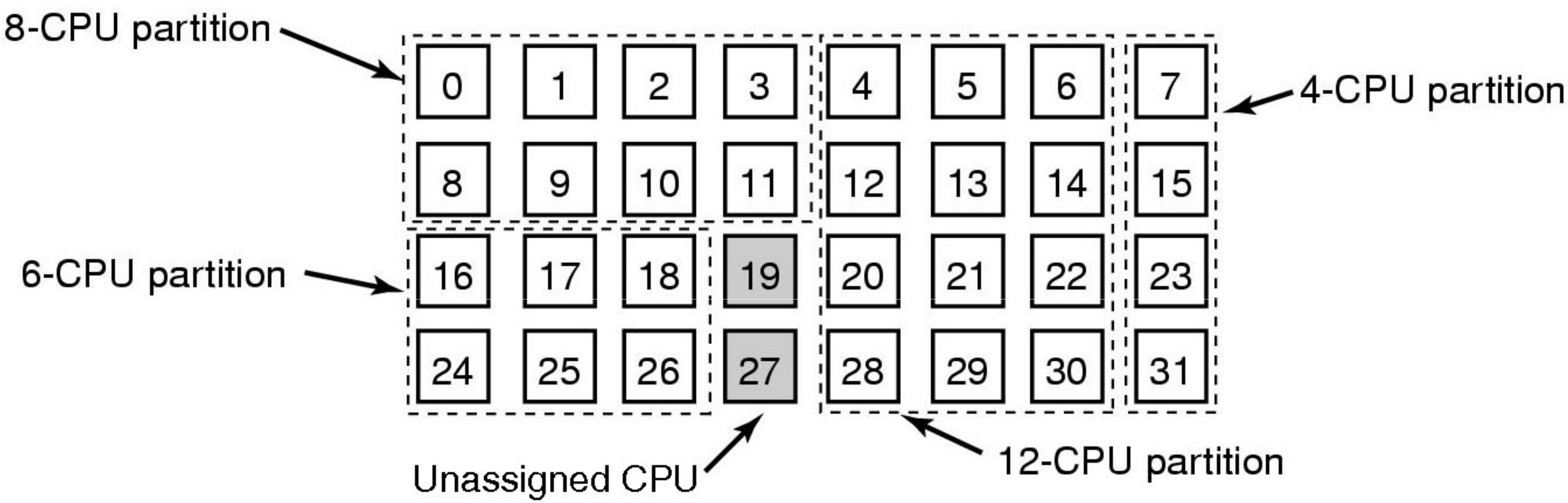
- Problem with communication between two threads
  - both belong to process A
  - both running out of phase
  - *(relates to the idle-time-adds-efficiency example)*

Design issues 3 (actually 2.5 ☺): Multiprogramming on processors?

Experience shows:

- Threads of the same process running on separate processors (to the extend of dedicating a processor to a thread) yields dramatic gains in performance
- Allocating processors to threads ~ allocating pages to processes (can use working set model?)
- Specific scheduling discipline is less important with more than on processor; the decision of "distributing" tasks is more important

Multiprocessor Scheduling: per processor or per-partition RQ



- (sharing space) - multiple threads of same process at same time across multiple CPUs

similar: Gang Scheduling

1. Groups of related threads scheduled as a unit (a gang)
2. All members of gang run simultaneously
   - on different timeshared CPUs
3. All gang members start and end time slices together

CPU

Time slot	0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Gang Scheduling: another option

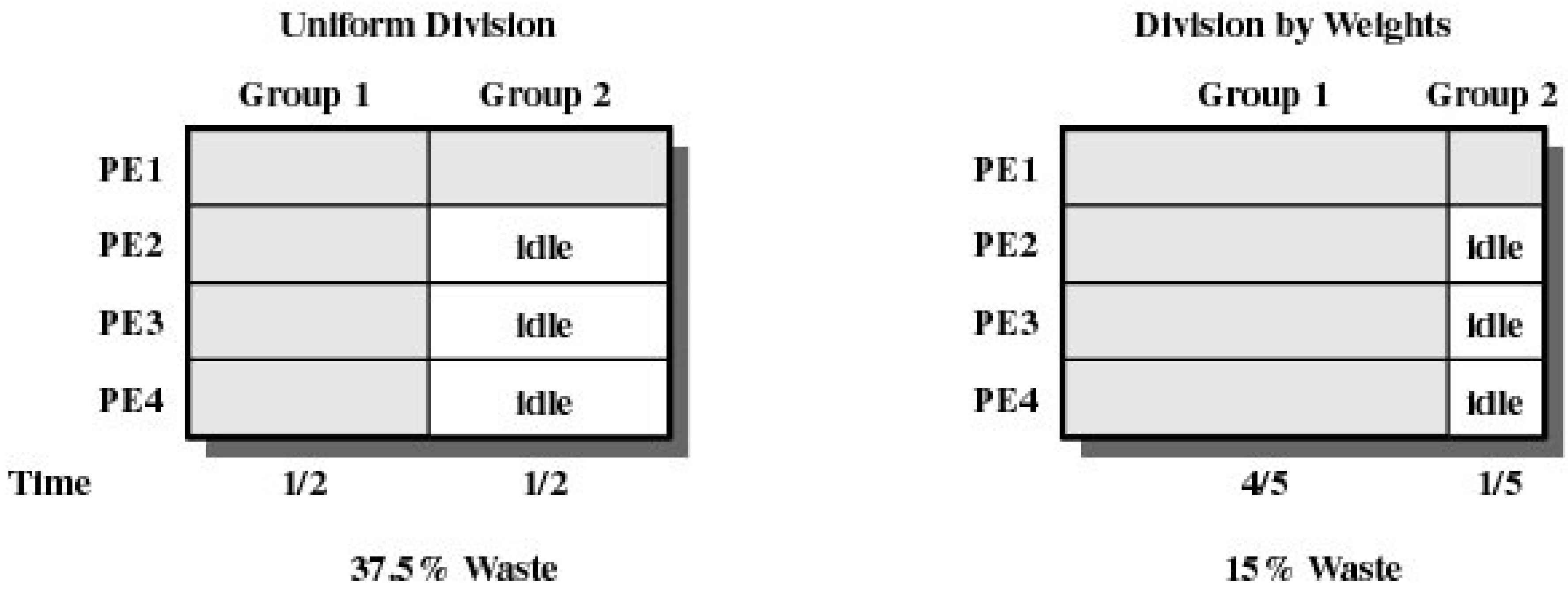
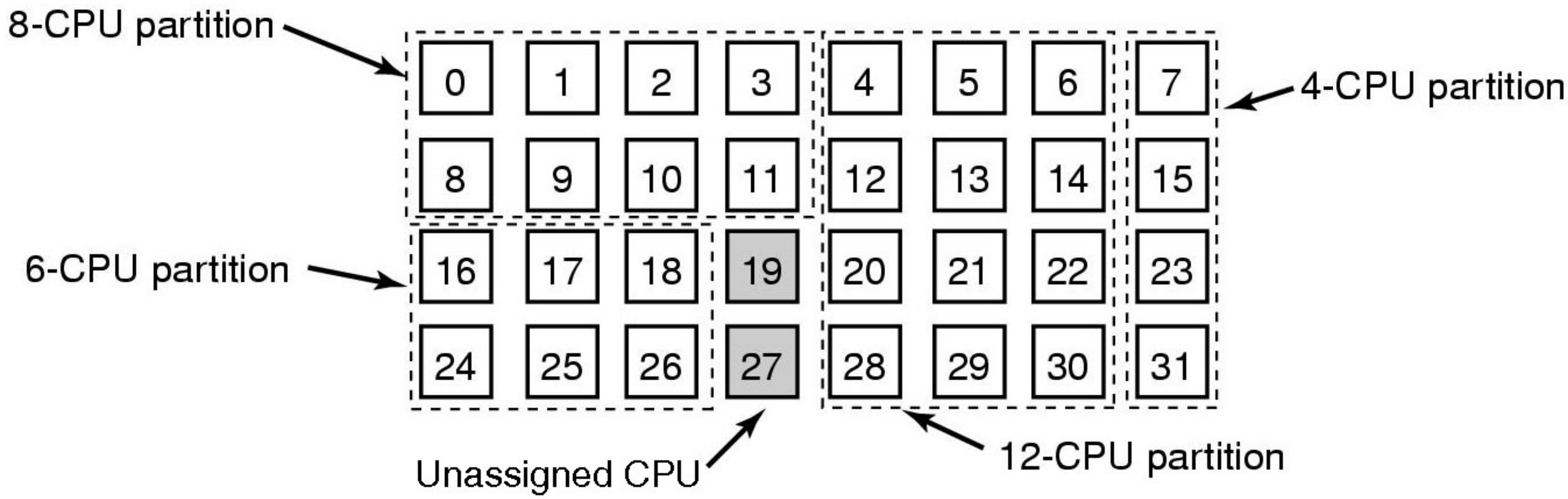


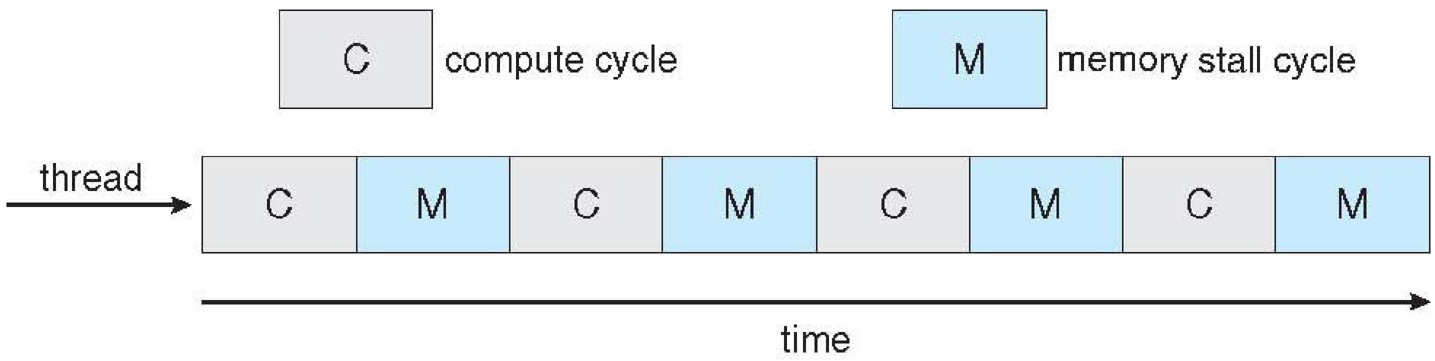
Figure 10.2 Example of Scheduling Groups with Four and One Threads [FEIT90]

Multiprocessor Thread Scheduling
Dynamic Scheduling

- Number of threads in a process are altered dynamically by the application
- Programs (through thread libraries) give info to OS to manage parallelism
  - OS adjusts the load to improve use
- Or os gives info to run-time system about available processors, to adjust # of threads (recall thread pools?).
- i.e dynamic vesion of partitioning:



Multithreaded Multicore System



Solution by architecture: hyperthreading

Needs OS awareness though to get the corresponding efficiency

Thread Scheduling

- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system
- E.g. Pthreads scheduling API allows specifying either PCS or SCS during thread creation

Summary: Multiprocessor Scheduling

Load sharing: processors/threads not assigned to particular processors

- load is distributed evenly across the processors;
- needs shared queues; may be a bottleneck

Gang scheduling: Assigns threads to particular processors (simultaneous scheduling of threads that make up a process)

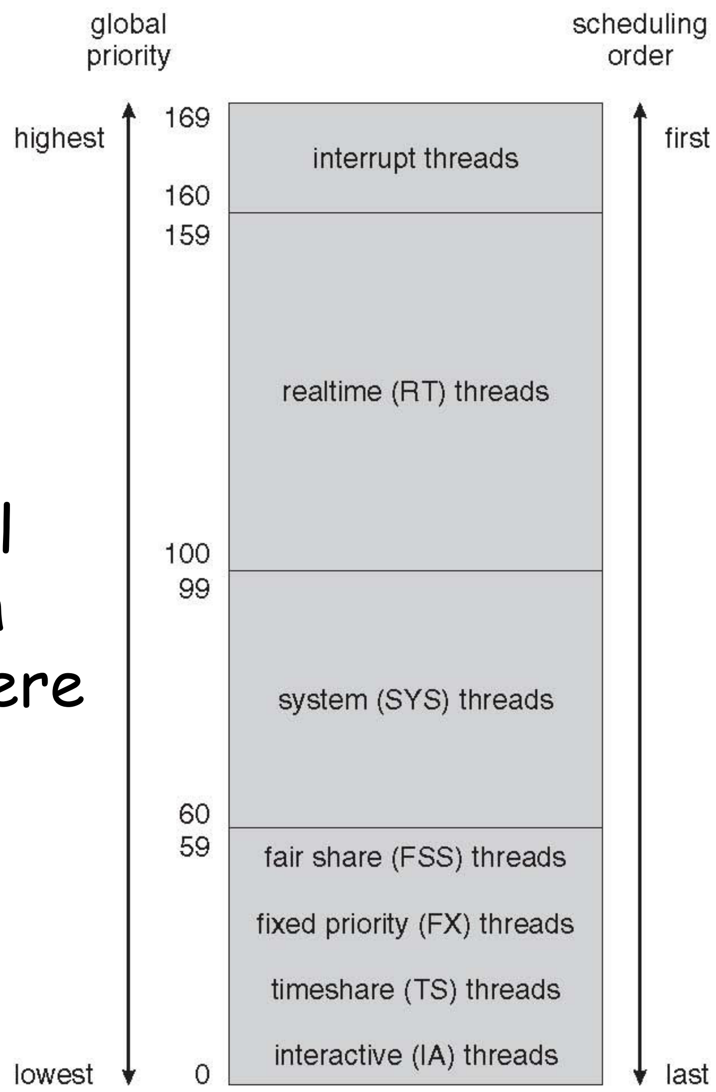
- Useful where performance severely degrades when any part of the application is not running (due to synchronization)
- Extreme version: **Dedicated processor assignment** (no multiprogramming of processors)

Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling

Solaris Scheduling

A la Multilevel Queues – with feedback, where applicable



Kernel preemptible by RT tasks in multiprocessors (unless interrupts disabled)

Solaris Dispatch Table interactive/timesharing threads

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Windows XP Priorities

Also multilevel priority queue scheduling

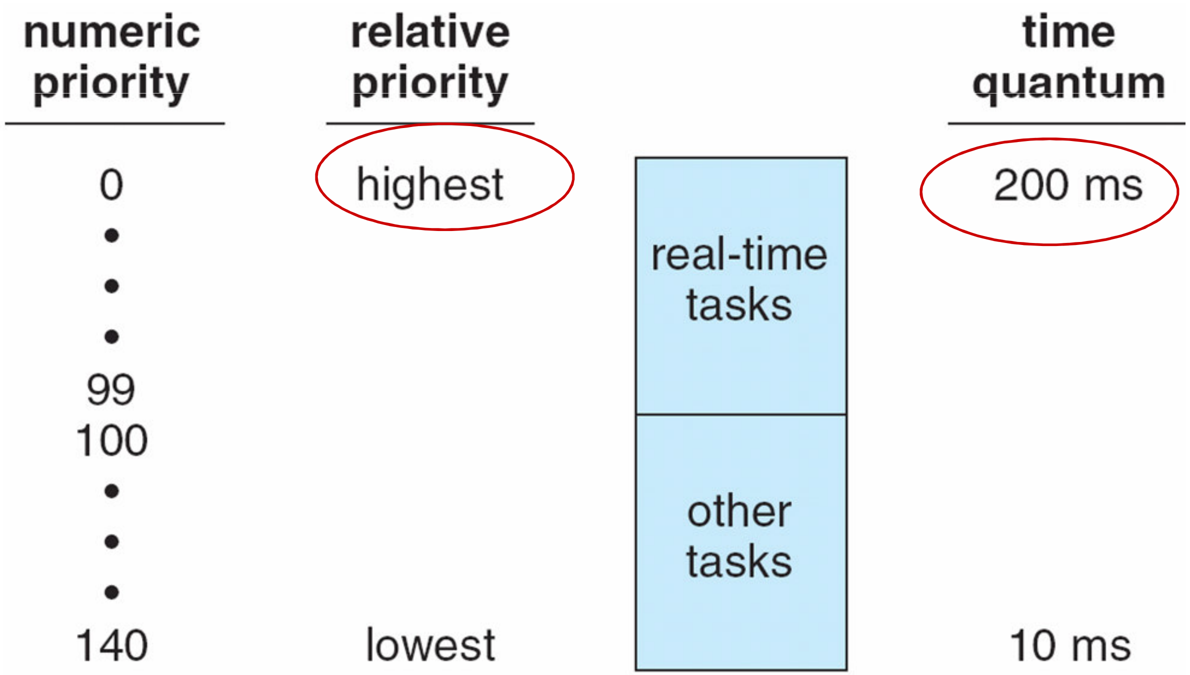
Priority classes

Relative Priority

	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

Linux Scheduling

- Two priority ranges: time-sharing and real-time
- One queue per processor/core



List of Tasks/readyQueue Indexed According to Priorities

one pair per core/processor