Chapter 5 in Old Ed:
Chapter 6 in 9th Ed:
CPU Scheduling

Pick one '***lucky***' process from ready queue

Thanks to the author of the textbook [**SGG**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.
Turgay Korkmaz

Chapter 5: CPU Scheduling

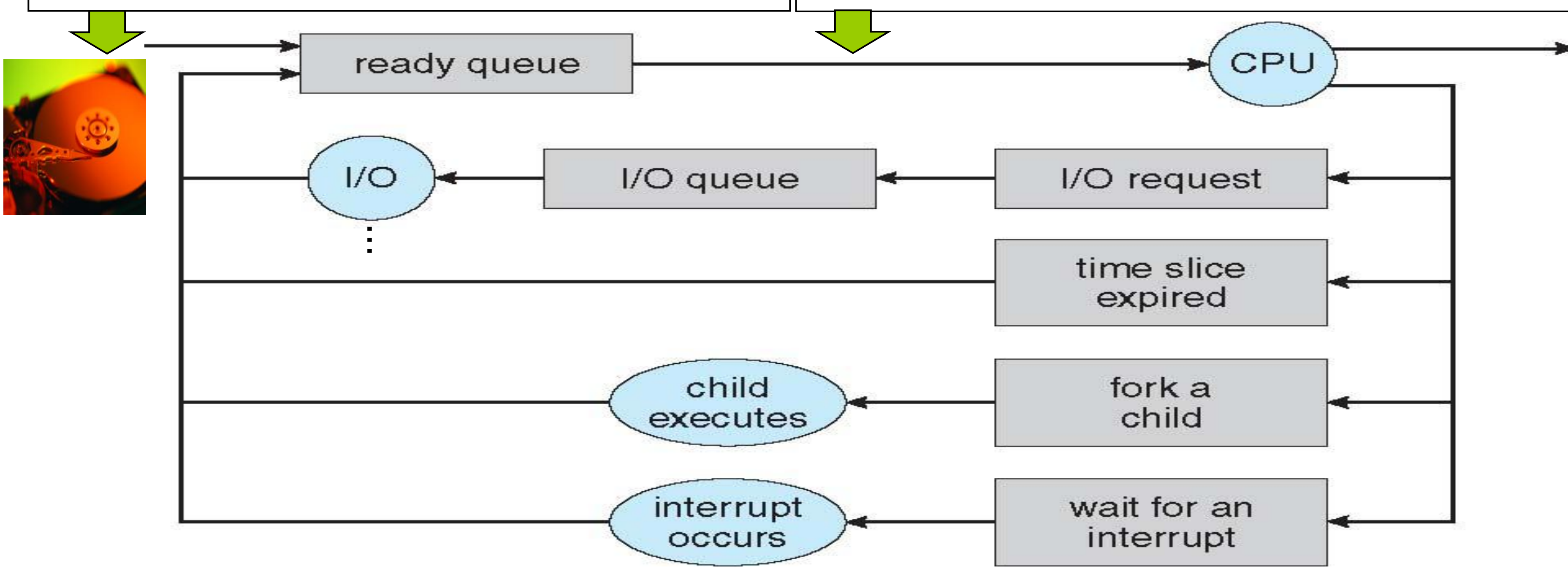
- Basic Concepts **
- Scheduling Criteria ****
- Scheduling Algorithms *****
- Multiple-Processor Scheduling ***
- Thread Scheduling ***
- Java Scheduling ***
- Algorithm Evaluation **
- Operating Systems Examples **

Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

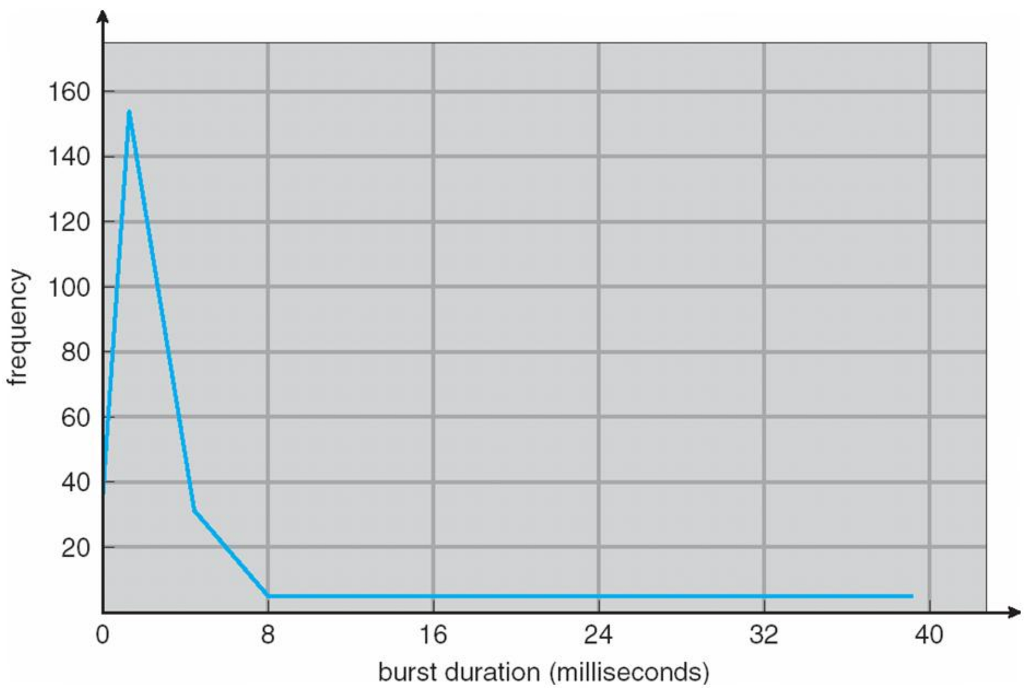
Recall "Schedulers" from Chapter 3

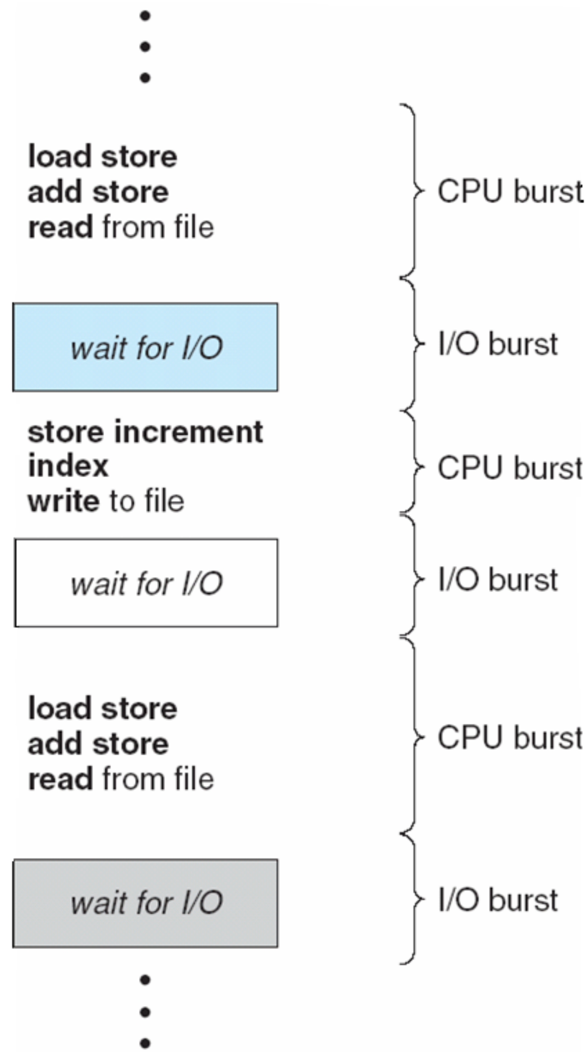
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - Less frequent
  - Controls degree of multiprogramming
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - More frequent (e.g., every 100 ms)
  - Must be fast (if it takes 10ms, then we have ~10% performance degradation)



Basic Concepts

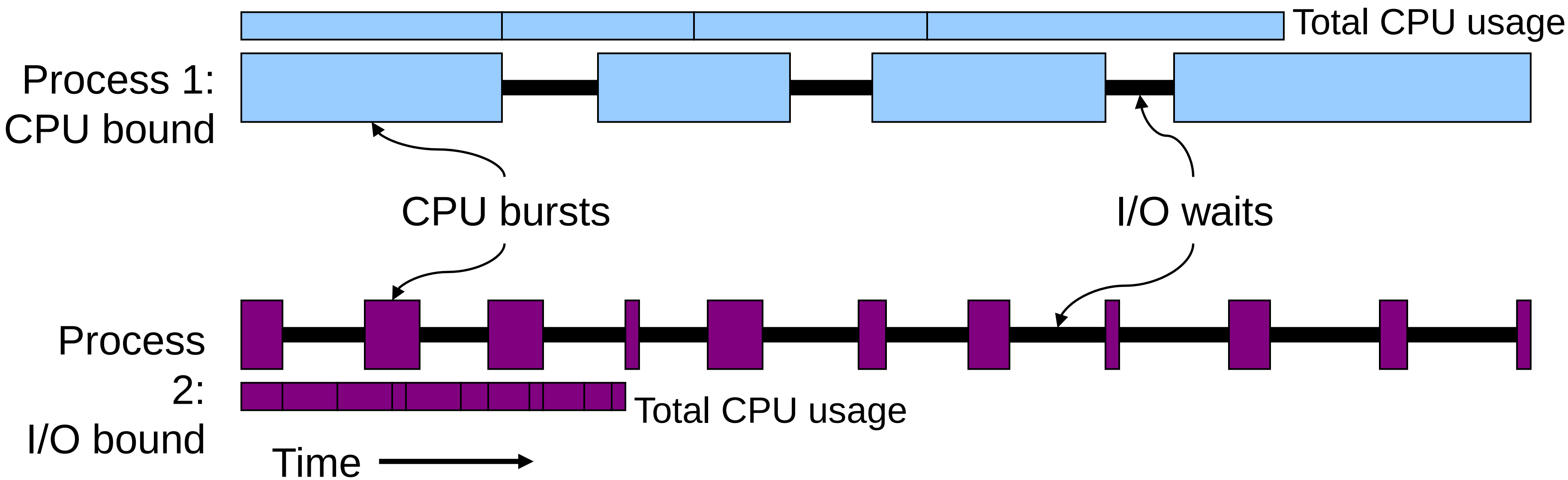
- Multiprogramming increases CPU utilization
- CPU —— I/O Burst Cycle
  - Process execution consists of a *cycle* of CPU execution and I/O wait
- **CPU burst** distribution





Basic Concepts (cont'd)

- Bursts of CPU usage alternate with periods of I/O wait
- ***CPU-bound***: high CPU utilization, interrupts are processed slowly
- ***I/O-bound***: more time is spending on requesting data than processing it



Basic Concepts (cont'd)

- **Non-preemptive** scheduling:
  - **Voluntarily give up CPU**
  - Once a process has the CPU: until it finishes or needs I/O
  - Not suitable for time-sharing
  - Only IO or process termination can cause scheduler action

- **Preemptive** scheduling
  - **Non-voluntarily give up CPU**
  - Process may be taken off CPU (e.g., quantum time expires)
  - Time-sharing systems have to be **preemptive**!

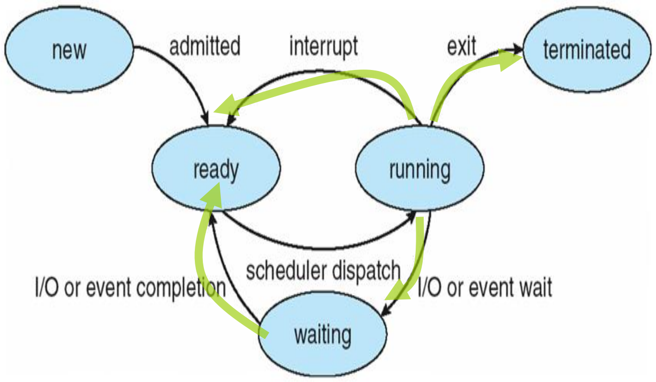
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them (short-term)

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (e.g., I/O request)
  2. Switches from running to ready state (e.g, quantum time passed)
  3. Switches from waiting to ready (e.g., I/O is complete)
  4. Terminates

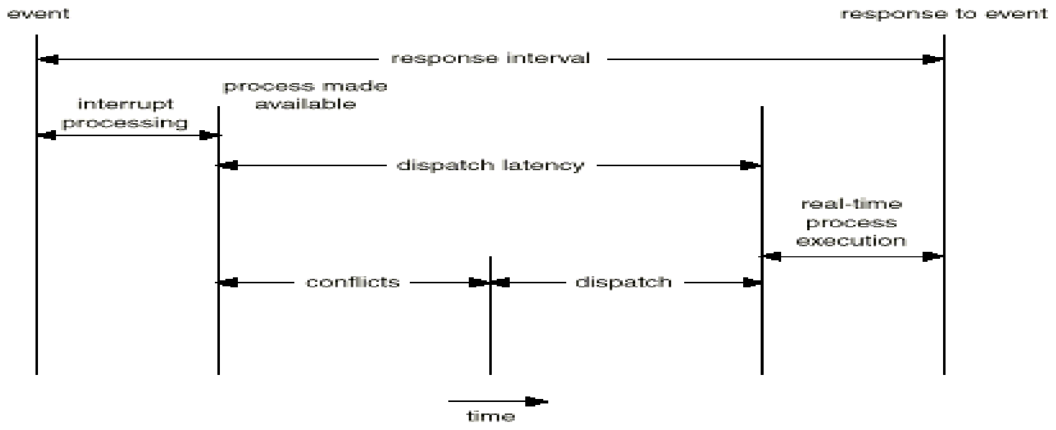
- No choice under 1 and 4 scheduling is **nonpreemptive**

- Under 2 and 3, scheduling is **preemptive**



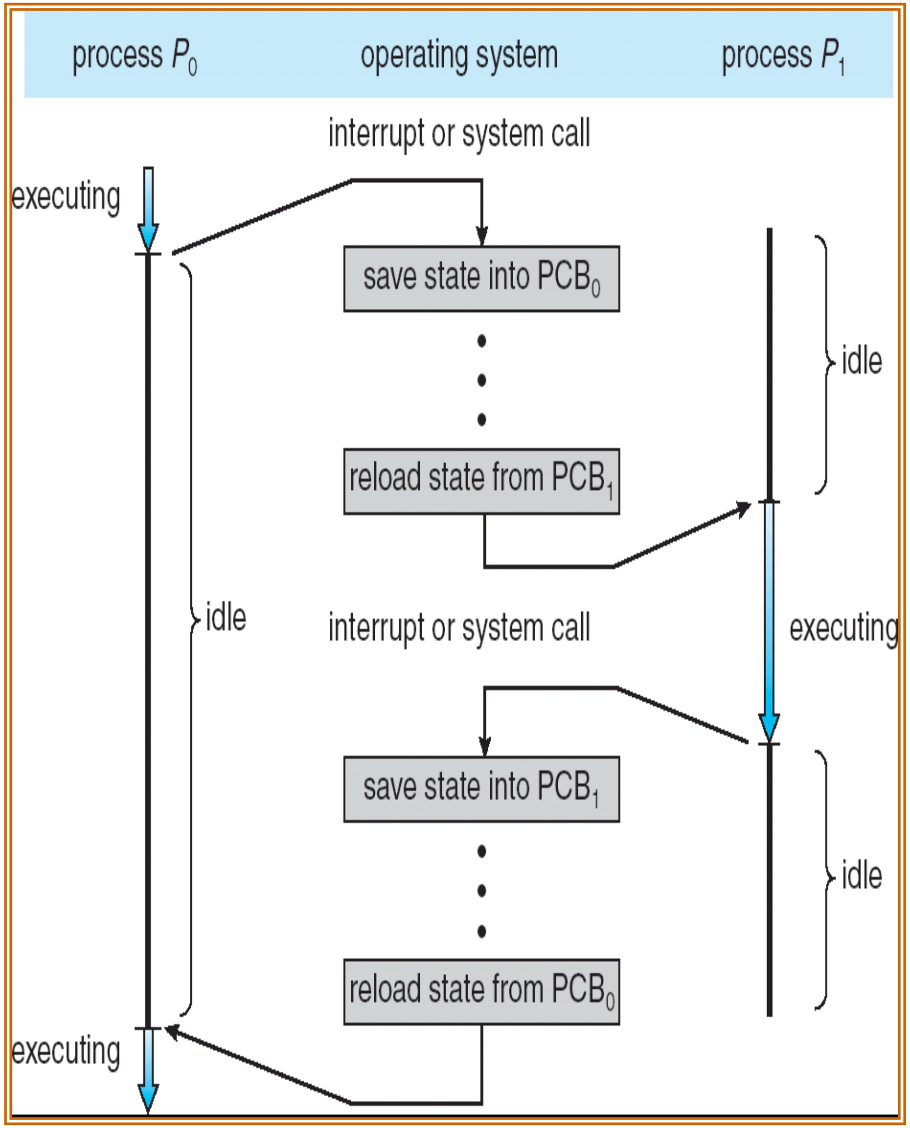
Dispatcher

- 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 (overhead)



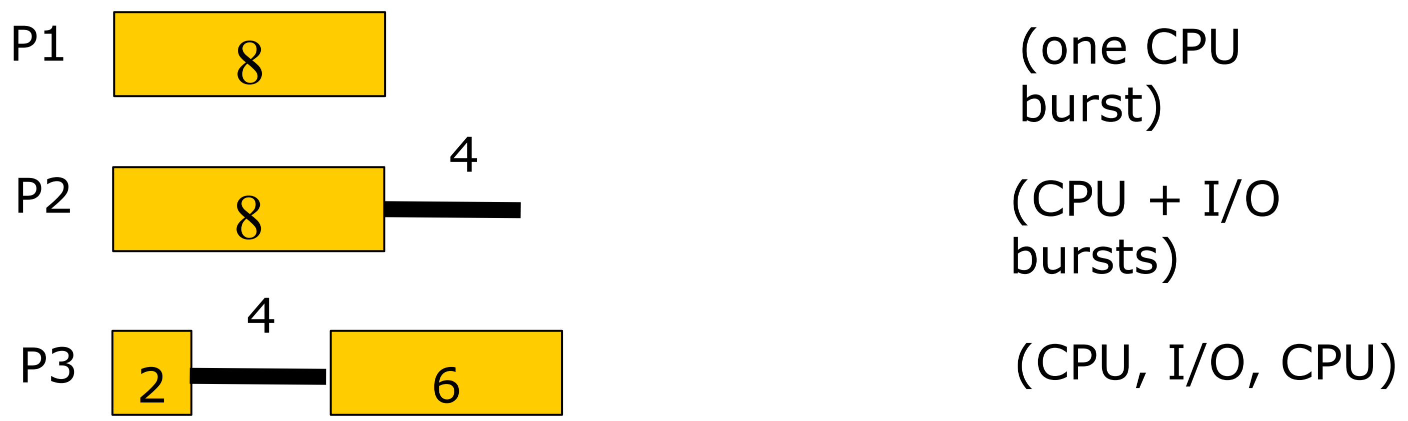
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Hardware support
  - Multiple set of registers then just change pointers
- Other performance issues/problems
  - Cache content: locality is lost
  - TLB content: may need to flush



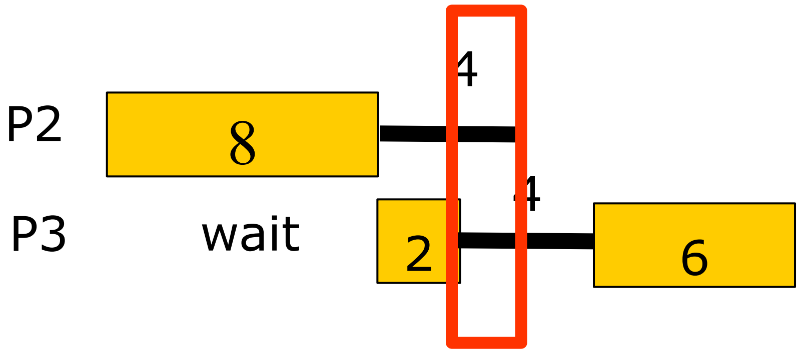
Representation of Process

- Model of Process
  - Cycle of (interleaving) CPU and I/O operations
- CPU bursts
  - The amount of time the process uses CPU before it is no longer ready
- I/O bursts: time to use I/O devices



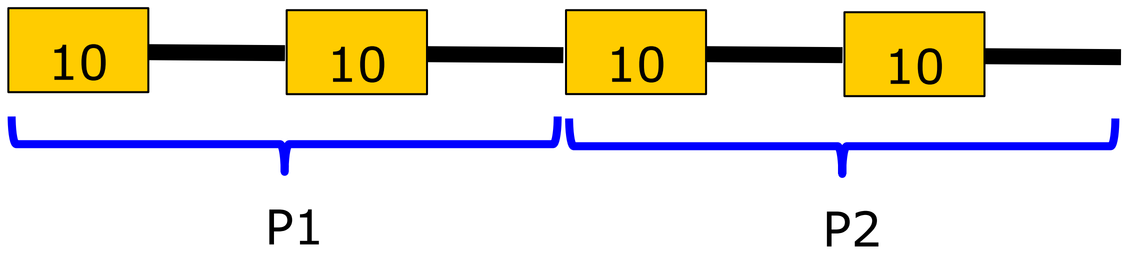
Models/Assumptions for CPU Scheduling

- CPU model
  - By default, assume **only a single CPU core**
  - **Exclusive use of CPU**: only one process can use CPU
- I/O model
  - Multiple I/O devices
  - Processes can access/request different I/O devices
  - I/O **operation time** of different processes can overlap



An Example: No Multiprogramming

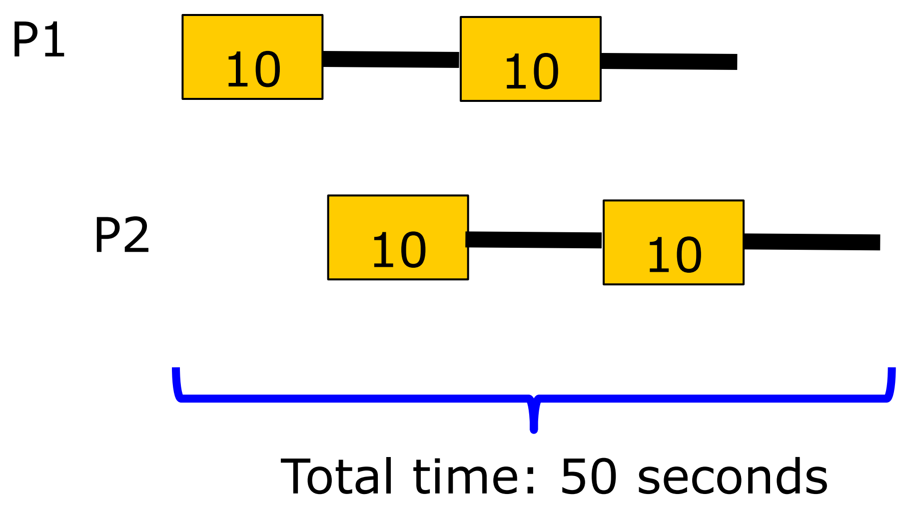
- Suppose 2 processes, where each process
  - Require 20 seconds of CPU time
  - Wait 10 second for I/O for every 10 seconds execution
- Without multiprogramming: runs one after another
  - Each takes 40 seconds: 20s run+**20s wait → total 80 sec**
  - **CPU utilization is about 40/80*100 = 50%**



An Example: with Multiprogramming

- Multiprogramming: both processes run **together**
  - The first process finishes in 40 seconds
  - The second process uses CPU (I/O) alternatively with first one and finishes 10 second later → **50 seconds**

> **CPU utilization is about 40/50*100 = 80%**



SCHEDULING GOALS
PERFORMANCE CRITERIA

Scheduling Goals

- Select the process that should be executed next
- All systems
  - **Fairness**: give each process a fair share of the CPU
  - **Balance**: keep all parts of the system busy; CPU vs. I/O
  - Enforcement: ensure that the stated policy is carried out
- Batch systems
  - **Throughput**: maximize jobs per unit time (hour)
  - Turnaround time: minimize time users wait for jobs
  - CPU utilization: CPU time is precious → keep the CPU as busy as possible
- Interactive systems (time sharing)
  - **Response/wait time**: respond quickly to users' requests
  - Proportionality: meet users' expectations
- Real-time systems: correct and in time processing
  - Meet **deadlines:** deadline miss → system failure!
  - Hard real-time vs. soft real-time: aviation control system vs. DVD player
  - Predictability: timing behaviors is predictable

Scheduling Criteria

- **CPU utilization**
  - What percent of the time the CPU is to run programs?
  - `util= (`t_{total} − t_{idle} − $t_{dispatch}$`) / `t_{total}
- **Throughput**
  - Number of processes that complete their execution per time unit
- **Turnaround time**
  - Amount of time to execute a particular process
- **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, not output (for time-sharing environment)

Usually NOT possible to optimize for *all* metrics with the same scheduling algorithm

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

Calculate total, wait, response times

- Given a process
  - Arrival time: t_a
  - First response time: t_r
  - Finish time: t_f
  - Total CPU burst time: t_{cpu}
  - Total I/O time: t_{io}
- **Turnaround time:** the process spent in the system
  - $T_{turn\_arround} = t_f - t_a = t_{cpu} + t_{io} + t_{wait}$
- **Waiting time:** the process spent in the ready queue
  - $t_{wait} = (T_{turn\_arround} - t_{cpu} - t_{io})$
- **Response time:** the process waited until the first response
  - $t_{response} = t_r - t_a$

SCHEDULING ALGORITHMS

Deciding which of the processes in the ready queue is to be selected.

FIFO	(First In First Out)	: non-preemptive, based on arrival time
SJF	(Shortest Job First)	: preemptive & non-preemptive
PR	(PRiority-based)	: preemptive & non-preemptive
RR	(Round-Robin)	: preemptive

Scheduling Policy Vs. Mechanism

- Separate what ***may*** be done from ***how*** it is done
  - Policy sets what priorities are assigned to processes
  - Mechanism allows
    - Priorities to be assigned to processes
    - CPU to select processes with high priorities
- Scheduling algorithm parameterized
  - Mechanism in the kernel
  - Priorities assigned in the kernel or by users
- Parameters may be set by user processes
  - Don’t allow a user process to take over the system!
  - Allow a user process to voluntarily lower its own priority
  - Allow a user process to assign priority to its threads

Classical Scheduling Algorithms

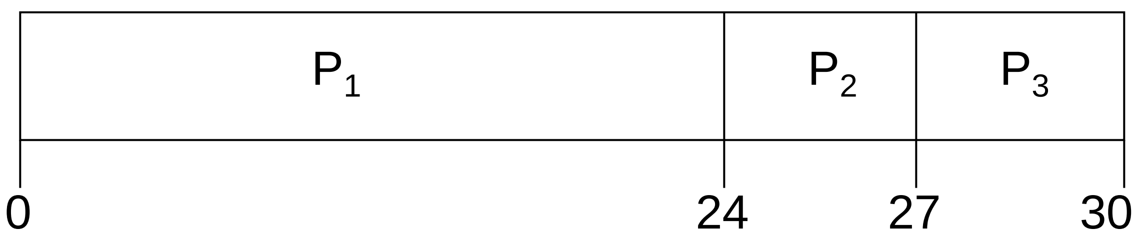
- **FIFO** or **FCFS** *: non-preemptive, based on arrival time*
  - Long jobs delay everyone else
- **SJF** *: preemptive & non-preemptive*
  - Optimal in term of waiting time
- **PR** *: preemptive & non-preemptive*
  - Real-time systems: earliest deadline first (EDF)
- **RR** *: preemptive*
  - Processes take turns with fixed time quantum e.g., 10ms
- Multi-level queue (priority classes)
  - System processes > faculty processes > student processes
- Multi-level feedback queues: change queues
  - short → long quantum

FIFO or First-Come, First-Served (FCFS) Scheduling

Suppose the following processes arrive at time t=0 in the given order

Process	Burst Time
P_1	24
P_2	3
P_3	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

Problem: long jobs delay every job after them.
Many processes may wait for a single long job.

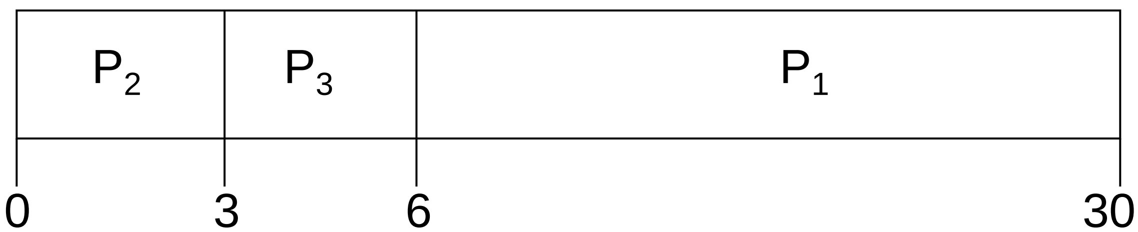
- **CPU utilization** : What percent of the time the CPU is used
- **Throughput** : Number of processes that complete their execution per time unit
- **Turnaround time :** Amount of time to execute a particular process
- **Waiting time:** Amount of time a process has been waiting in the ready queue
- **Response time :** Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

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

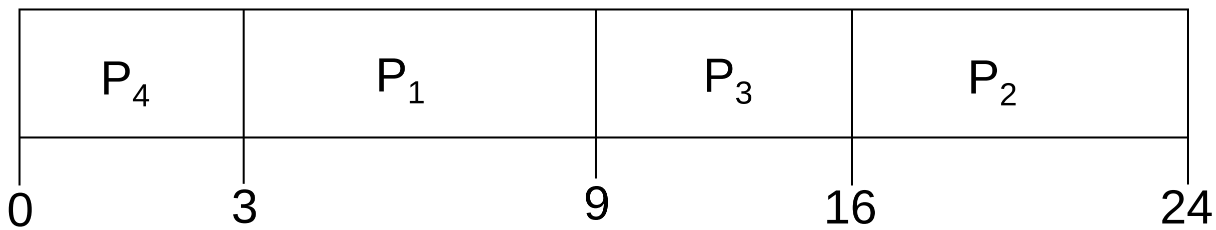
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time
- SJF is **optimal**
  - gives minimum average waiting time for a given set of processes
- The difficulty is how to know the length of the next CPU request
  - Long term schedulers might use it based on program size, but
  - Short-term schedulers cannot use this; but, they may try to predict it by averaging previous CPU burst times

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Exercise: SJF

Process	Arrival Time	Burst Time
P_1	0	6
P_2	1	8
P_3	2	7
P_4	3	3

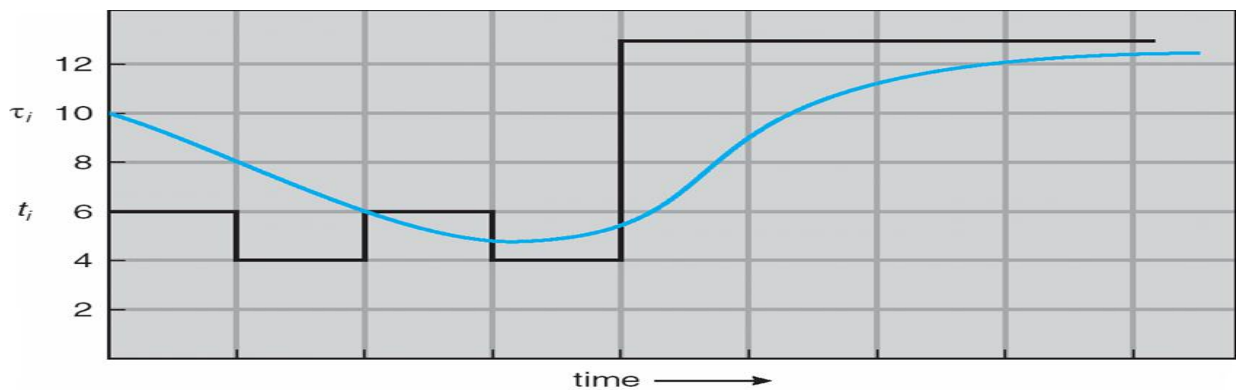
- Give **Gantt chart** under both **preemptive** and **nonpreemptive** SJF scheduling:

- Compute Average waiting time?

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. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define: $\tau_{n=1} = \alpha\, t_n + (1-\alpha)\tau_n.$



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

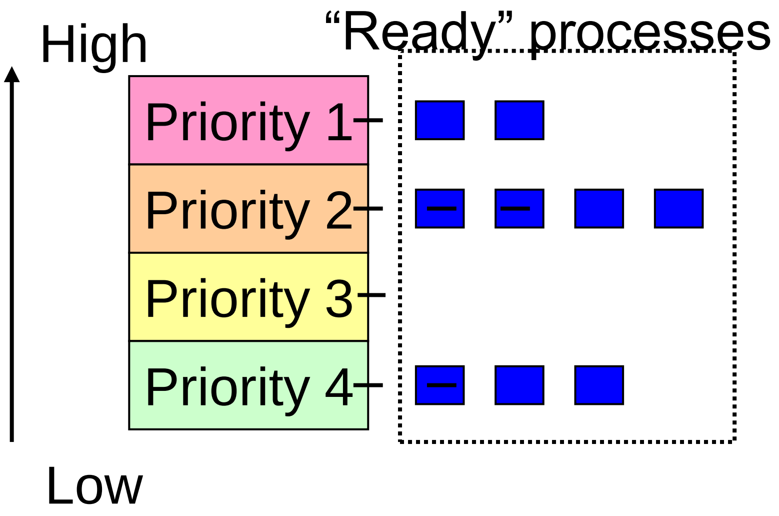
- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

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

Priority (PR) 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)
  - Preemptive
  - Nonpreemptive

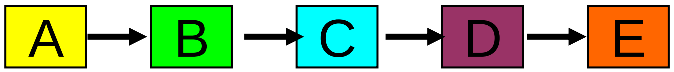


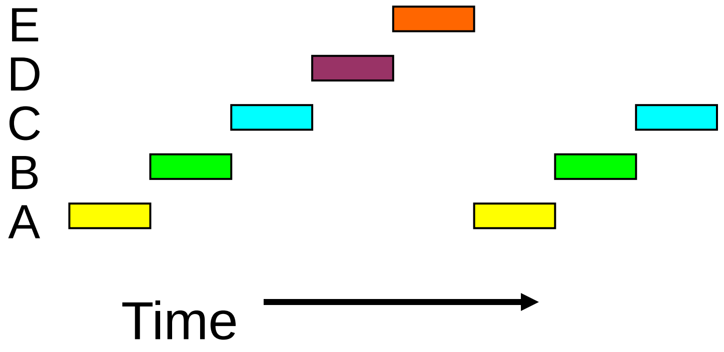
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

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 ⇒ FIFO
  - *q* small ⇒ fluid model

  q must be large (but not much) with respect to context switch; otherwise, overhead is too high





Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

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

Time Quantum and Context Switch Time

What's a good quantum?

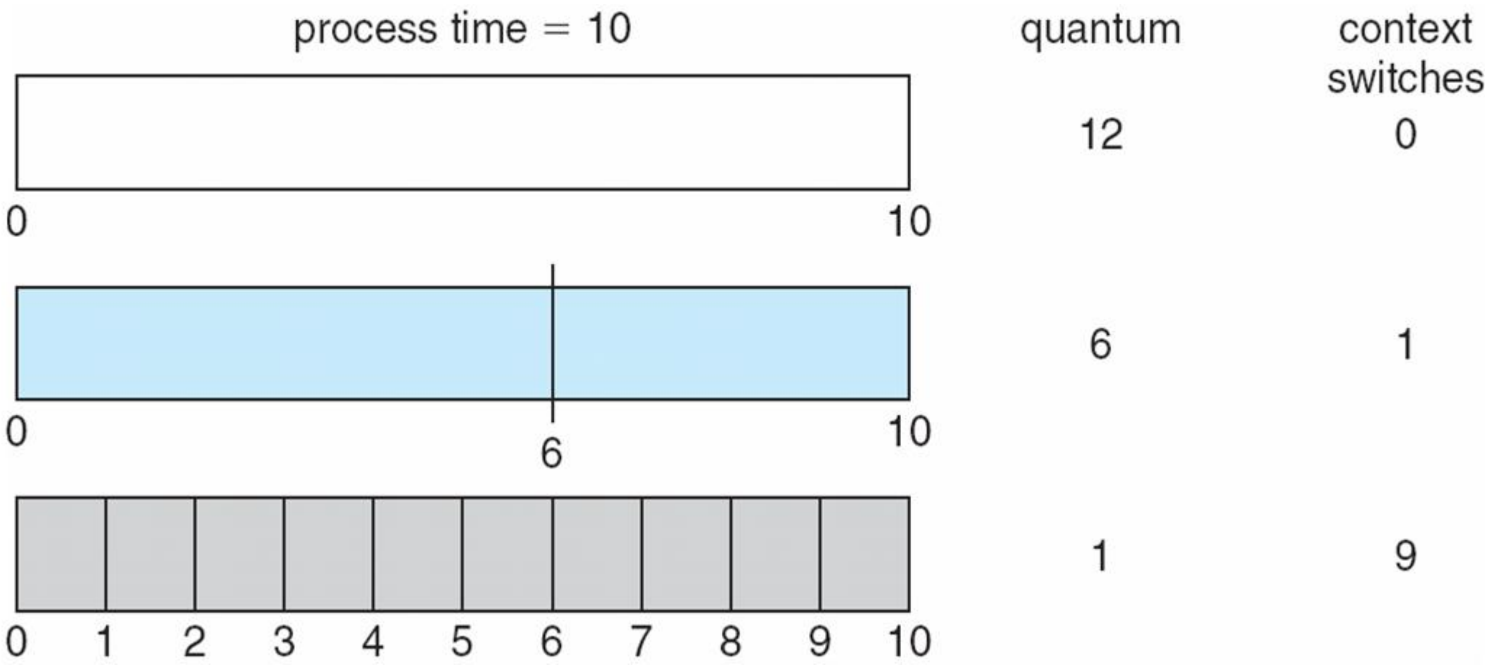
Too short:

many **context switches** hurt efficiency

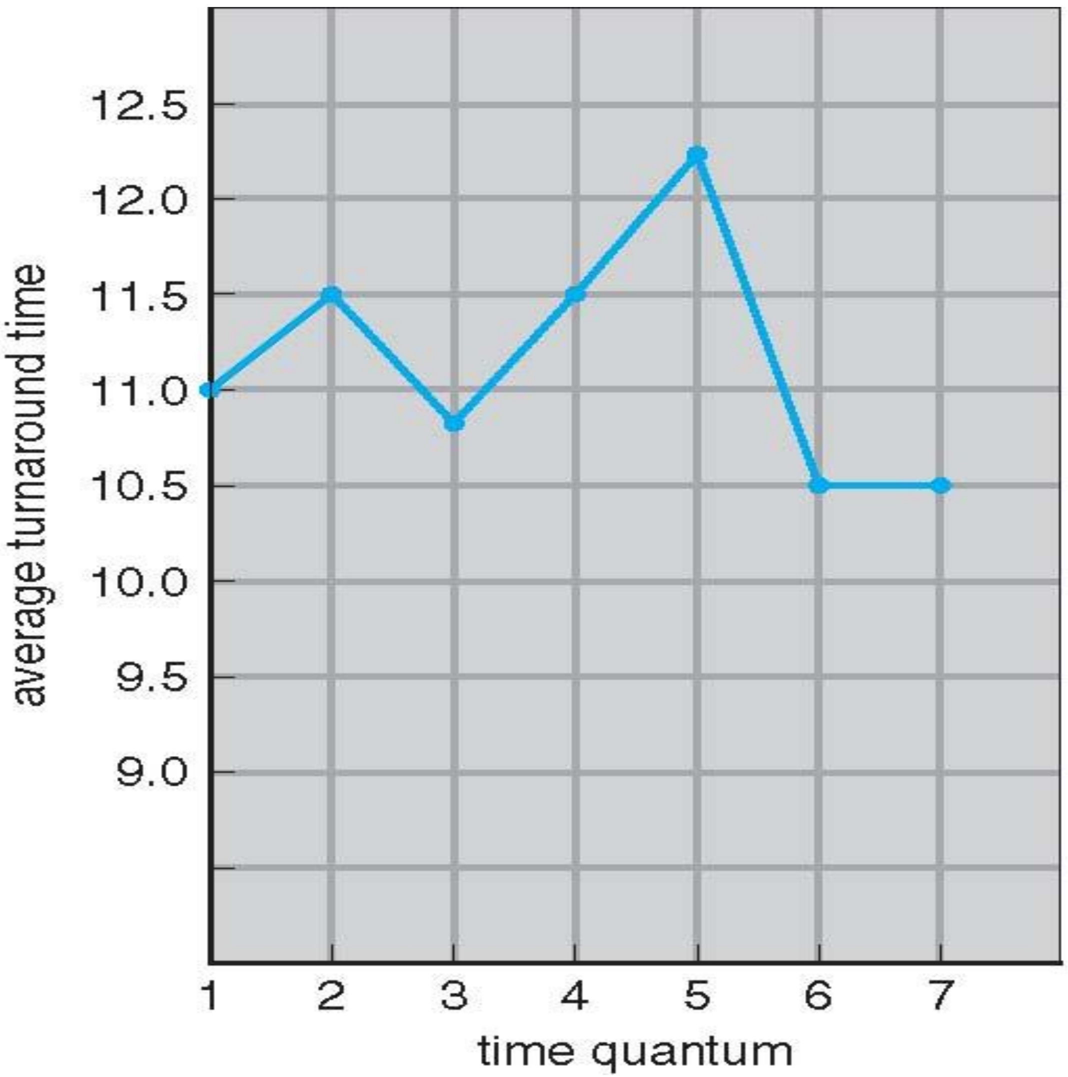
Too long:

poor response to interactive requests

Typical length: 10–50 ms

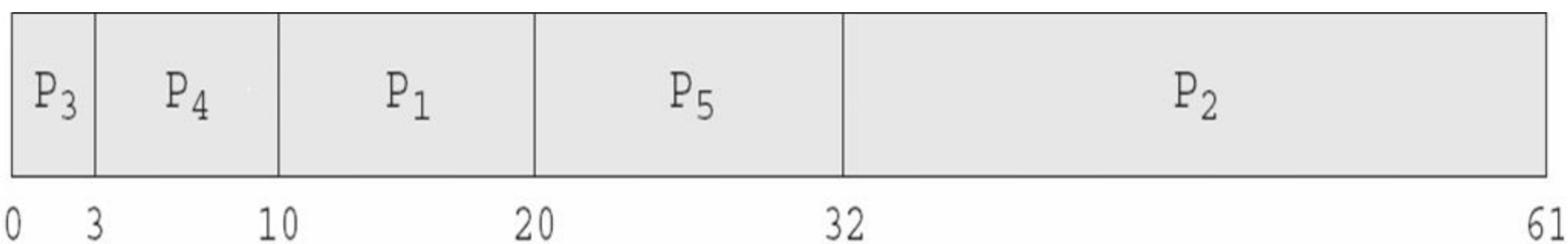


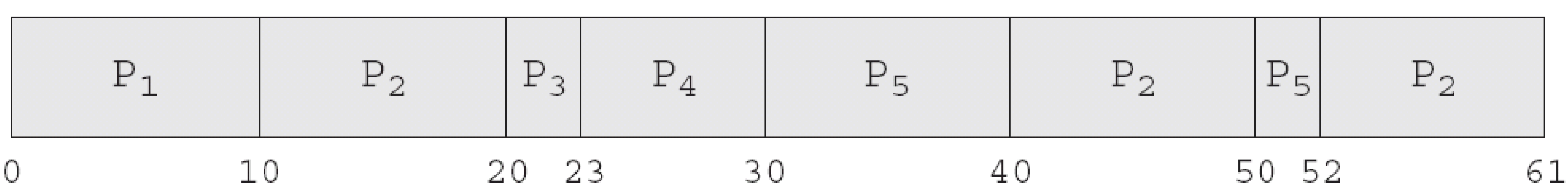
Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

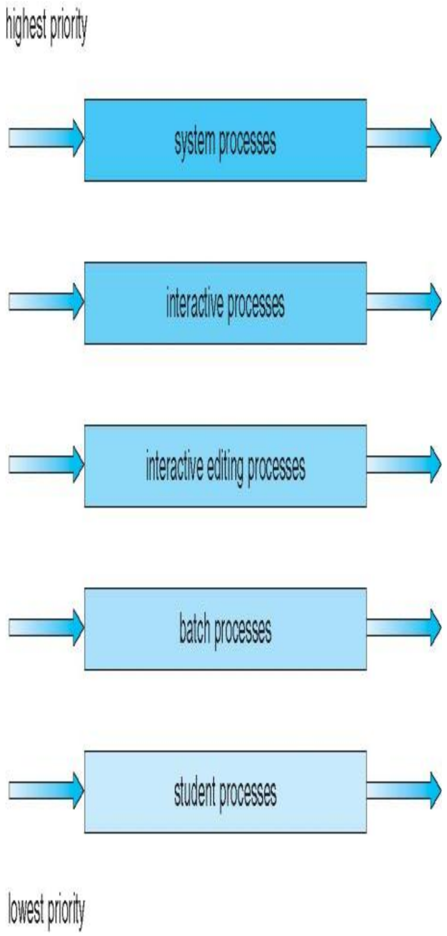
Exercise: Compute Avg waiting time





Multilevel Queue

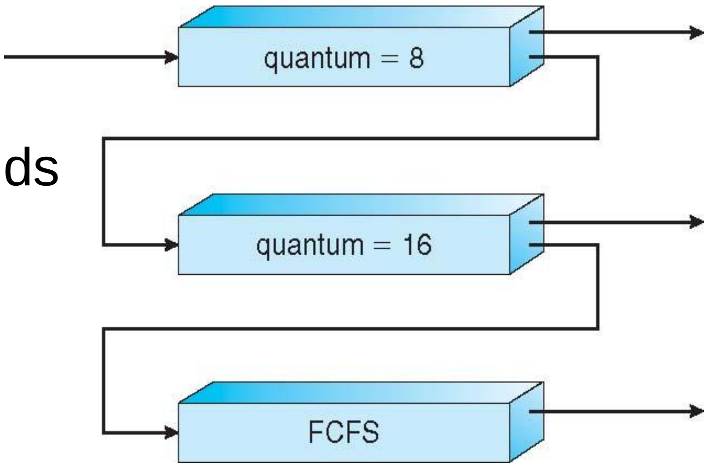
- Ready queue is partitioned into separate queues:
  foreground (interactive), **background** (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling;
    - Serve all from foreground then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



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
  - CPU bound → move into low priority queue
  - I/O bound → move into high priority queue
- 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
- Most flexible and general, but hard to configure

Example of Multilevel Feedback Queue



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR 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.
  - Processes requiring less than 8 ms will be served quickly…

Load sharing is possible

CPU scheduling will be more complex (no single best solution)

We consider **Homogeneous processors** (could be heterogeneous too)

MULTIPLE-PROCESSOR SCHEDULING

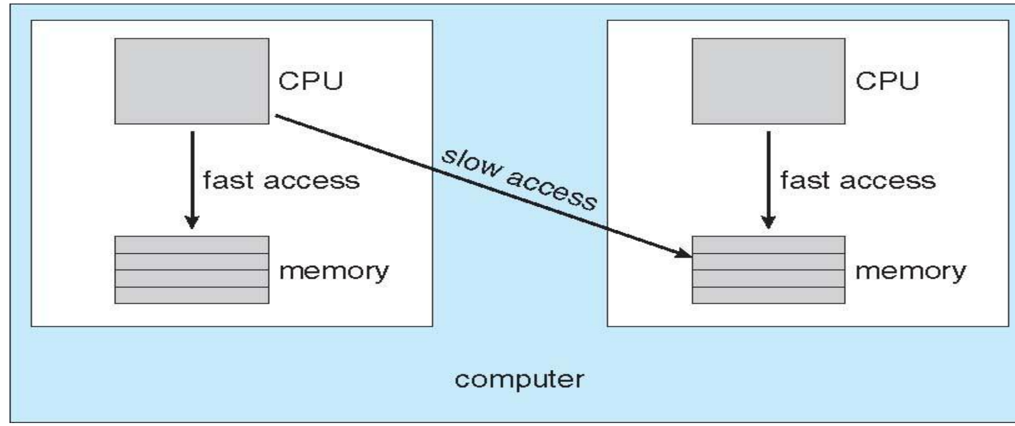
Approaches to Multiple-Processor Scheduling

- **Asymmetric multiprocessing**
  - only one processor (master) accesses the system data structures,
  - others (slaves) run user code
  - easy, but single point of failure and could be the bottleneck
- **Symmetric multiprocessing (SMP)**
  - each processor is self-scheduling,
  - all processes in common ready queue, or

    each has its own private queue of ready processes
  - Modern OSes support this

Processor Affinity

- What would happen if a process migrates to another processor?
  - Clear cash
- NUMA architecture
  - Slow access



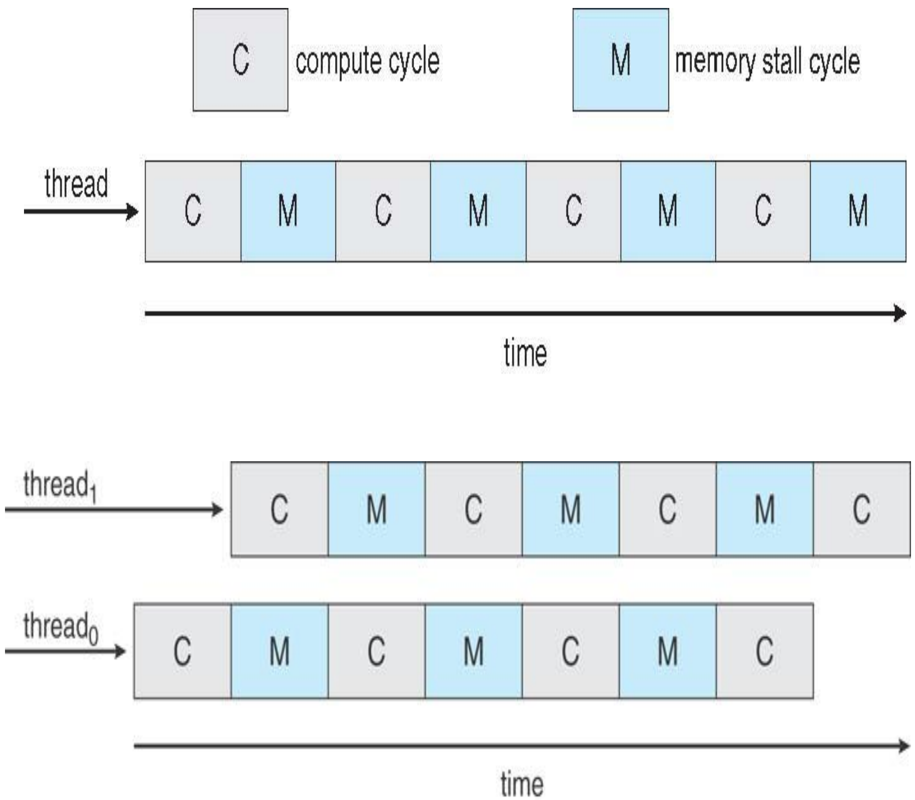
- A process has affinity for processor on which it is currently running
- This is known as Processor affinity
  - **soft affinity**
  - **hard affinity**

Load Balancing

- Keep the load evenly distributed
  - Easy in Asymmetric multiprocessing (AMP) why?
  - Hard in Symmetric multiprocessing (SMP) why?
- Two Approaches for SMP
  - **Push** migration
    - Busy CPU checks load on others, and pushes load to them
  - **Pull** migration
    - Idle CPU pulls load from others
  - Often implemented together
- Affects processor affinity
  - (have some threshold)

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



Virtualization and Scheduling

- Even a single-CPU system acts like a multiprocessor system
- Host and guest systems could have different scheduling
  - But what happens if VM allocates 100ms while host allocates 10ms???

THREAD SCHEDULING

Thread Scheduling

- Distinction between
  - **user-level** threads managed by a thread library in user space
  - **kernel-level** threads managed by OS scheduler
    - User threads must me mapped to kernel threads
- 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

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS

    schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM

    schedules threads using SCS scheduling.

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr t attr;
   pthread_attr_init(&attr);      /* get the default attributes */
         /* set the scheduling algorithm to PROCESS or SYSTEM */
   // if (pthread_attr_getscope(&attr, &scope) !=0) // error
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
         /* set the scheduling policy - FIFO, RT, or OTHER */
   pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
   for (i = 0; i < NUM THREADS; i++) /* create the threads */
         pthread_create(&tid[i],&attr, runner, NULL);
   for (i = 0; i < NUM THREADS; i++) /* now join on each thread */
         pthread_join(tid[i], NULL);
}
 /* Each thread will begin control in this function */
void *runner(void *param)
{  printf("I am a thread\n"); pthread exit(0); }
```

JAVA SCHEDULING

Java Thread Scheduling

- JVM uses **PR** Scheduling Algorithm
  - Could be preemptive or not
- If there are multiple threads with the same priority, FIFO Queue is used
- JVM does not specify whether threads are Time-Sliced or not
- JVM schedules a thread to run when:
  1. It exits its run()
  2. It blocks for I/O
  3. Its time quantum expires **(if time-sliced)**
  4. A higher priority thread enters the Runnable State **(if preemptive)**

Time-Slicing

- Since the JVM doesn't ensure Time-Slicing, the yield() method may be used to give the CPU to some other threads, called cooperative multitasking

```
while (true) {
    // perform CPU-intensive task
    . . .
    Thread.yield();
}
```

- This yields control to another thread of equal priority

Thread Priorities

Priority		Comment
Thread.MIN_PRIORITY	1	Minimum Thread Priority
Thread.NORM_PRIORITY	5	Default Thread Priority
Thread.MAX_PRIORITY	10	Maximum Thread Priority

Priorities may be set using setPriority() method:

```
Thread.currentThread().setPriority(Thread.NORM_PRIORITY + 2);
```

Rule of Thumb:

- At any given time, the highest-priority thread is running. However, this is not guaranteed.
- The thread scheduler may choose to run a lower-priority thread to avoid starvation.
- For this reason, use thread priority only to affect scheduling policy for efficiency purposes;
- **So,**

  Do *not* rely on it for algorithm correctness.

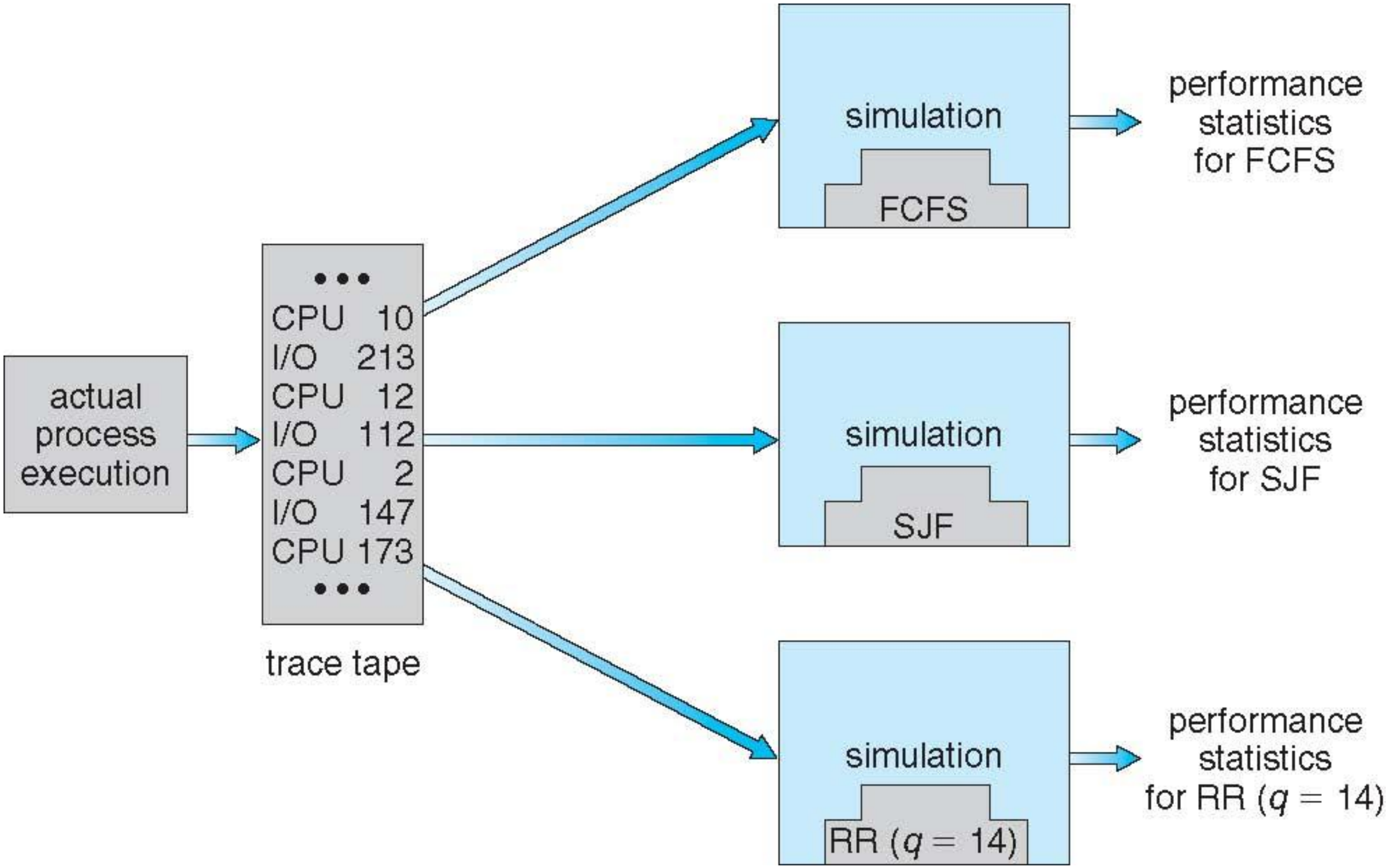
Skip the rest

PERFORMANCE EVALUATION

Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
  - Little's Formula $n = \lambda \times W$
    - n: average queue length,
    - W: average wait time
    - λ : average arrival rate
- Simulation
  - Random load
- Implementation

Evaluation of CPU schedulers by Simulation



Solaris scheduling

Windows XP scheduling

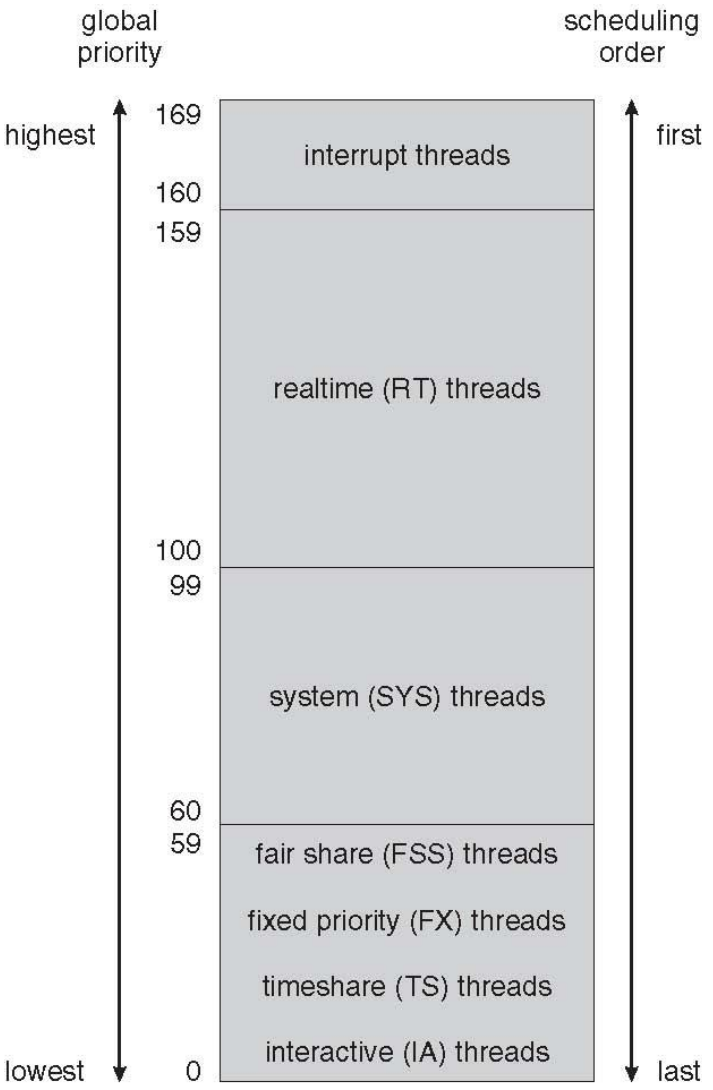
Linux scheduling

OPERATING SYSTEM EXAMPLES

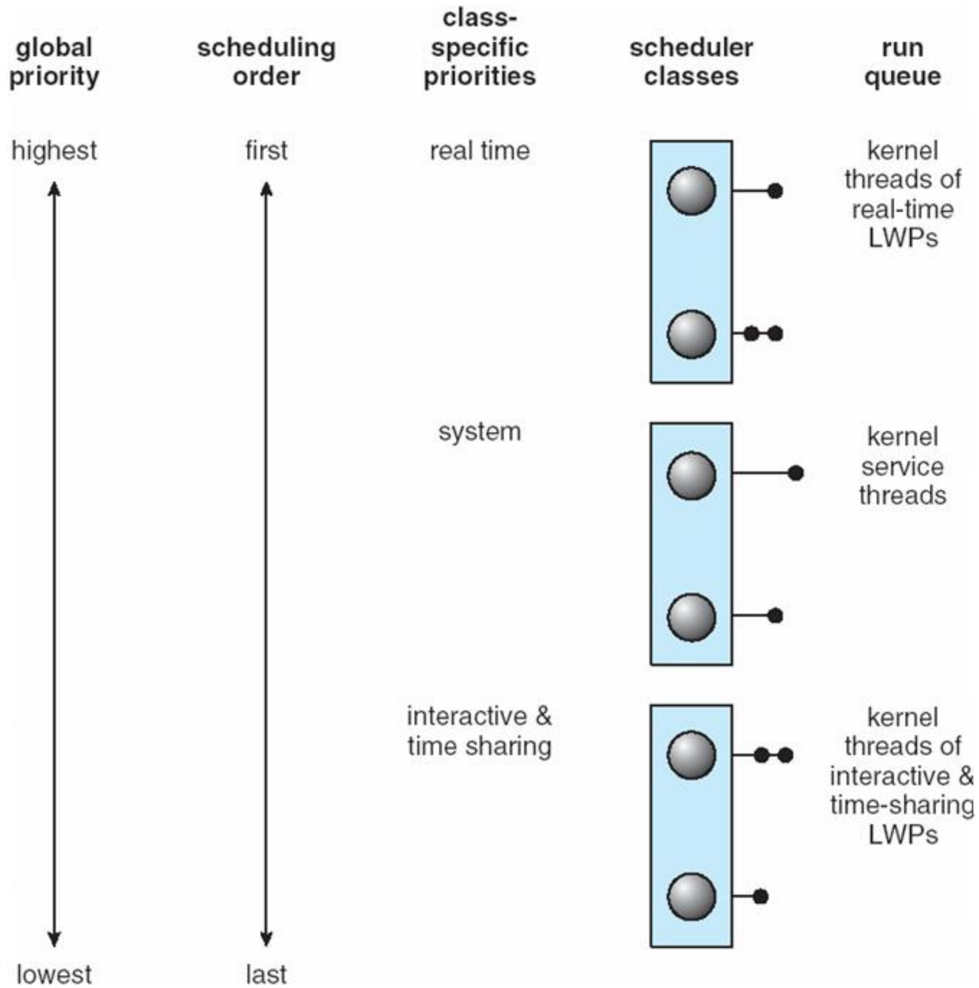
Solaris scheduling

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

Solaris Dispatch Table



Solaris 2 Scheduling

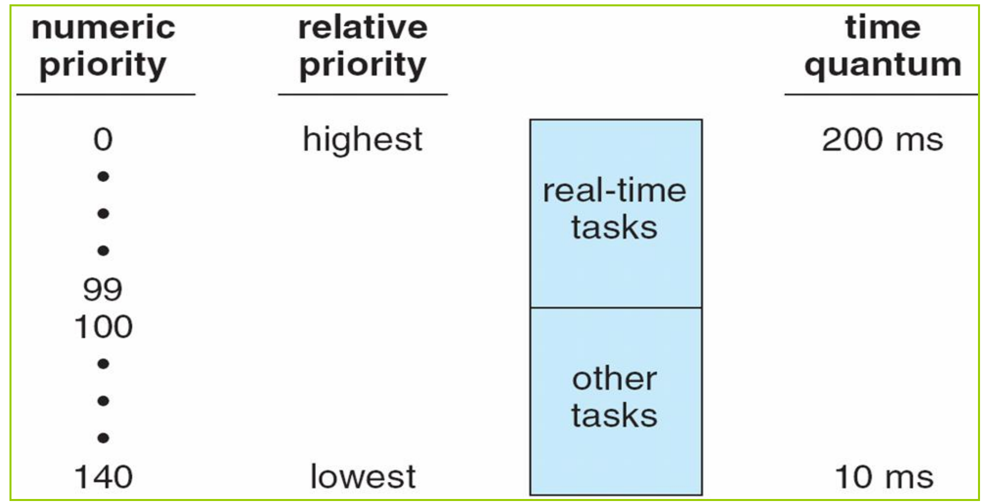


Windows XP Priorities

	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

Linux Scheduling

- Constant order *O*(1) scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140



Priorities and Time-slice length

List of Tasks Indexed According to Priorities

End of Chapter 5