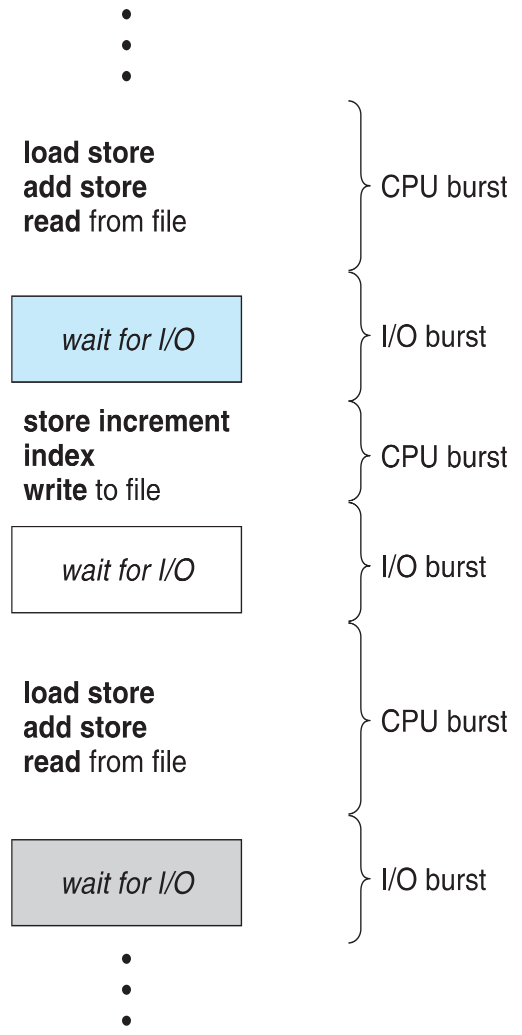
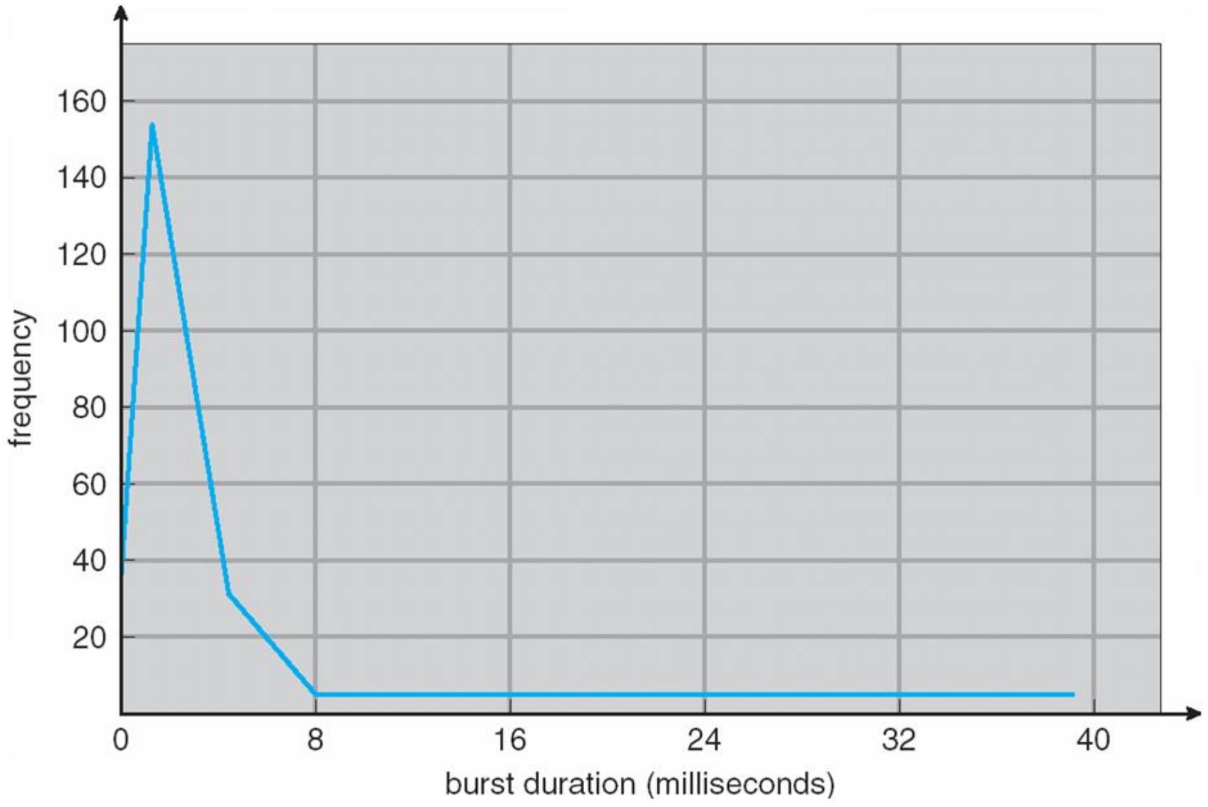
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

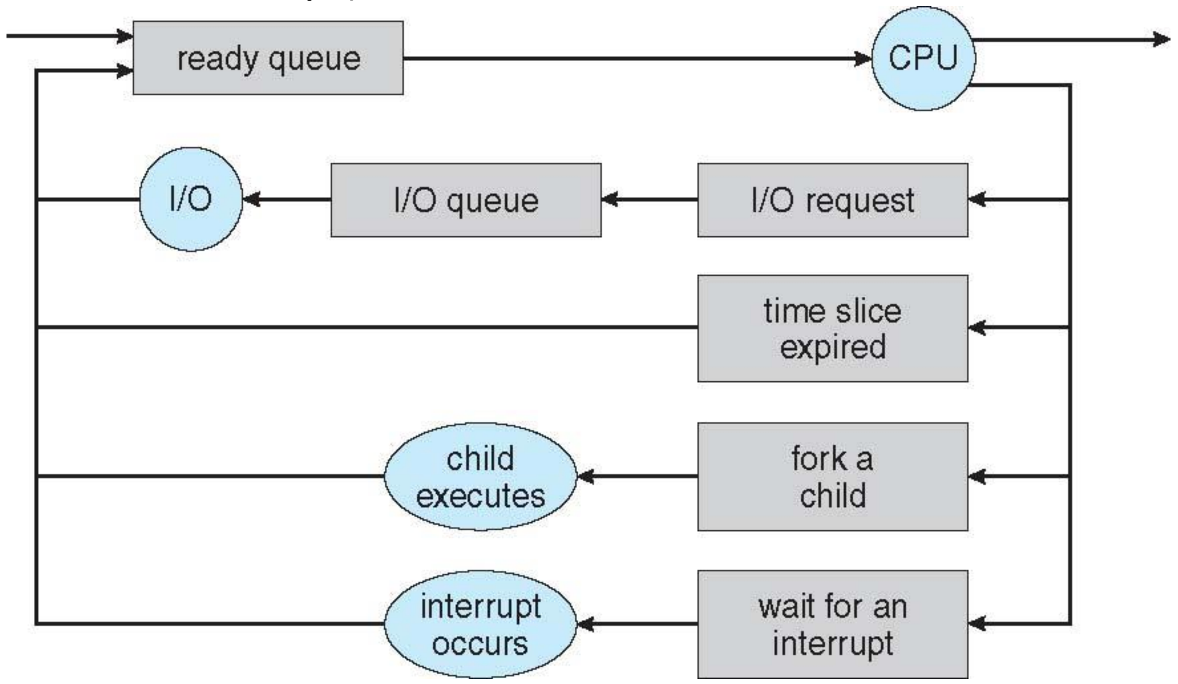


Histogram of CPU-burst Times



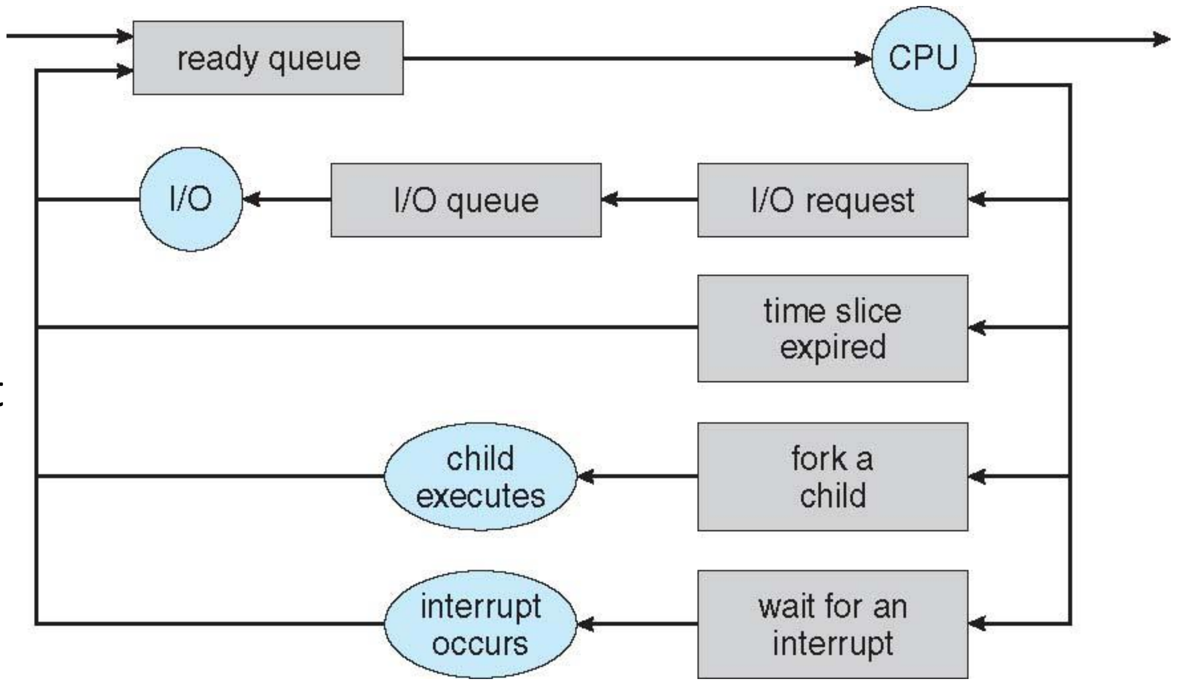
CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- Scheduling may be **non-preemptive** (e.g. FCFS, SJF and priority scheduling). Thus, CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state (i.e. blocked for a resource/device, e.g. I/O read or a semaphore)
     - Instigated by a system request (software interrupt)
  2. Terminates



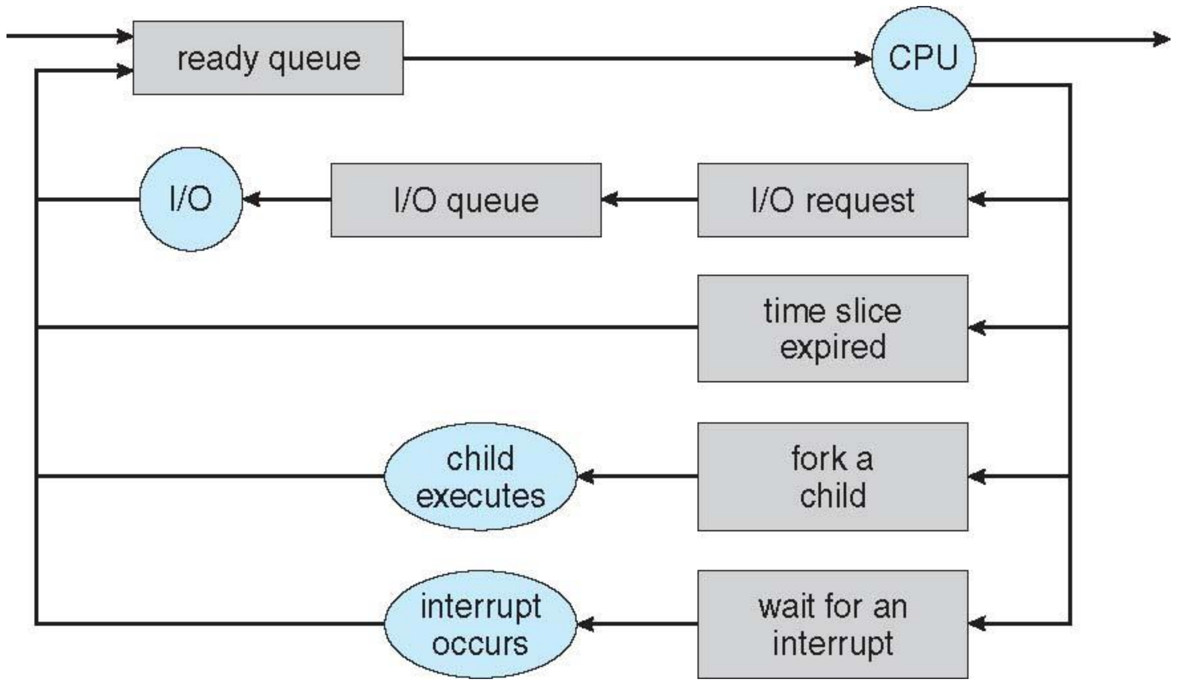
CPU Scheduler

- In **preemptive** scheduling (e.g. round robin), the scheduler may be invoked when a process:
  1. Switches from running to waiting state (i.e. blocked for a resource/device, e.g. I/O read or a semaphore)
     - Instigated by a system request (software interrupt)
  2. Switches from running to ready state
     - Instigated by a timer interrupt (I.e. the timer tick, commonly 10 ms)
  3. Switches from waiting to ready state
     - Instigated by a hardware interrupt (e.g. an I/O completion interrupt) OR
     - A system request (e.g. to signal a semaphore)
  4. Terminates



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



Scheduling Criteria (metrics targeted for optimization)

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit

- **Turnaround time** – amount of time to execute a particular process (since the time it arrived/ready)

  Completion time – arrival time
- **Waiting time** – amount of time a process has been waiting in the ready queue

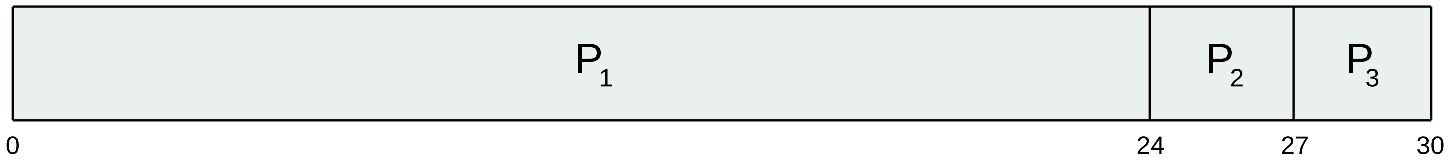
  Turnaround time – burst time

- **Response time** – amount of time it takes from when a request was submitted until the first response is produced

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:



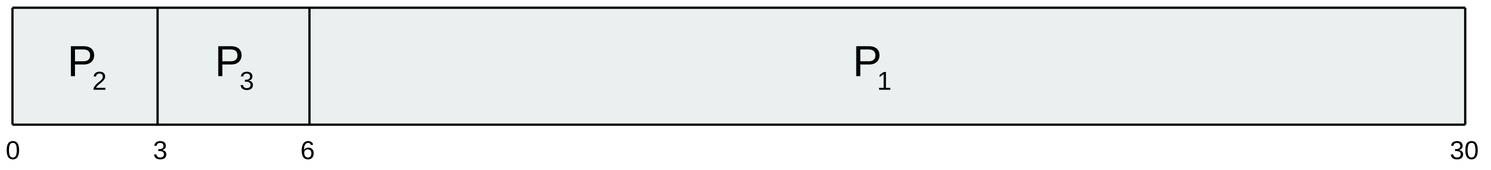
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

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
    - Consider one CPU-bound (long burst) and many I/O-bound processes (short bursts)

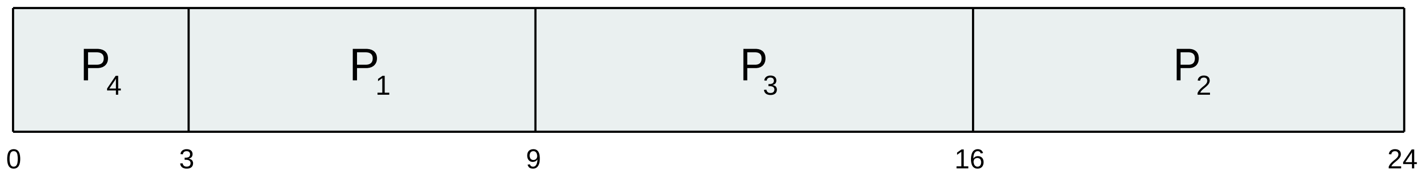
Shortest-Job-First (SJF) Scheduling

- 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 knowing the burst length of the next CPU request

Example of SJF

<u>Process</u>	<u>Burst Time</u>
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



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

Determining Length of Next CPU Burst

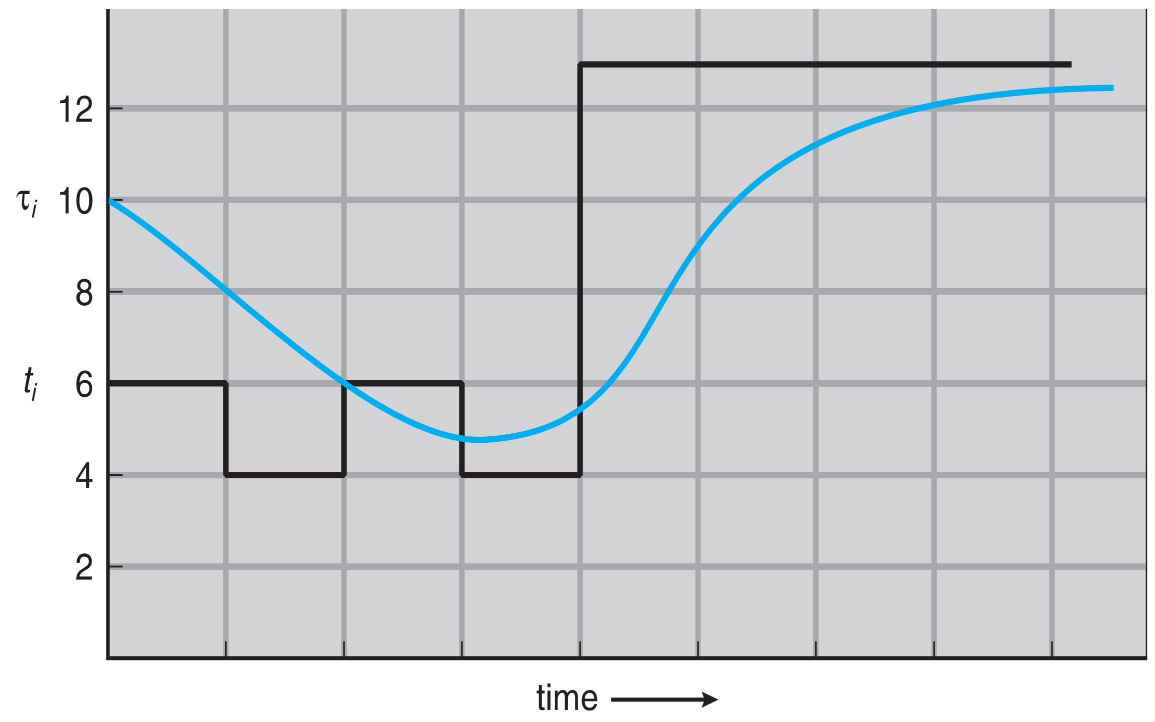
- Can only estimate the length – should be similar to the previous one (of same process of course)
  - Then pick process with shortest predicted next CPU burst

- Can be implemented using the length of previous CPU bursts and exponential averaging

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

- Commonly, α (the weight) is set to ½
- Preemptive version called **shortest-remaining-time-first**

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

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

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

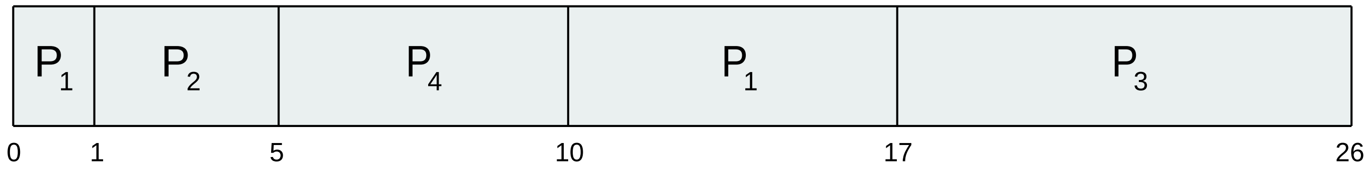
- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec
- OR (turnaround time – burst time) = [(17-0-8) + (5-1-4) + (26-2-9) + (10-3-5)]/4=6.5

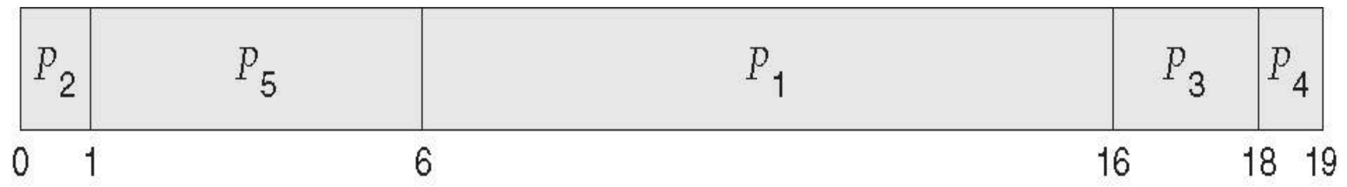
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

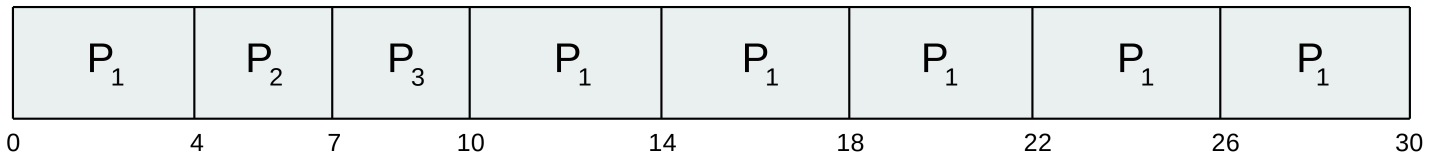
Round Robin scheduling(RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 1-100 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.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

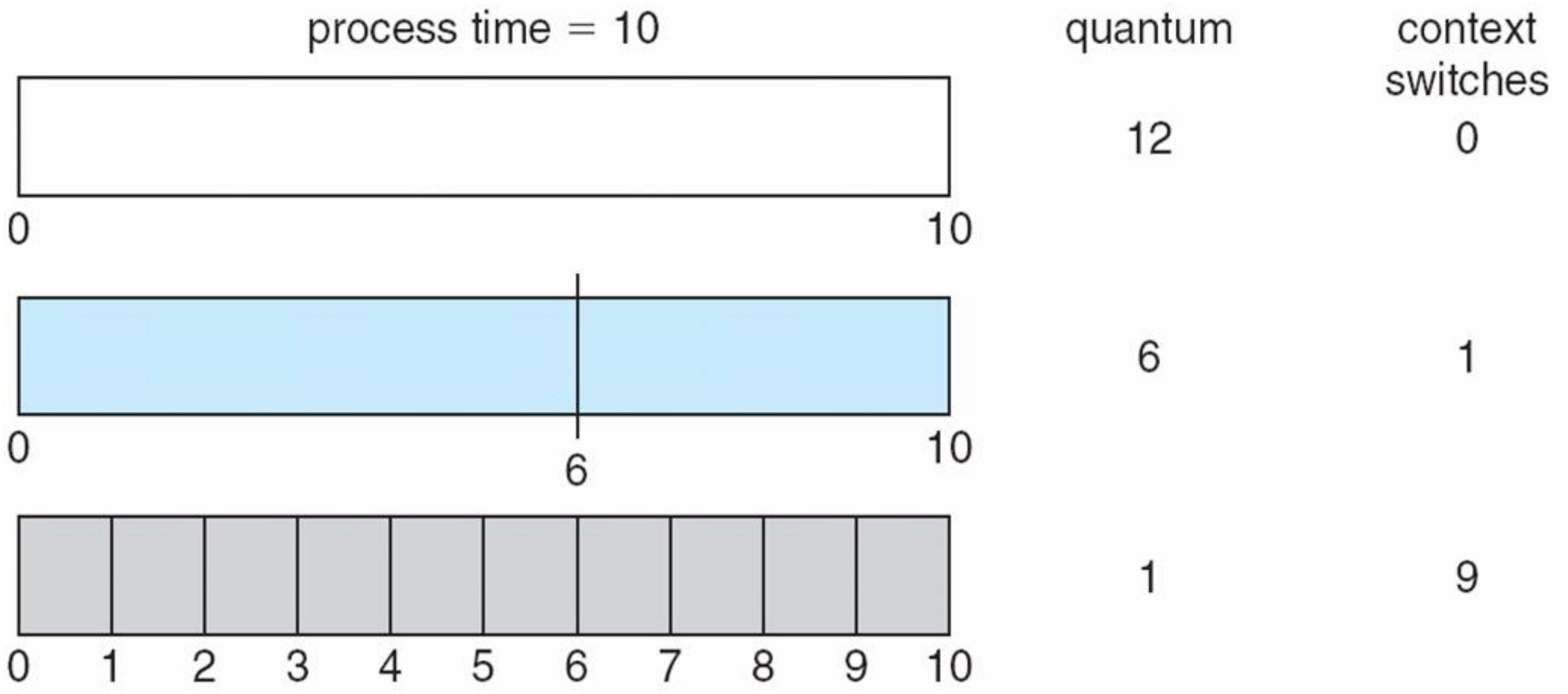
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

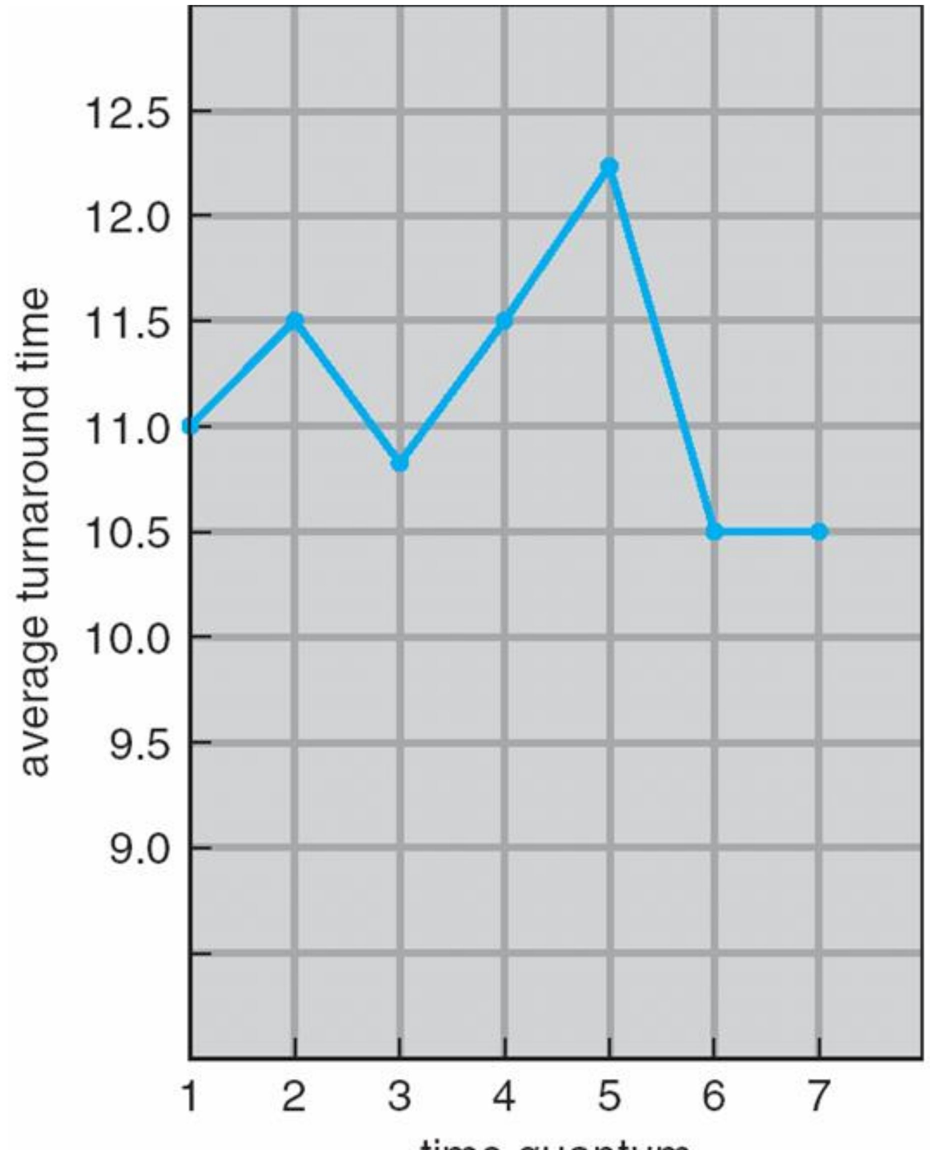


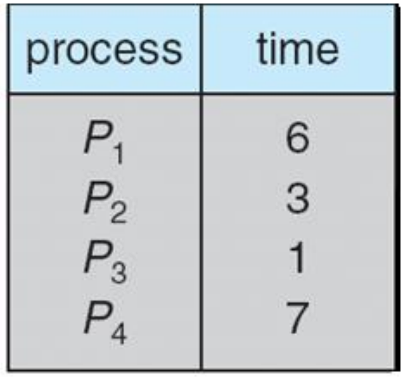
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 1ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum





process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

Multilevel Queue

- Ready queue is partitioned into separate queues (2 or more), e.g.:
  - **foreground** (interactive)
  - **background** (non-interactive or batch)
- Processes are permanently assigned to a given queue
- Each queue may have its own scheduling algorithm, e.g.:
  - 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

Thread Scheduling

- Distinction between user-level and kernel-level threads
- **When threads are supported, threads are scheduled, not processes.**
- 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
  - Typically done via priority set by programmer
- 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
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

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;

   /* get the default attributes */

   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)

      fprintf(stderr, "Unable to get scheduling
scope\n");

   else {

      if (scope == PTHREAD_SCOPE_PROCESS)

         printf("PTHREAD_SCOPE_PROCESS");

      else if (scope == PTHREAD_SCOPE_SYSTEM)

         printf("PTHREAD_SCOPE_SYSTEM");

      else
         fprintf(stderr, "Illegal scope value.\n");

   }
```

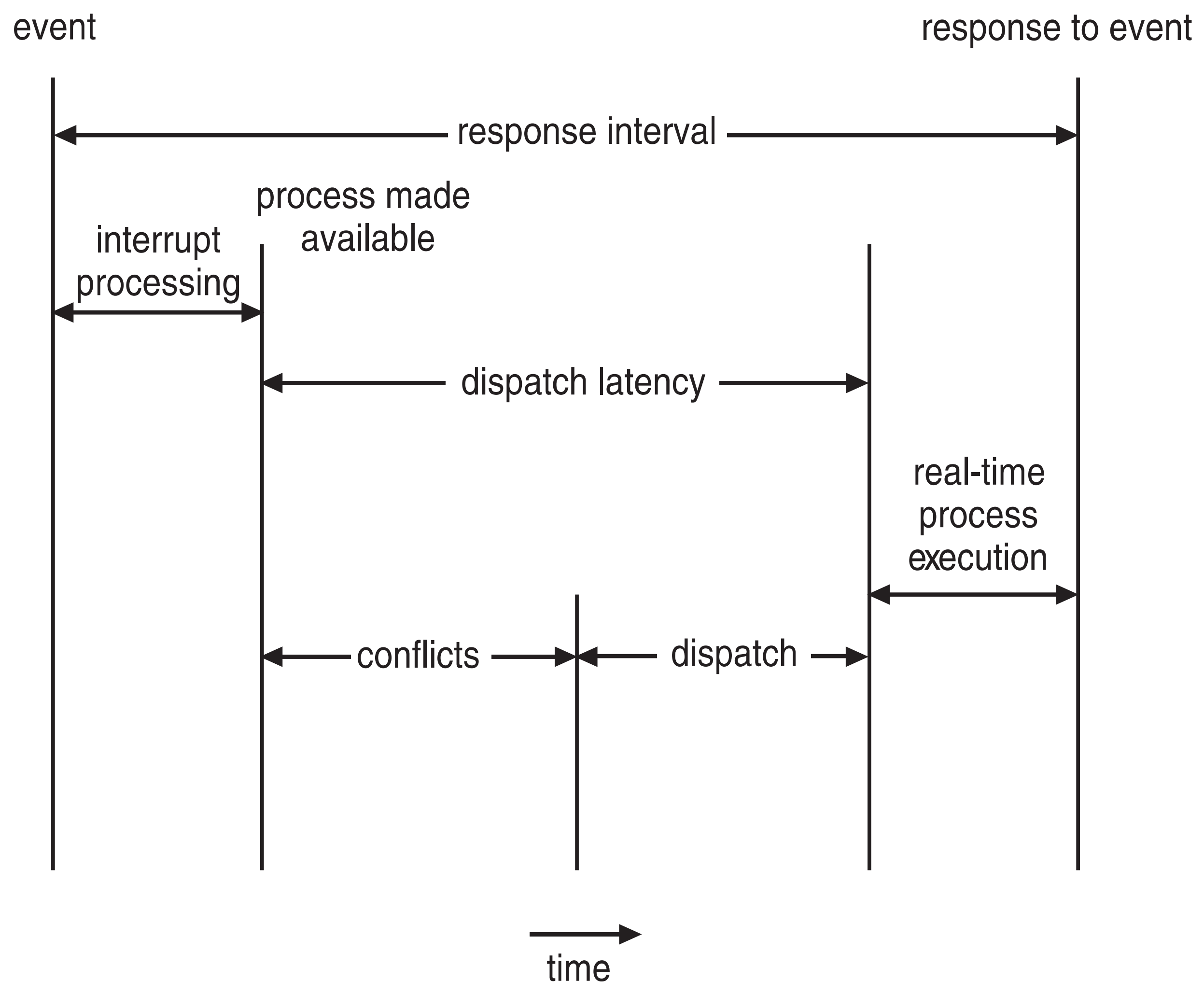
```
   /* set the scheduling algorithm to PCS or
SCS */

   pthread_attr_setscope(&attr,
PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)


pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)

      pthread_join(tid[i], NULL);

}

/* Each thread will begin control in this
function */

void *runner(void *param)
{

   /* do some work ... */

   pthread_exit(0);

}
```

Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – guarantees a real-time process will be given preference over other non real-time processes. No guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. **Interrupt latenc**y – time from arrival of interrupt to start of interrupt service routine (ISR):
     - Hardware **stops current instruction** – may be delayed if interrupts were disabled.
     - Hardware and/or software perform **interrupt context switching** and then start executing the interrupt handler for the particular event or external pin.
  2. **Dispatch latency** – time for scheduler to take current process off CPU and switch to another (**process context switching**)

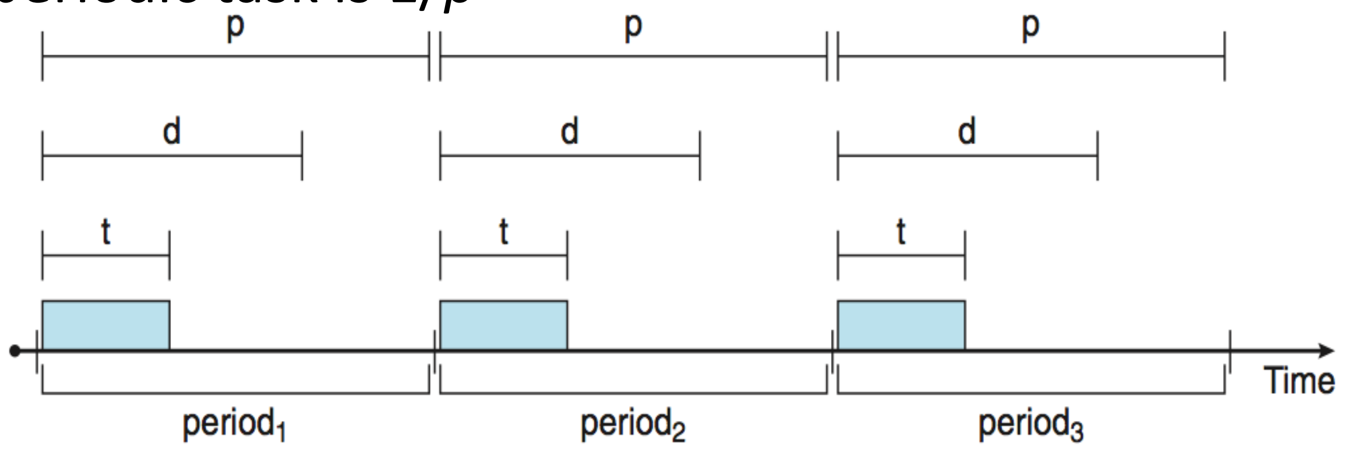
Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
    1. Preemption of any process running in kernel mode
    2. Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

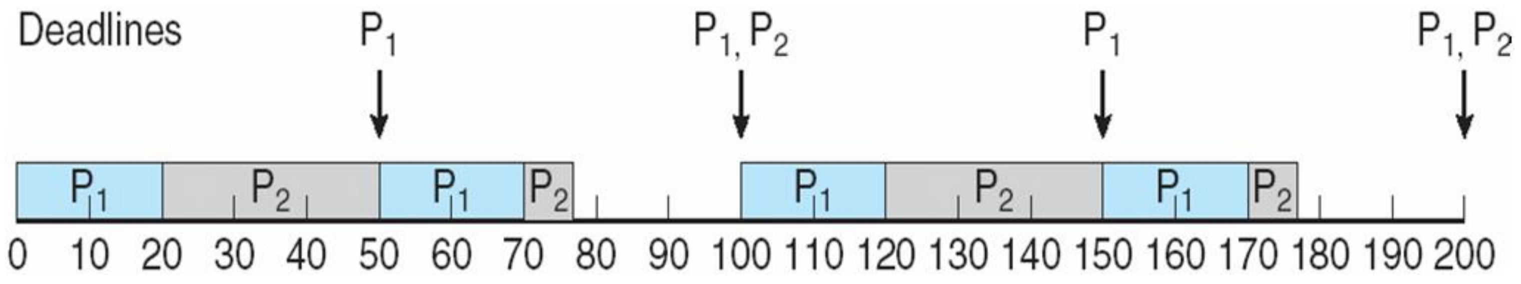
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But this only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals (i.e. they have periodic deadlines)
  - Has processing time *t*, deadline *d,* period *p*
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is 1/*p*



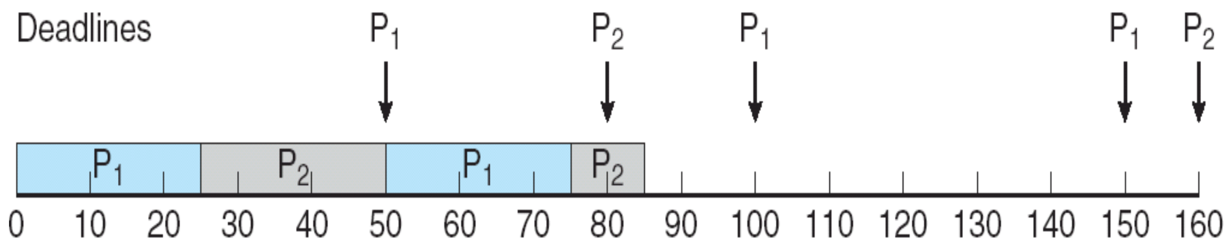
Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its required period (e.g. a task may require processing every 100 time units)
  - Shorter periods = higher priority;
  - Longer periods = lower priority
- Ex: P_1 is assigned a higher priority than P_2.

Rate Monotonic scheduling

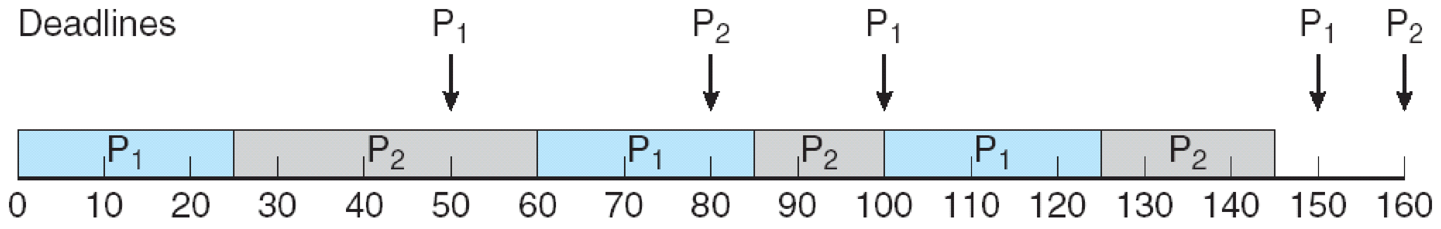


Missed Deadlines with Rate Monotonic Scheduling



Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority;
  - The later the deadline, the lower the priority



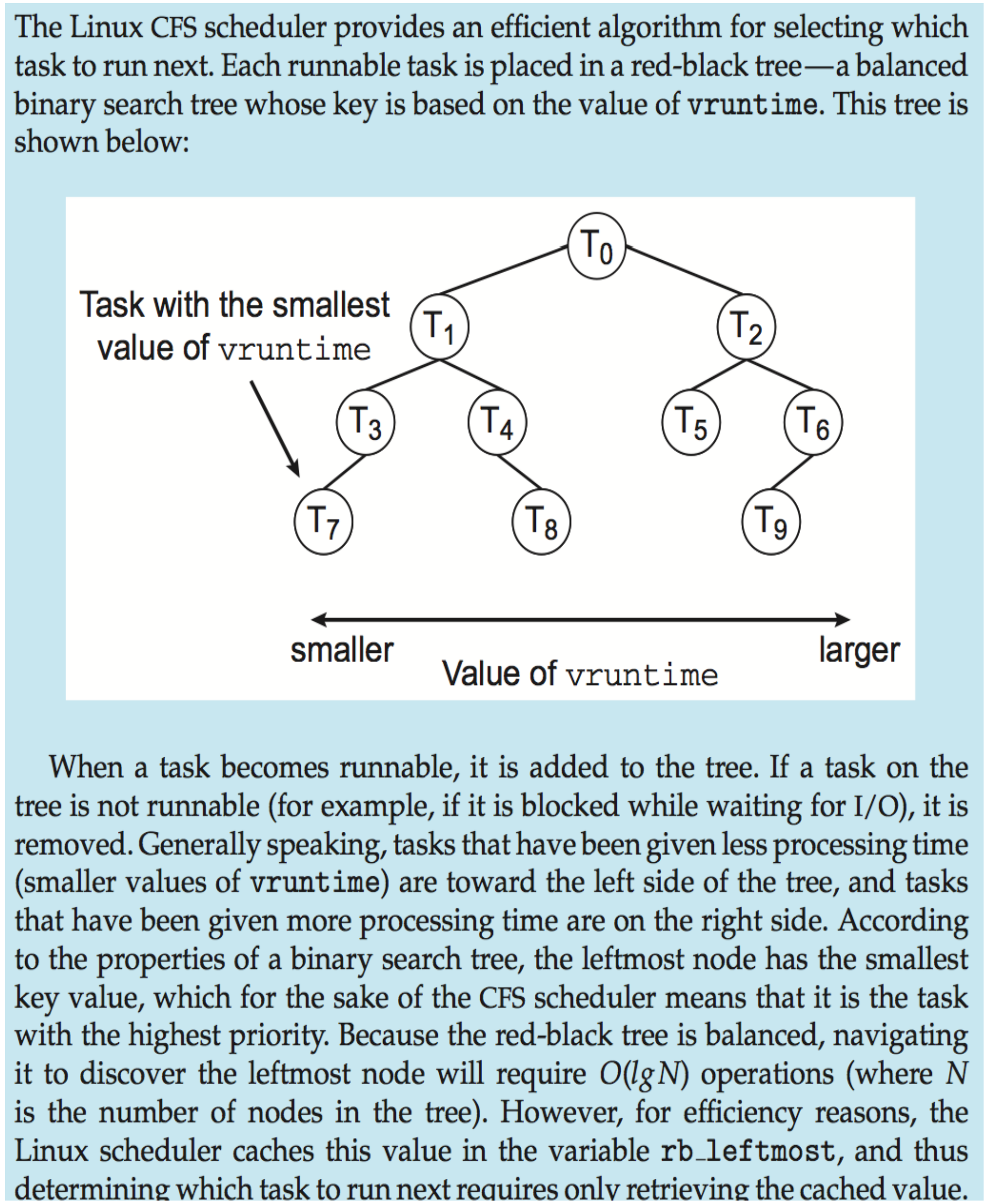
Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares where $N < T$
- This ensures each application will receive ***N*** */ T* of the total processor time

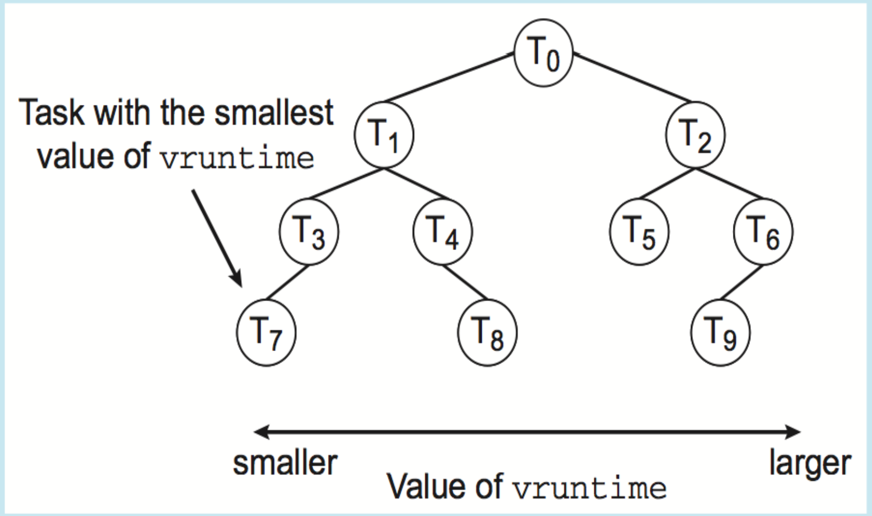
Linux Scheduling in Version 2.6.23 +

- **Scheduling classes**
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - Rather than quantum based on fixed time allotments, based on proportion of CPU time
  - 2 scheduling classes included, others can be added
    1. Default – uses ***Completely Fair Scheduler*** (CFS)
    2. real-time
- Default (non real-time) scheduling:
  - **nice value** from -20 to +19
  - Lower value is higher priority
  - CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
    - Associated with decay factor based on priority of task – lower priority is higher decay factor
    - Normal default priority yields virtual run time = actual run time
    - To decide next task to run, scheduler picks task with lowest virtual run time.
      - Lower vruntime task preempts a higher vruntime task.

CFS Performance



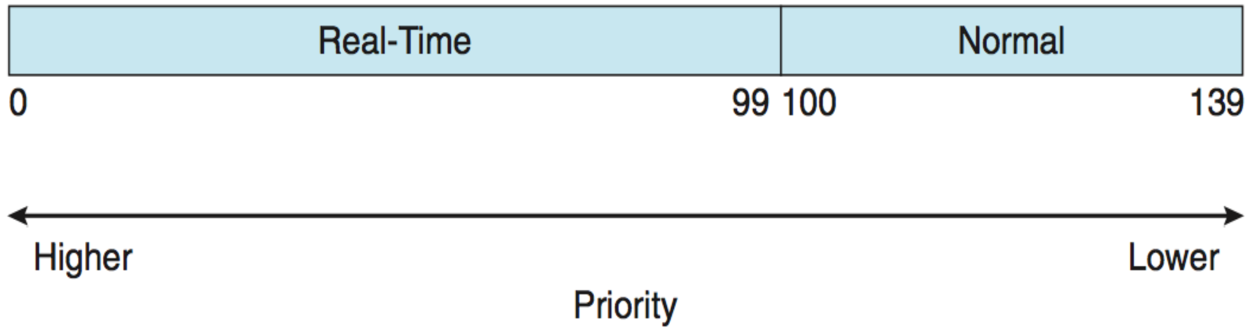
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of **vruntime**. This tree is shown below:



When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of **vruntime**) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable **rb_leftmost**, and thus determining which task to run next requires only retrieving the cached value.

Linux Scheduling – POSIX.1b

- Real-time scheduling according to POSIX.1b
  - Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
  - Nice value of -20 maps to global priority 100
  - Nice value of +19 maps to priority 139
- POSIX.1: The nice value is a per-process attribute; that is, the threads in a process should share a nice value.
- Linux: The nice value is a per-thread attribute: different threads in the same process may have different nice values.



Linux/POSIX1.b non real-time classes

- **Non real-time scheduling policies**:
  - **SCHED_OTHER**
    - The **default** scheduling policy for non real-time threads
    - Has static priority of 0.
    - Dynamic priority (nice value) ranges from -20 to 19 (19 has least priority). With each scheduler tick, a thread's dynamic priority decreases → pseudo **round-robin** sharing (this is a logical view, actual implementation uses the CFS/vruntime method)
  - **SCHED_BATCH**
    - Has less static priority than SCHED_OTHER.
    - Used for batch processing, i.e. CPU intensive threads
    - Has a similar dynamic priority scheme as SCHED_OTHER
  - **SCHED_IDLE**
    - For extremely low priority (background) jobs, even lower than SCHED_BATCH with nice values of 19.

Linux/POSIX1.b Real-Time Scheduling

- The POSIX.1b standard - API provides functions for managing real-time threads
- **Real-time scheduling classes**
  - Threads have a priority from 1 to 99
  - Higher priority threads always preempt lower priority threads
  - Always preempts non real-time threads.
  - For real-time threads of equal priority, two policies are offered.

  1. **SCHED_FIFO** – uses a FCFS strategy with a FIFO queue, no time-slicing for threads
  2. **SCHED_RR** - time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:

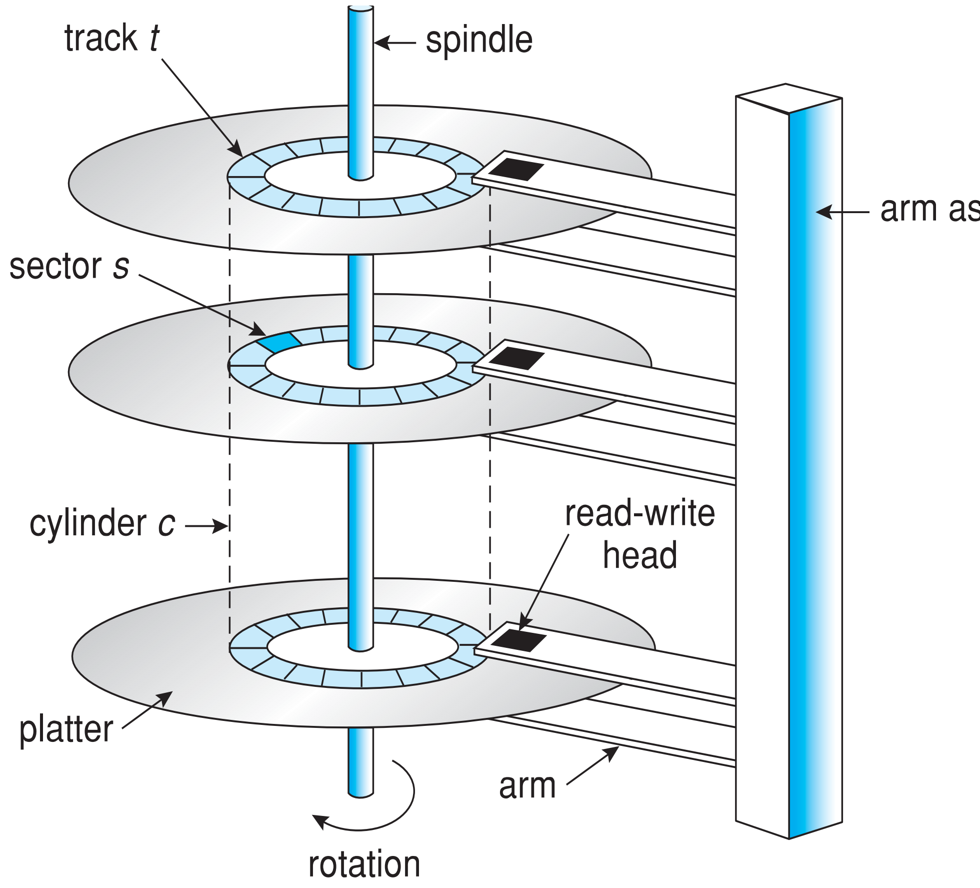
```
1.pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)
2.pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
```

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr,
&policy) != 0)
      fprintf(stderr, "Unable to get
policy.\n");
   else {
      if (policy == SCHED_OTHER)
printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR)
printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
printf("SCHED_FIFO\n");
   }
   /* set the scheduling policy */
   if (pthread_attr_setschedpolicy(&attr,
SCHED_FIFO) != 0)
      fprintf(stderr, "Unable to set policy.\n");
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this
function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

10.1 Overview of Mass Storage Structure

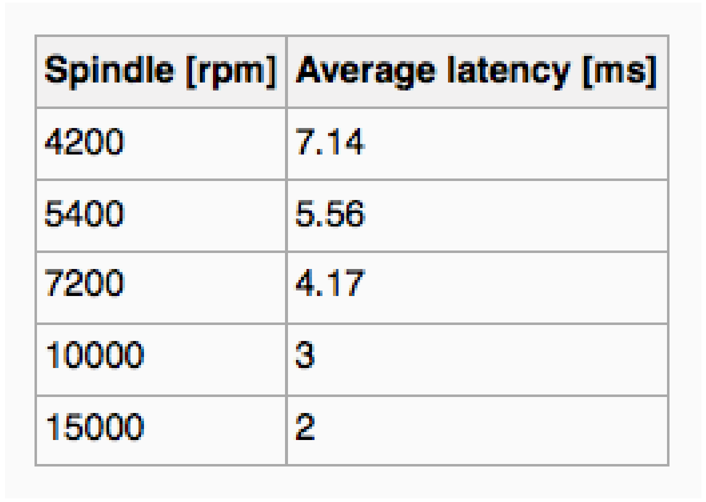
- **Magnetic disks** provide bulk of secondary storage for modern computers
  - Drives rotate at 60 to 250 cycles per second
  - **Positioning time** = **seek time** + **rotational latency**
    - **Seek time: T**ime to move disk arm to desired cylinder
    - **Rotational latency**: Time for desired sector to rotate to the disk head
  - **Transfer rate** is rate at which data flow between drive and computer
  - **Head crash** results from disk head making contact with the disk surface -- That's bad
- Disks can be fixed or removable



Moving-head Disk Mechanism

Hard Disks

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Today, they may reach 16TB of storage per drive
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
    - Effective Transfer Rate – real – 1Gb/sec
  - Average seek time measured or calculated based on 1/3 of tracks
    - Ranges from 3ms to 12ms – 9ms common for desktop drives
  - Latency based on spindle speed
    - 1 / (RPM / 60) = 60 / RPM
  - Average latency = ½ latency



Spindle [rpm]	Average latency [ms]
4200	7.14
5400	5.56
7200	4.17
10000	3
15000	2

(From Wikipedia)

Hard Disk Performance

- **Access Latency** = **Average access time** = average seek time + average latency
  - For fastest disk 3ms + 2ms = 5ms
  - For a typical desktop 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- **Example:** to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/sec transfer rate with a 0.1ms controller overhead =
  - 5ms + 4.17ms + 0.1ms + transfer time =
  - Transfer time = 4KB / 1Gb/s * 8Gb / GB * 1GB / 1024^2KB = 32 / (1024^2) = 0.031 ms
  - Average I/O time for 4KB block = 9.27ms + .031ms = 9.301ms

Solid-State Disks

- Nonvolatile memory used like a hard drive
  - Many technology variations, e.g. NOR flash, NAND flash, etc.
  - Becoming more commonly used than hard disks for laptops and mobile devices
- Can be more reliable than HDDs
- More expensive per MB
- Maybe have shorter life span
  - Due to limited number of writes on flash disks.
- Less capacity
- No moving parts, so no seek time or rotational latency → much faster
  - Busses can be too slow -> connect directly to PCI for example

Magnetic Tape

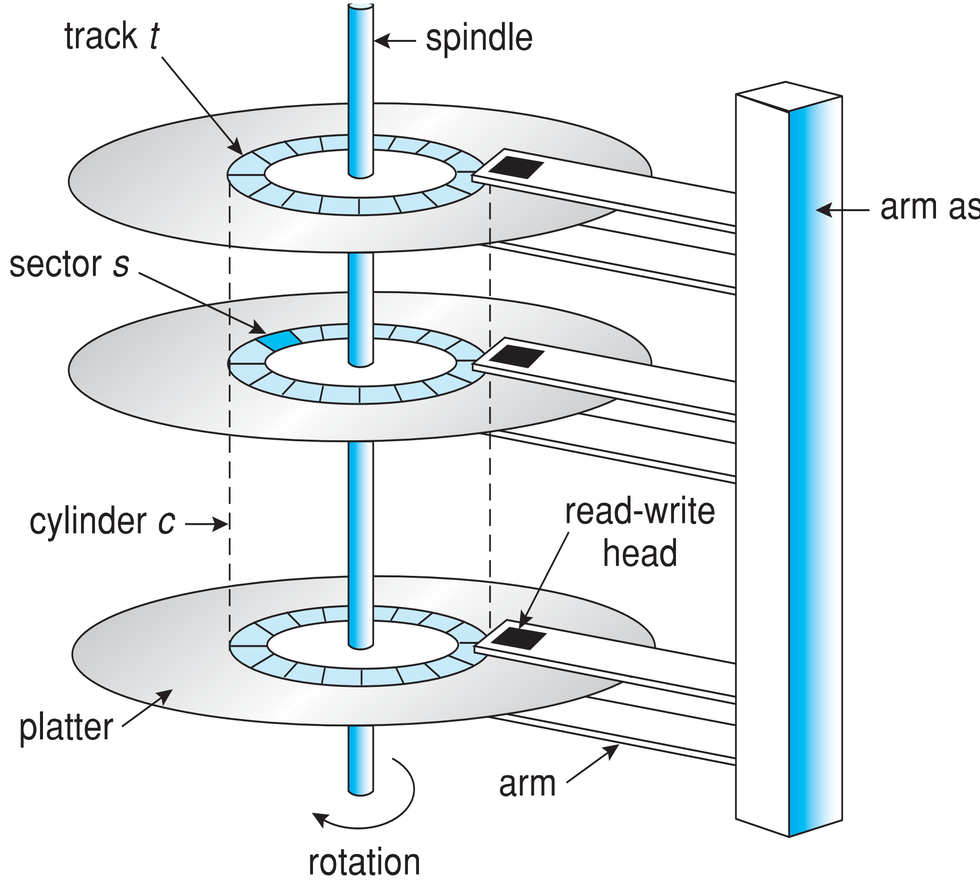
- An early secondary-storage medium
  - Evolved from open spools to cartridges
- Kept in spool and wound or rewound past read-write head

- **Relatively permanent** and holds large quantities of data
  - 200GB to 1.5TB typical storage
- Access time slow
  - Random access ~1000 times slower than disk
- Once data under head, transfer rates comparable to disk
  - 140MB/sec and greater

- → Mainly used for backup, storage of infrequently-used data or transfer medium between systems

10.2 Disk Structure

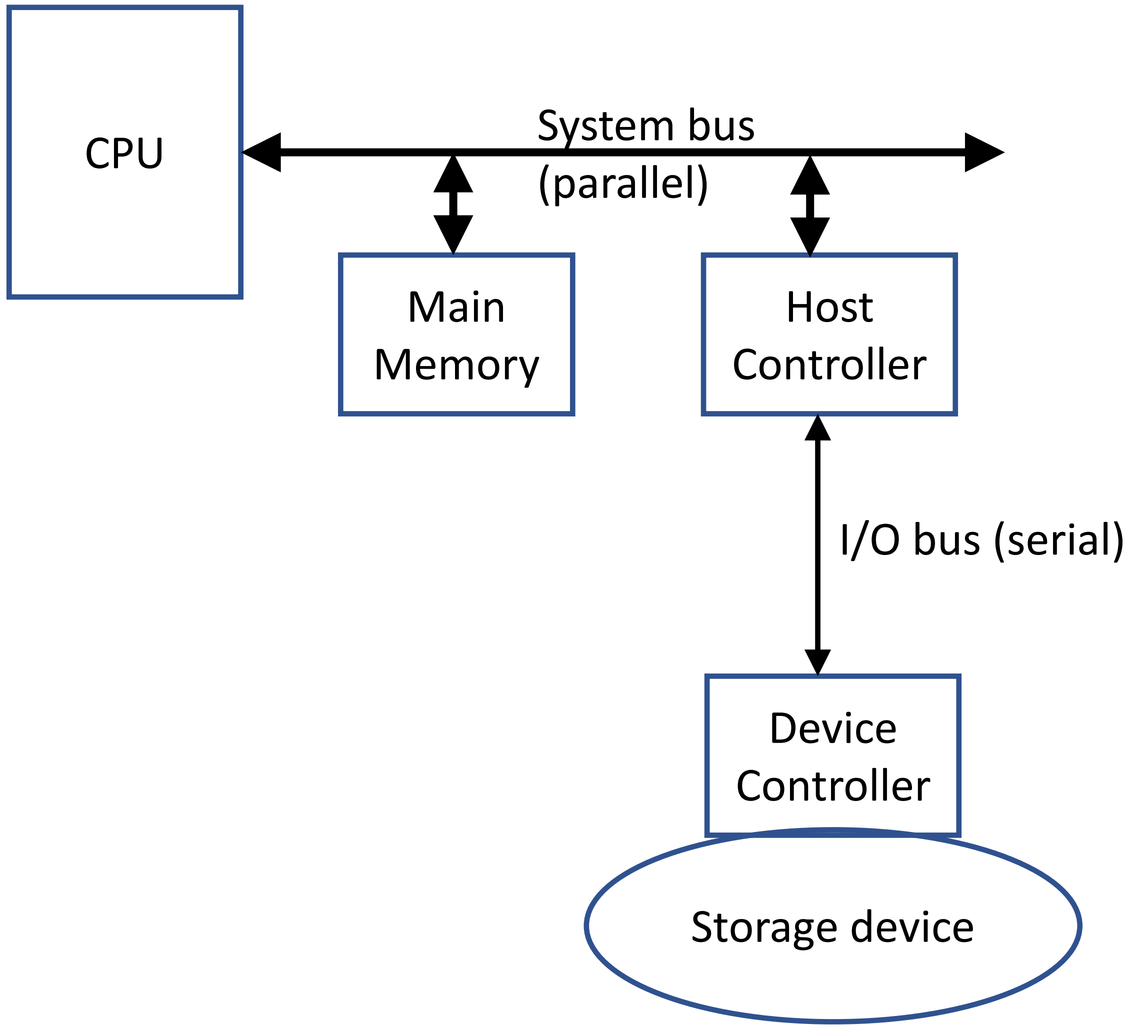
- Disk drives are addressed as large 1-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer
  - Low-level formatting creates **logical blocks** on physical media
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address translation involves
    - Skipping the bad sectors
    - Non-constant # of sectors per track (outer tracks may have more sectors than inner tracks)



10.3 Disk Attachment

Host-attached storage:

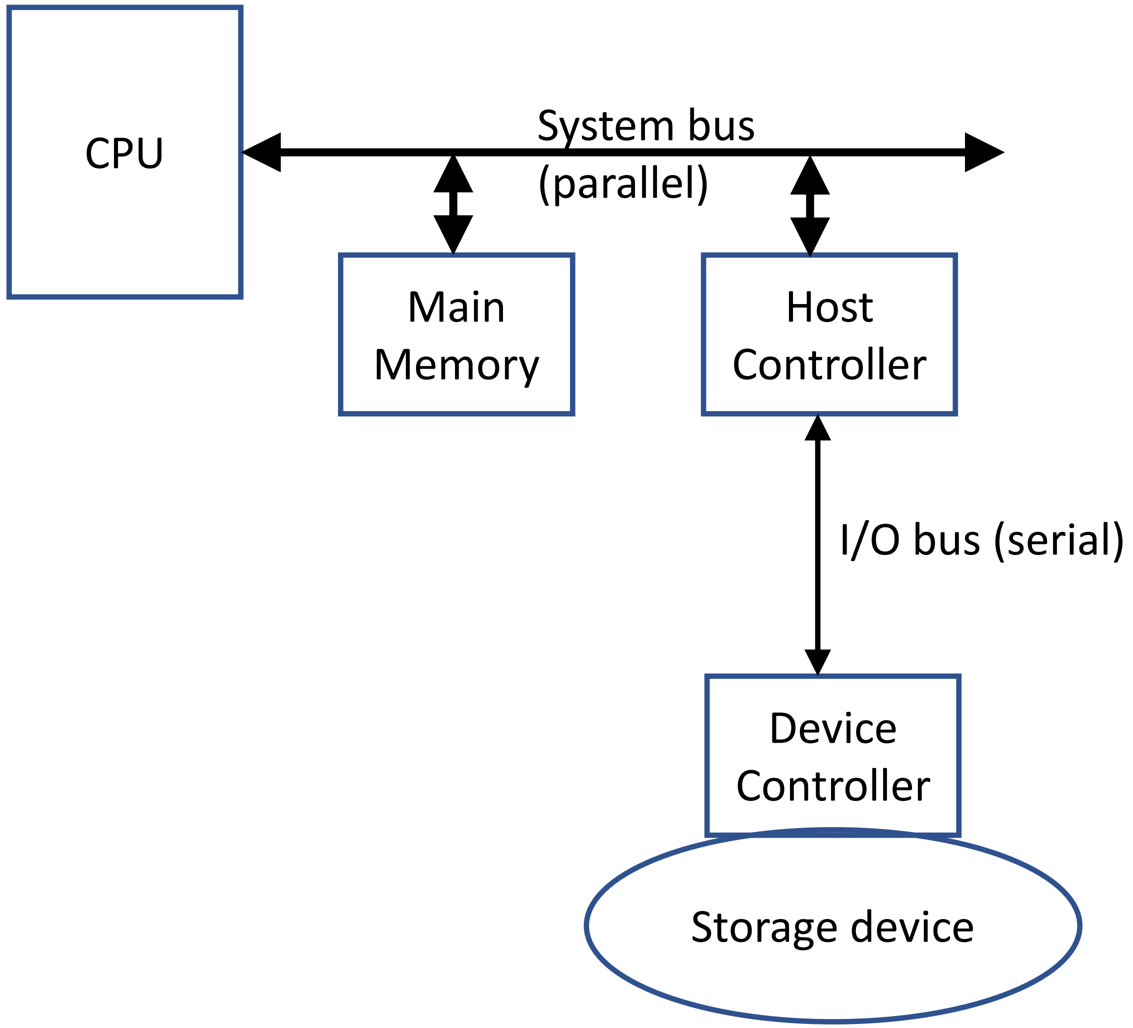
- Accessed through I/O ports on I/O busses.
- The storage I/O bus is attached to the system bus via a hardware controller, the **host controller**. The disk is attached to a **device controller**, which is connected to a port on the I/O bus.
- The I/O commands that initiate data transfers to a host-attached storage device are **reads** and **writes** of **logical blocks** directed to **storage units** (e.g. hard disks) addressed on **ports/channels** on the bus.
- SCSI (Small Computer System Interface) is an I/O bus:
    - Up to 16 device controllers on one cable
    - Each target (controller) can have up to 8 **logical units** (i.e. disks) attached.
    - **SCSI initiator** requests operation and **SCSI targets** perform tasks
    - Initiator sends one-byte operation code followed by five or more bytes containing command-specific parameters.
    - Target performs command and returns a status byte.



Disk Attachment – cont.

Host-attached storage – cont.

- **ATA** (Advanced Technology Attachment), SATA (Serial ATA) and ATAPI (ATA Packet Interface) are other examples of local attachment buses.
  - Host controller does not do much, just an electrical interface.
  - Host's device driver has to compose and send all the commands
- **IDE** (Integrated Drive Electronics) is capable of connecting to drives that support any of the three protocols but has the software drivers running within the IDE controller chip.
  - Host driver does not need to send low level ATA commands.
- **Fiber Channel, USB, Firewire, etc.**


CPU
System bus
(parallel)
Main Memory
Host Controller
I/O bus (serial)
Device Controller
Storage device