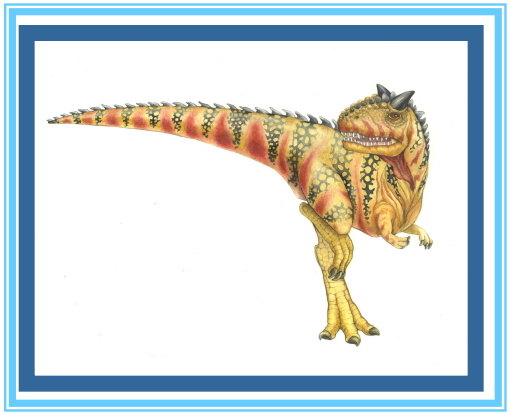
Section 5: CPU Scheduling



Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples

Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems

CPU Scheduling

- CPU scheduling refers to the execution order of processes or threads on the CPU

Scheduling Processes or Threads

- It does not matter
- Scheduling policies apply the same to processes or threads

Context Switch

- Always do processes context switch if OS does not implement multithreads
  - Need context switch of the whole PCB
- Perform kernel threads context switch if multithreading OS
  - Context switch between threads in the same process is faster than context switching between processes.
  - Context switch between threads from different processes still requires a process context switch

Costs for Context Switch

- The context (PCB) of a process is much heavier than for thread
  - More info to copy during context switch
- Memory management is complex because processes do not share their memory
- Thread do not worry about accounting
- Threads share files, no change when context switch between threads
- Context switch between threads from different processes pay the same overhead as processes context switch

Processes/threads execution

- When a process/thread executes, it either
  - Execute some instructions on CPU (CPU execution)
  - Wait for some I/O request (read or write data to a file or to get input from a user).
- The period of computation between I/O requests is called the **CPU burst**.

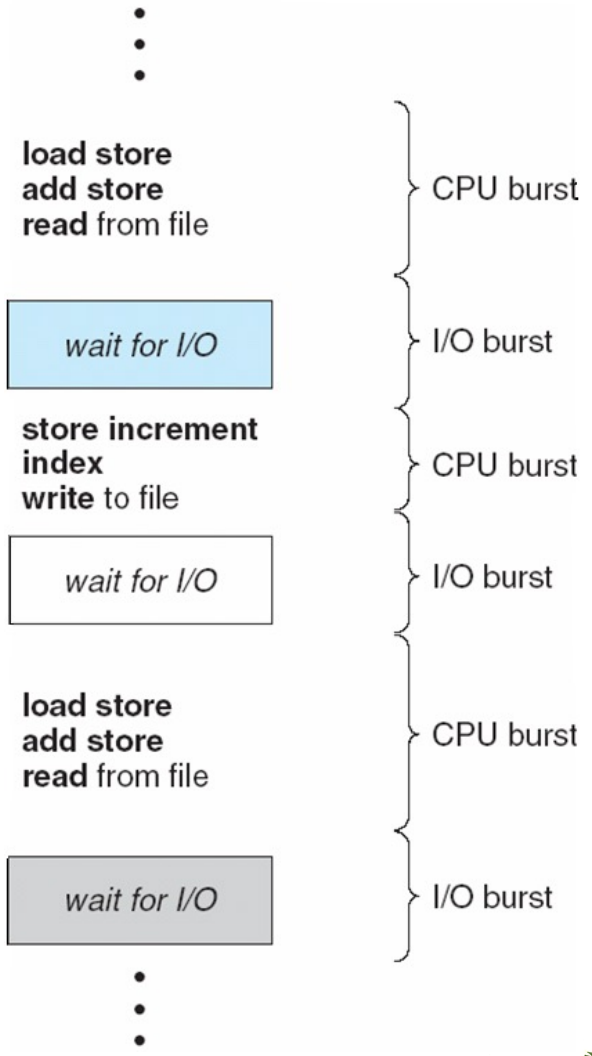
CPU	I/O	CPU	I/O

Item	Time		Time in human terms
Processor cycle	0.5ns	2Ghz	1 second
Memory access	15ns		30 seconds
Context switch	5,000ns	5μs	167 minutes
Disk access	7,000,000ns	7ms	162 days
One keystroke	100,000,000ns	100ms	6.3 years

Table 1: Time scales (A fast typist can type a keystroke every 100 milliseconds)

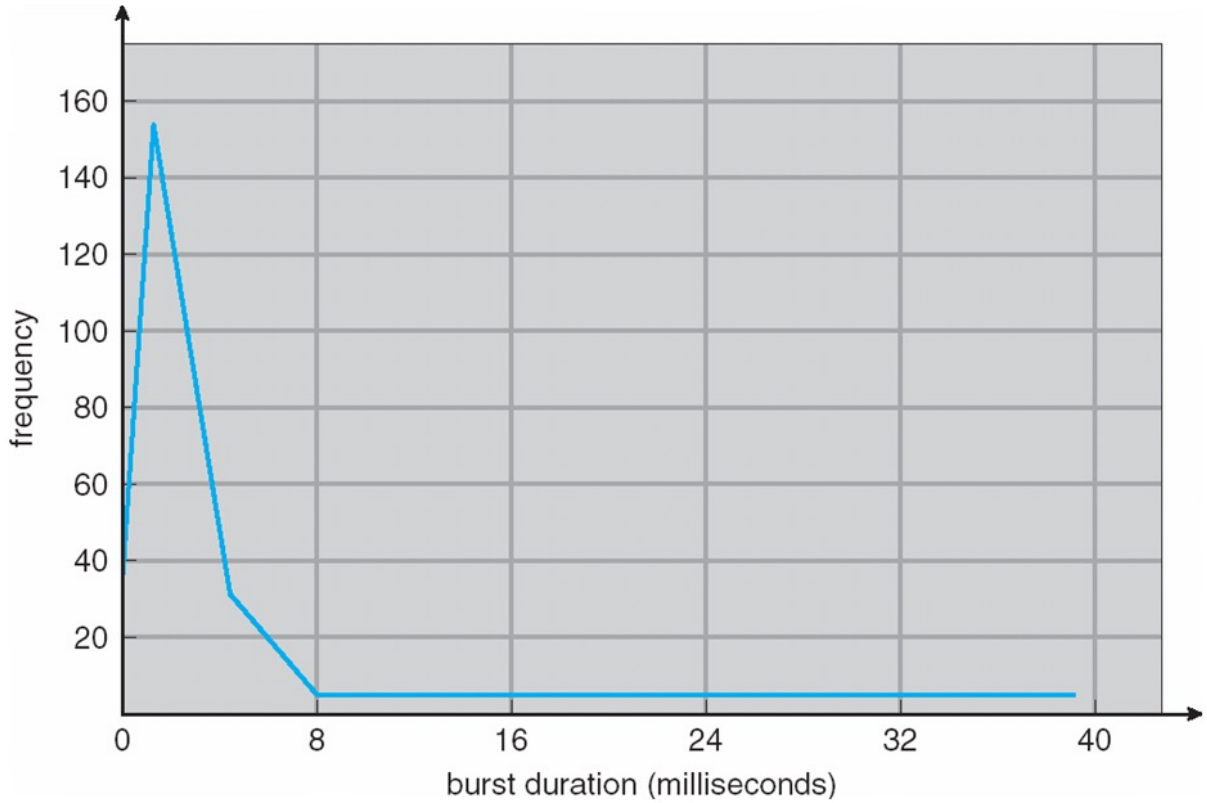
Processes execution

- CPU scheduling depends on process execution which can be divided into *cycles* of CPU execution and I/O wait
  - CPU–I/O Burst Cycles
- Processes alternate between these two states
  - CPU burst follows by a I/O burst follows by CPU burst follows by I/O burst, and so on



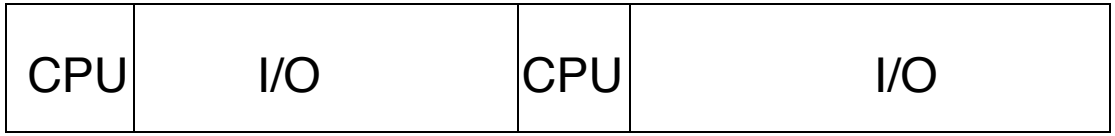
Histogram of CPU-burst Times

- CPU burst distribution has been measured
- Large number of small CPU burst and a small number of large CPU burst

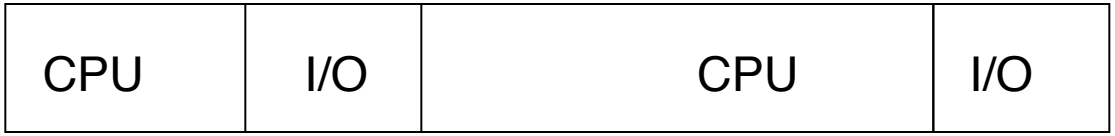


Processes characterization

- Interactive processes spend more time performing and waiting for I/O and generally experience short CPU bursts:



- Compute-intensive processes, conversely, spend more time running instructions and less time on I/O. They exhibit long CPU bursts:



Scheduling algorithms

- Scheduling algorithms are classified as either preemptive or nonpreemptive
- In nonpreemptive scheduling algorithms:
  - once the CPU has been allocated to a process, the process keeps the CPU until it releases it either:
    - by terminating or
    - by switching to the waiting state
- In preemptive scheduling algorithms, the scheduler
  - may place time quantums for the execution of processes, then processes switch from running to ready state or
  - may forcibly removes a process from the CPU once a process with higher priority enters the ready queue

Preemptive Scheduling

- +
  - Better service to the overall set of processes (no monopolization of CPU)
- -
  - more overhead then nonpreemptive
  - might de-schedule a process that update a data structure needed by another process
    - Data structure is in inconsistent state
    - To prevent this, need synchronization mechanisms

Scheduling Criteria

- CPU utilization – How busy the CPU is?
  - Usually try to maximize this criterion, i.e. keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
  - Usually try to maximize this criterion
- Turnaround time – amount of time to execute a particular process, from submission to completion
  Waiting in memory, ready and I/O queues, running on the CPU (minimize)
- Waiting time – amount of time a process spent in the ready queue (minimize)
- 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, minimize)

Scheduling Criteria

- System, performance oriented:
  - Focus on effective and efficient utilization of the CPU
    - Max CPU utilization
    - Max throughput
- User oriented:
  - How the user or process perceived execution
    - Min response time
- User oriented and performance related:
  - Min turnaround time; Min waiting time; Min response time

Scheduling Algo: Optimization Criteria

- When selecting a scheduling algorithm, which of the following criteria are optimized:
  - CPU utilization, throughput, turnaround time, waiting time, response time
- Usually not possible to optimize all of them because they may contradict each other
  - Min response time requires frequent context switching which reduces throughput
- May seek to optimize average over all executing processes

Different Scheduling Algorithms

- Real CPU schedulings are complexe combinations of different scheduling algorithms
- We describe a small set of basic scheduling algorithms:
  - First-Come First-Serve (FCFS scheduling algorithm)
  - Shortest-Job-First (SJF scheduling algorithm)
  - Round-Robin (RR scheduling algorithm)
  - Priority scheduling algorithm
  - Multilevel queues

First-Come, First-Served Scheduling

- FCFS:
  - Ready queue is FIFO
  - The simplest of the process scheduling algo.
  - Easy to implement
  - A non-preemptive scheduling ago, once a process is scheduled it runs until the end or until it executes an I/O
- FCFS is not optimal, further we should expect large variations in the average waiting time
- Tend to favor CPU-bound over I/O-Bound, once a CPU-bound has the CPU all the I/O-bound processes must wait

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 and many I/O-bound processes

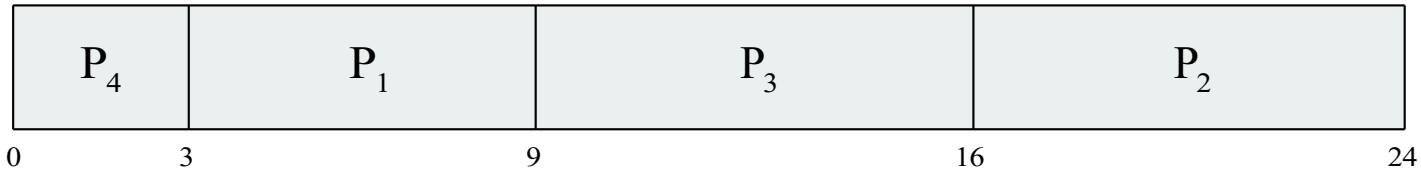
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Select from the ready queue the process with the shortest next CPU burst
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

Example of SJF

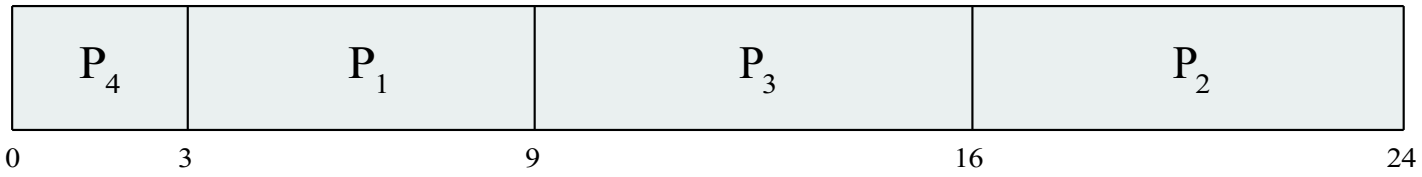
Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- With FCFS scheduling algo, average waiting would have been 10.25

SJF



- SJF is provably optimal with respect to minimizing the average waiting time
  - Moving short process before long one decreases the waiting time of the short more than increases the waiting time of the long process
  - However, the length of the next CPU burst time is unknown

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

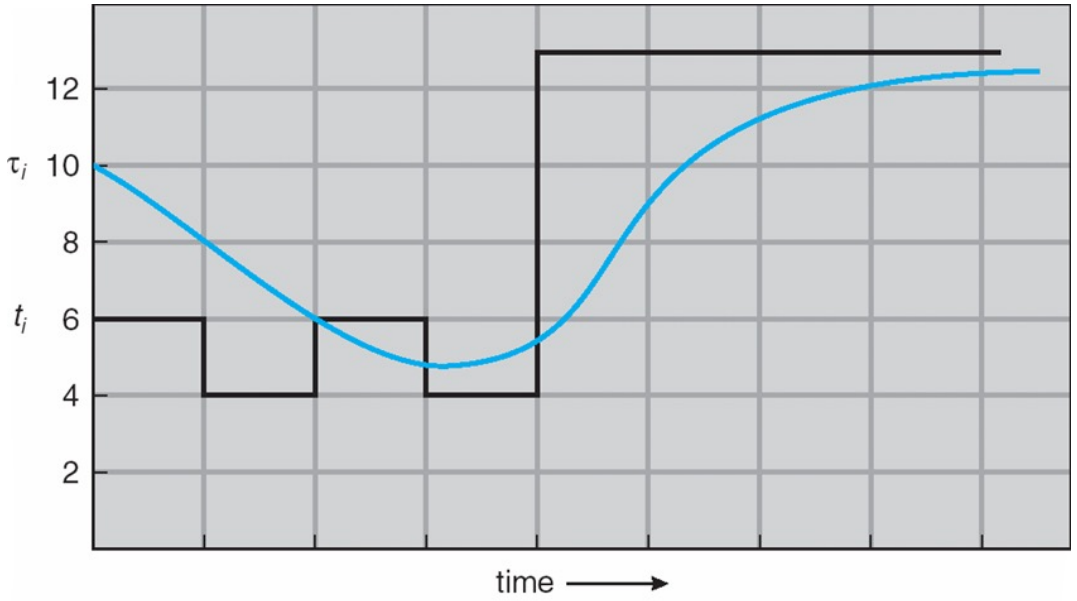
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. $\alpha, 0 \le \alpha \le 1$
  4. Define :

- If α =0, $\tau_{n+1} = \tau_n$, recent information does not count
- If α =1, $\tau_{n+1} = \alpha\ t_n$, only the actual last CPU burst counts
- Commonly, α set to ½ $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

Example of Exponential Averaging

- The initial prediction can be defined as a constant or as an overall system average, below $\tau_0 = 10$ (estimation at t_0)
- So, the figure below shows an exponential average with $\alpha = 1/2$ and $\tau_0 = 10$.

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



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

Example of Exponential Averaging

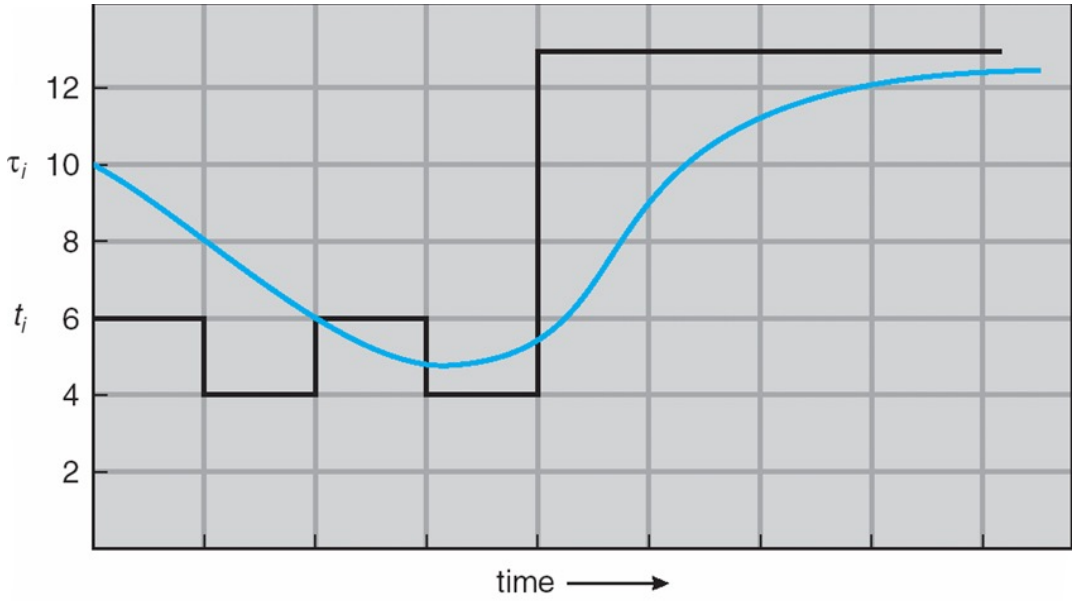
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

- $\tau_1 = .5 * t_0 + .5 * \tau_0$
- $= .5 * 6 + .5 * 10 = 8$
- $\tau_2 = .5 * t_1 + .5 * \tau_1$
- $= .5 * 4 + .5 * 8 = 6$
- $\tau_3 = .5 * t_2 + .5 * \tau_2$

  $= .5 * 6 + .5 * 6 = 6$
- $\tau_4 = .5 * t_3 + .5 * \tau_3$

  $= .5 * 4 + .5 * 6 = 5$
- $\tau_5 = .5 * t_4 + .5 * \tau_4$

  $= .5 * 13 + .5 * 5 = 9$



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

Actual

Guess

Example of Exponential Averaging

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

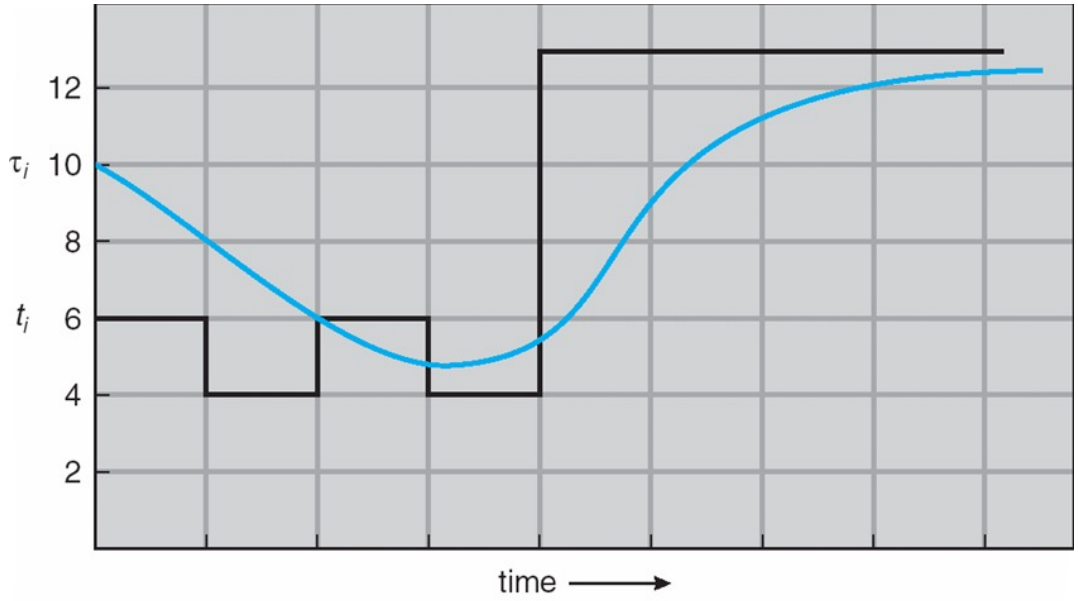
- $\alpha = 0.7$, $\tau_0 = 10$
- $\tau_1 = . * t_0 + . * \tau_0$
- $= . * 6 + . * 10 =$
- $\tau_2 = .7 * t_1 + .5 * \tau_1$
- $=$
- $\tau_3 = .7 * t_2 + .5 * \tau_2$

  $=$

  $\tau_4 = .7 * t_3 + .5 * \tau_3$

  $=$
- $\tau_5 = .7 * t_4 + .5 * \tau_4$

  $=$



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

Preemptive SJF: Shortest-remaining-time-first

- A new process arrives at the ready queue while a previous process is still executing.
  - The next CPU burst of the newly arrived process is shorter than what is left of the currently executing process.
  - A preemptive SJF algorithm preempts the currently executing process

- In the next example Process *P*1 is started at time 0, since it is the only process in the queue.

- Then process *P*2 arrives at time 1. The remaining time for process *P*1 (7 milliseconds) is larger than the time required by process *P*2 (4 milliseconds)

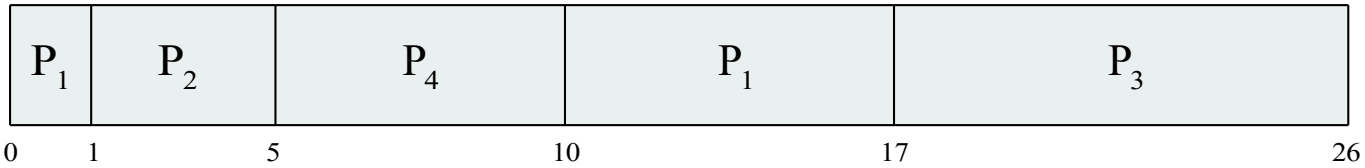
- Process *P*1 is preempted, and process *P*2 is scheduled.

Preemptive SJF: 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

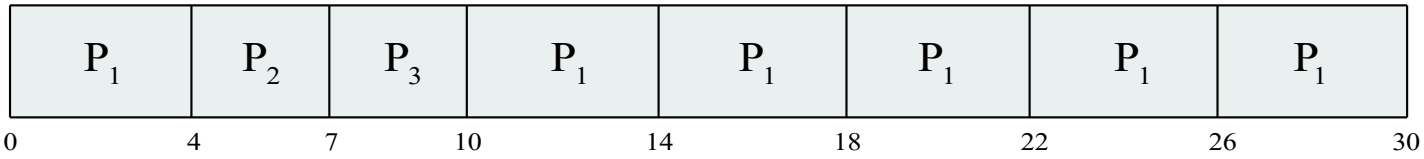
Round Robin (RR)

- Similar FCFS as new process enter at the tail of the ready queue and process at the head is next for the CPU
- However, it is a preemptive FCFS
- 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 tail of the ready queue.

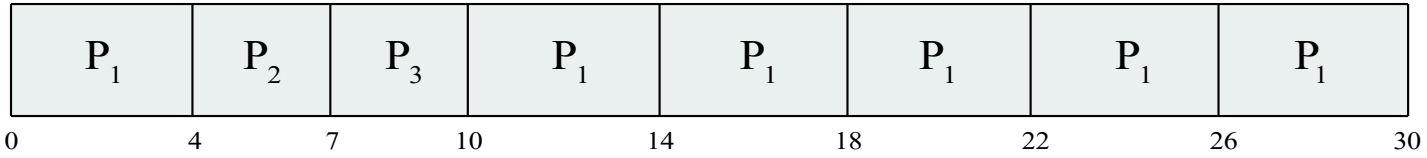
Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



Example of RR with Time Quantum = 4



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

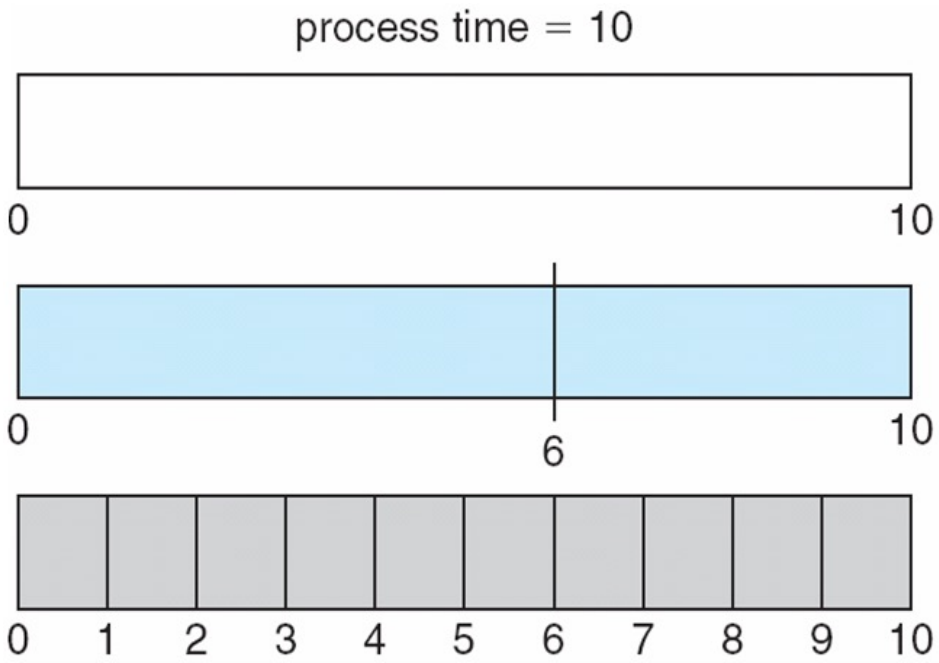
Item	Time		Time in human terms
Processor cycle	0.5ns	2Ghz	1 second
Memory access	15ns		30 seconds
Context switch	5,000ns	5μs	167 minutes
Disk access	7,000,000ns	7ms	162 days
One keystroke	100,000,000ns	100ms	6.3 years

Table 1: Time scales (A fast typist can type a keystroke every 100 milliseconds)

Round Robin (RR)

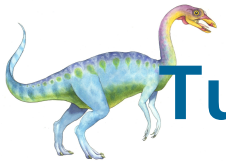
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once.
- No process waits more than (*n*-1)*q* time units.
- Performance depends on the size of the time quantum
  - *q* large ⇒ performance same as FCFS
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Quantum Time and Context Switch Time



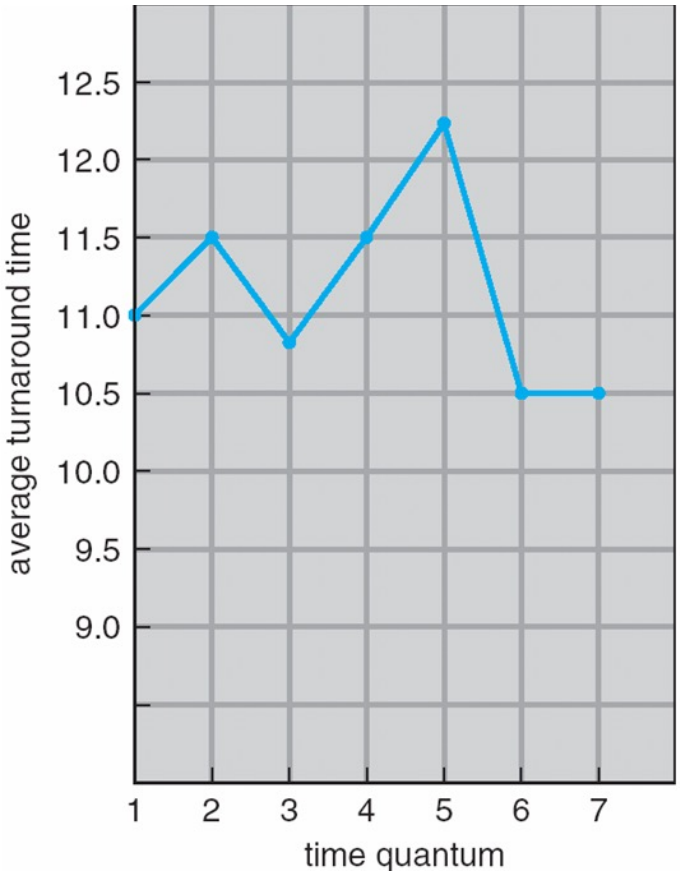
quantum	context switches
12	0
6	1
1	9

- Process time = 10, quantum time = 12, then 0 context switch
- If quantum time = 1, needs 9 context switches for a process time of 10



Turnaround Time Varies With Time Quantum

- Turnaround time – amount of time to execute a particular process
- See detail calculations



process	time
P_1	6
P_2	3
P_3	1
P_4	7

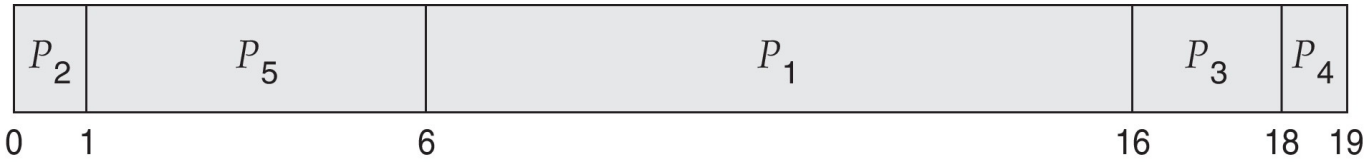
Priority Scheduling

- 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

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

Priority Scheduling

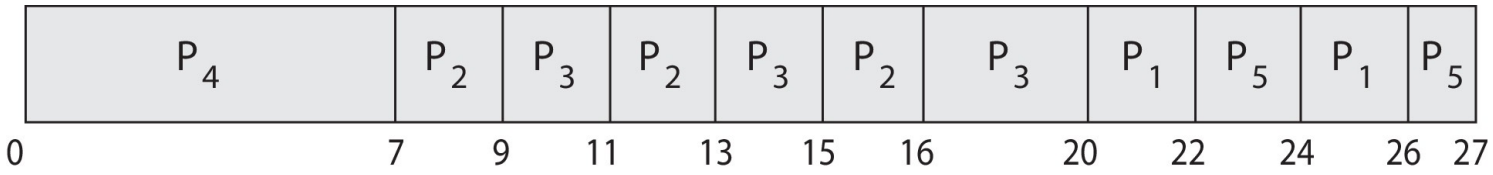
- Problem ≡ **Starvation** – low priority processes may never execute
  - IBM 7094 at MIT in 1973 they found a process that had been submitted in 1967 but not yet scheduled
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Priority Scheduling w/ Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin

- Gantt Chart with time quantum = 2

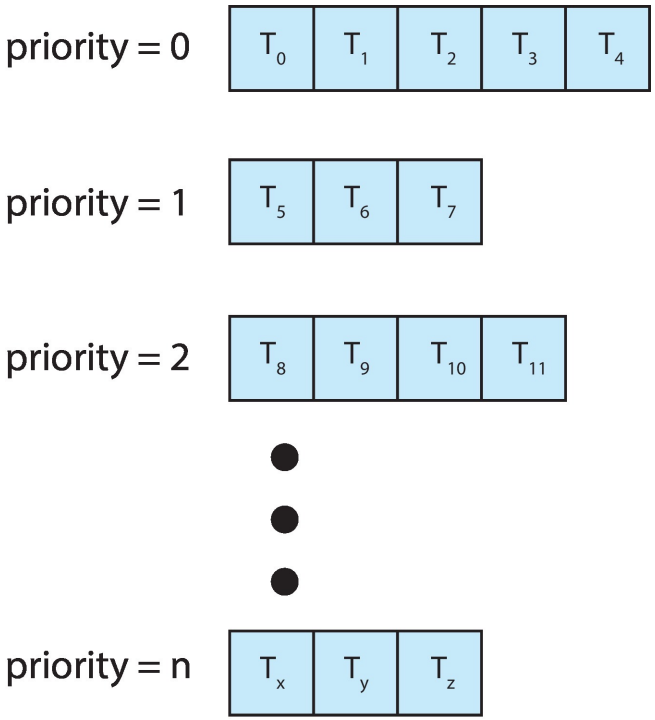


How priorities are defined

- Internally or externally
- Internally:
  - Use criteria or quantities inside the OS such as
    - time limits, memory requirements, number of open files, ratio of average I/O burst to CPU burst
- Externally:
  - Use criteria outside the OS such as
    - Importance of the process, type and amount of funds paid for computer use, dept sponsoring the work, others…

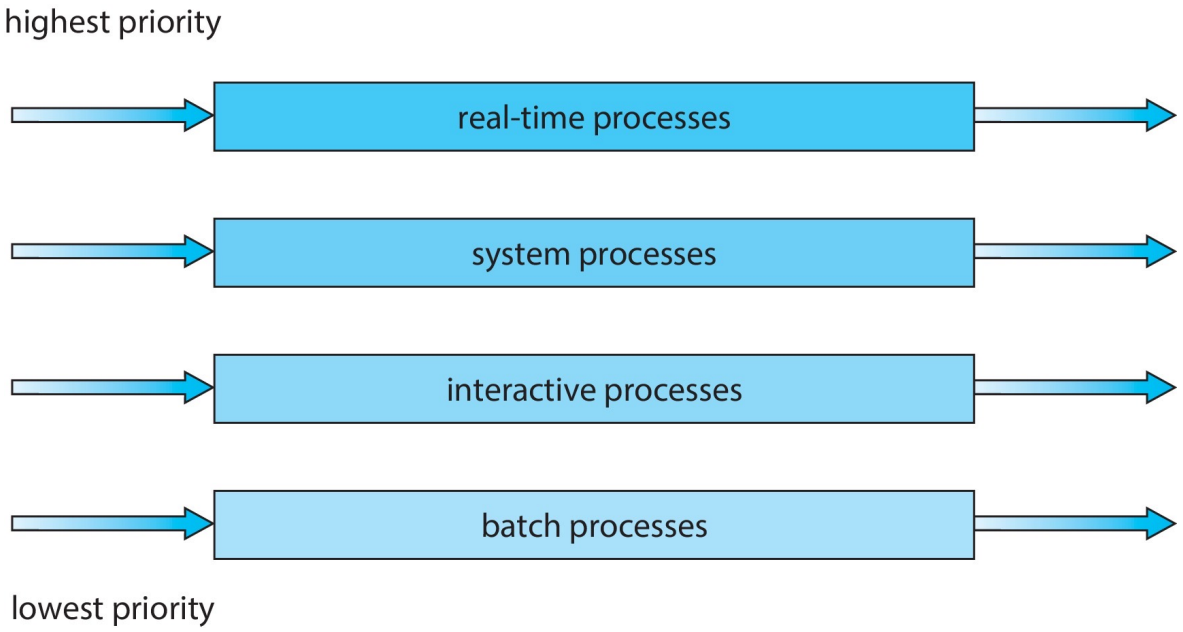
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type



Multilevel Feedback Queue

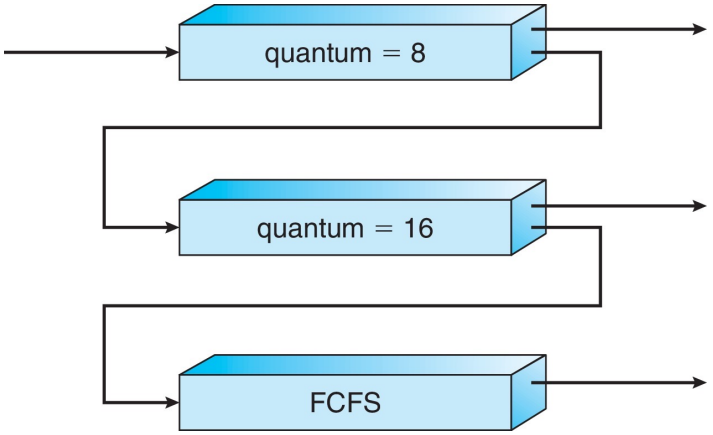
- Allow processes to move between the various queues
- Intend is to separate processes according to their CPU bursts:
  - CPU bound processes (with large burst time) get demoted to lower priority queues
  - I/O bound and interactive processes (short CPU burst) move into higher priority queues
- Each queue has its scheduling strategy adapted to the type of processes entering the queue

Multilevel Feedback Queue

- A process can move between the various queues.
- 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
- Aging can be implemented using multilevel feedback queue

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 process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads 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 macOS 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");
   }
```

Pthread Scheduling API

```
   /* 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);
}
```

Operating System Examples

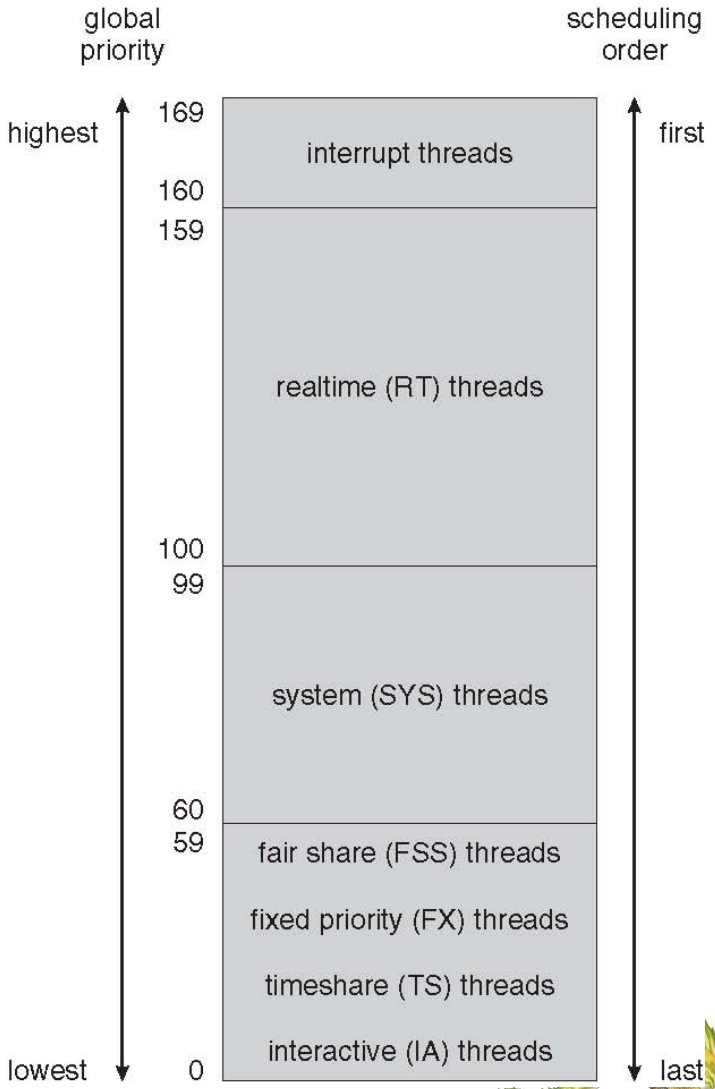
- Solaris scheduling
- Windows scheduling
- Linux scheduling

Solaris

- Preemptive priority-based scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Each class has a pre-defined range of priorities
- A thread belongs to only one class
- Each class has its own scheduling algorithm
- The time sharing class is a multi-level feedback queue

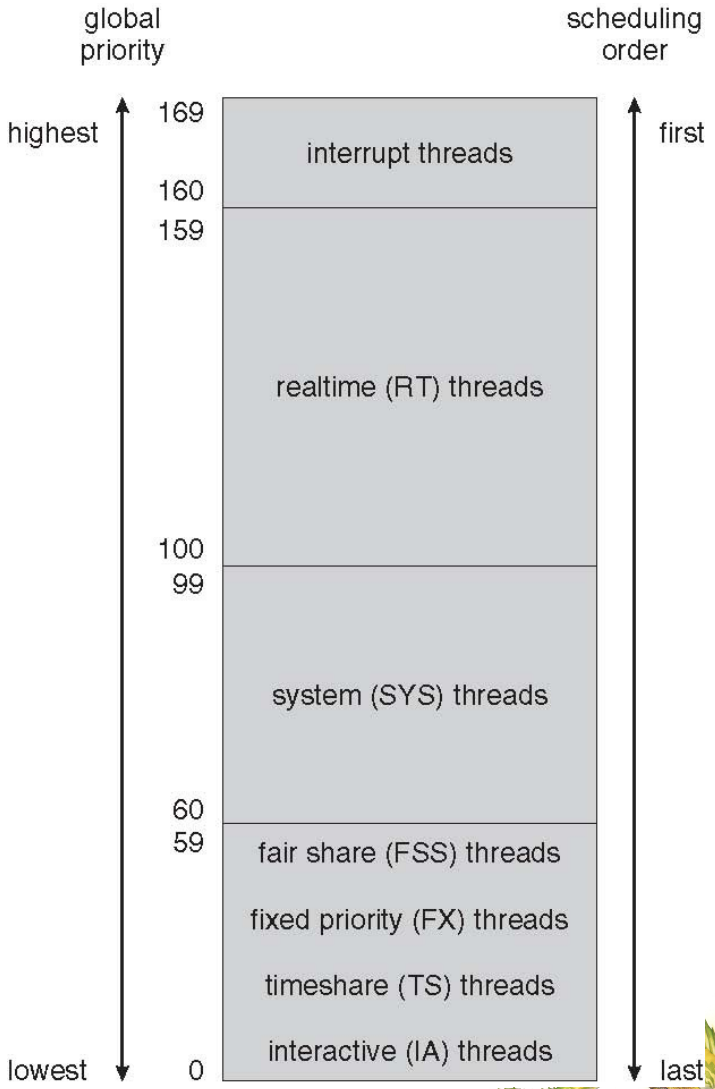
Solaris: Global Priority

- Within each class there are different priorities and different scheduling algorithms.
- Kernel threads (such as the scheduler) are run in the system class
- Threads in the real time class are given the highest priority
  - A real-time thread (i.e., audio, video processes) will run before threads in any other class
  - In general, **few** threads belong to the real-time class.



Solaris: Global Priority

- The default scheduling class for a thread is time sharing.
- The scheduling policy for the time-sharing class dynamically alters priorities and assigns time slices (quantum) of different lengths using a multilevel feedback queue.
- There is an inverse relationship between priorities and time slices. The higher the priority, the smaller the time slice; and the lower the priority, the larger the time slice



Solaris

- The scheduling policy for the time-sharing and interactive threads. Classes **dynamically alters priorities** of threads using a **multilevel feedback queue**

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

Inverse relationship between priorities and time slices

CPU-bound processes have lower priorities

Interactive processes have good response time

New priority of a thread that has used its entire time slice

Penalize CPU-bound processes

Priority of a thread that is returning from Waiting for I/O

Provide good response time for interactive processes

Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Windows Scheduling

- Windows scheduler is a **priority-based**, **preemptive** scheduling algorithm
- Use a 32 priority levels to determine the order of threads execution
  - Uses a queue for each scheduling priority
  - Traverses the set of queues from highest to lowest until it finds a thread that is ready to run
- Priorities divided into two classes:
  - The variable class contains threads with priority 1 to 15 and the real-time class contains threads from priority 16 to 31

Determining threads priority

- The priority of each thread is determined by the following criteria:
  - The priority class of its process
  - The priority level of the thread within the priority class of its process
- The *priority class* and *priority level* are combined to form the *base priority* of a thread.

Process priority classes

- Each process belong to one of 6 priority classes:
  1. REALTIME_PRIORITY_CLASS
  2. HIGH_PRIORITY_CLASS
  3. ABOVE_NORMAL_PRIORITY_CLASS
  4. NORMAL_PRIORITY_CLASS
  5. BELOW_NORMAL_PRIORITY_CLASS
  6. IDLE_PRIORITY_CLASS
- Except for real time processes, process priorities in other classes can vary, so these processes belong to the variable class.

Thread priority levels

- Each thread has one of the following relative priority levels:
  - TIME_CRITICAL
  - HIGHEST
  - NORMAL
  - BELOW_NORMAL
  - LOWEST
  - IDLE
- Windows overall thread priorities depends on the process priority and relative class of the thread

Windows thread priority

- Windows processes can belong to one of 6 priority classes
- Within a priority class, a thread has a relative priority level

	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

Windows Scheduling

- New processes are typically members of the `NORMAL_PRIORITY_CLASS`.
- The initial priority of a thread is typically the **base priority** (which is the normal relative priority) of the process the thread belongs to
  - For example, if a process belongs to high priority, then a new thread priority for this process will be 13

PROCESSES

THREADS

	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

Priorities

- Threads priorities are divided into two classes:
  - the **variable class** contains threads having priorities from1 to 15
  - the **real-time class** contains threads with priorities ranging from 16 to 31. (There is also a thread running at priority 0 that is used for memory management.)
- The scheduler uses a queue for each scheduling priority and traverses the set of queues from highest to lowest until it finds a thread that is ready to run.

	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

Pre-emptive scheduling

- A thread selected to run will run until it is either
  - preempted by a higher-priority thread
  - it terminates
  - its time quantum ends
  - it calls a blocking system call, such as for I/O.
- If a higher-priority real-time thread becomes ready while a lower-priority thread is running, the lower-priority thread will be preempted.
- This preemption gives a real-time thread preferential access to the CPU when the thread needs such access.

Multilevel feedback

- When a thread's time quantum runs out and its priority is lower, then the thread is interrupted
  - Its priority is lowered to limit the CPU consumption of CPU-bound thread.
- However, the priority of a thread is never lowered below the base priority of the thread to which it belongs (the normal thread priority)
  - For example, the priority of a thread belonging to a process with high priority can never go lower than priority 13
- The function SetThreadPriority() can set threads to lower 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

Multilevel feedback

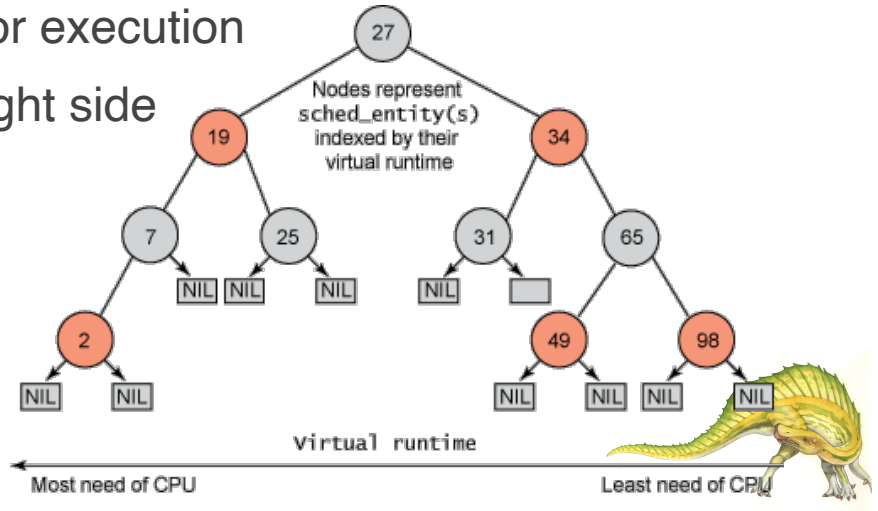
- When a variable-priority thread is released from a wait operation
  - Its priority is increased. Tend to give good response times to interactive threads
    - Increases more when the I/O is a keyboard strike
    - Increases less when the I/O is file access

Linux

- Like Solaris and Windows, the Linux scheduler is a **preemptive**, **priority-based** algorithm.
- Has two separate class of processes: **normal** and **real time** (unlike Solaris and Windows which have 6 classes)
- Each class has a different scheduling algorithm
- Real-time tasks are assigned **static** priorities
- The "normal" class use the **completely fair scheduling** (CFS) algorithm
- CFS is designed to favor interactive tasks

The CFS scheduler

- Linux CFS does not have priority queues, rather it uses a red-black tree where nodes in the tree are tasks ready to be scheduled
- CFS records in nanosecond the time a tasks has run, this is called **vruntime** (virtual runtime)
- The scheduler tracks the vruntime for all tasks.
  - The lower a task's vruntime, the more deserving the task is for scheduling on the CPU
- A task ready to execute is inserted in the red-black tree in a position according to its vruntime, the smallest the vruntime the more on the left of the RBT the task will be
- The leftmost node is scheduled next for execution
- Pre-empted tasks are placed on the right side of the tree



Target latency

- CFS does not have fixed timeslices (run robin) and explicit priorities.
- The amount of time for a given task on a processor is computed dynamically
- The **target latency** is an elapse time during which all the tasks must have got access to the CPU
- For example, if the target latency is 20ms, then all the tasks will execute inside the 20 ms
- In the idealized case where there is *n* "runnable" tasks, then each task gets 1/*n* time slice of the of the target latency, if *n = 4,* then each task get *5ms* time slice
- If the number of runnable tasks double (from 4 to 8) then the time slice for each task becomes 20/8 ms
- In this idealized context, the first task to run will be the one that has the smallest vruntime

Impacts of "nice" on priorities

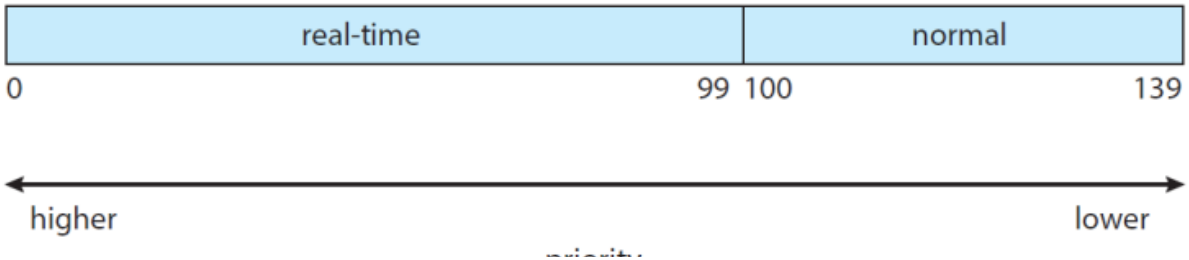
- The priority of a task can be modified by the nice value.
  - The range of nice values [-20,19].
  - Default of nice is 0, if nice > 0, a task has a lower priority.
  - If nice < 0, the task gets a higher priority
- Nice impacts the scheduling priority as follow:
  - If a task has nice = 0, its vruntime = its actual physical run time, for example 200ms
  - If a task has nice > 0, its vruntime > its actual physical run time
  - If a task has nice < 0, its vruntime < its actual physical run time
- An interactive task, tends to spend a lot of time in I/O queues, it is I/O-bound; hence, such a task tends to have a relatively low vruntime,
  - which tends to move the task towards the front of the scheduling line.

Impacts of "nice" on time slices

- Nice values impact time quantas as well
- Nice can be determined by the user with the command nice
  - Nice -10 command will lower the priority of "command"
- The scheduler also change the priorities of tasks
  - Tasks with shorter I/O times are often more CPU-bound, and will have adjustments closer to +5 after execution
    - To have shorter time quantas.
  - Tasks that are interactive are more likely to have adjustments closer to -5.
- Nice values change the weight a task has in the determination of the time quantas for each task in the latency target.
  - Assume 3 tasks T1, T2, T3, with respective nice -5, 0, 5. T1 may have weight 5, T2 = 3 and T1 = 2. Total of weights = 10. Thus, T1 gets ½ of the latency target, T2 get 3/10 and T1 get 1/5
  - If target latency = 20ms, T1 runs 10ms, T2 = 6ms, T3 = 2ms

Relations between real-time and normal

- Real-time tasks are assigned static priorities within the range of 0 to 99
- Normal tasks are typically assigned priorities in the range between 100-139

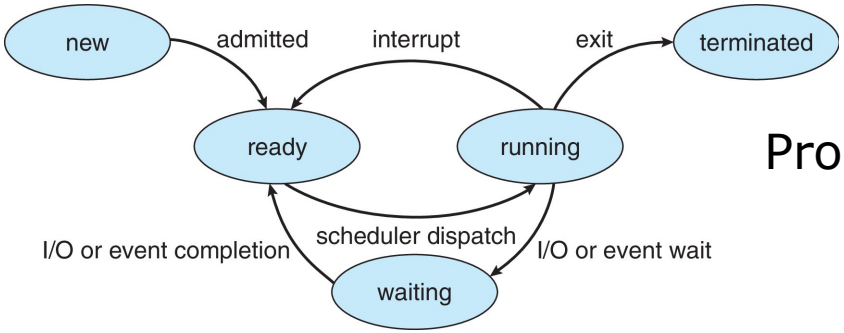


- These ranges may vary across different Linux flavors
- However, in all cases, all real-time tasks must be executed (completed, or idle) before any normal task can run on the CPU

Ubuntu priority levels

- See the priority of tasks Ubuntu: ps –l
- Or “top”
- Different output of the priority levels for same tasks
- Linux implementations are required to have minimum 32 real-time priority levels
  - 40 levels dynamically changing priorities

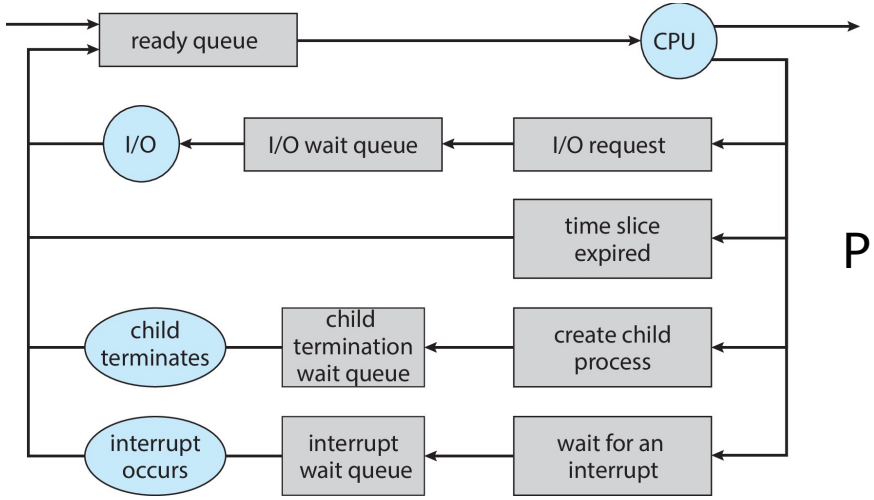
Summary on scheduling



Process states

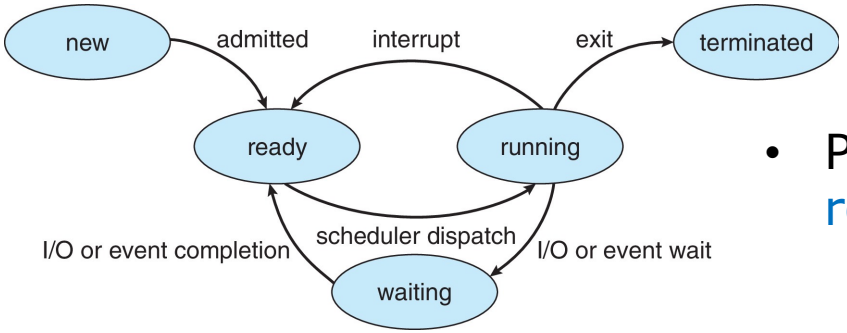
Process execution cycle

CPU	I/O	CPU	I/O



Process scheduling

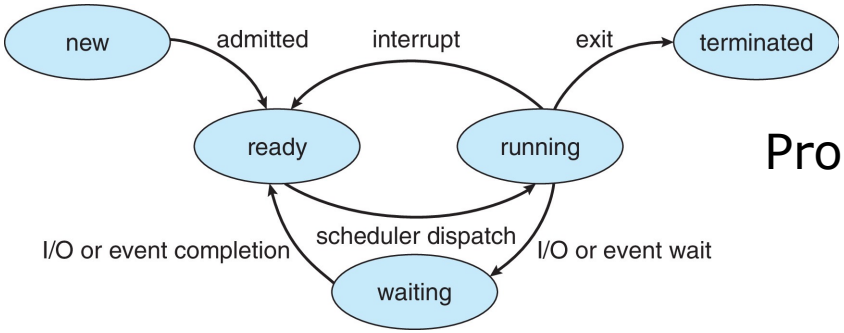
Summary on scheduling



- Process have states which are recorded into their PCB

1. What are two scheduling events that may interrupt a process cause it to return directly to the ready queue?

Summary on scheduling



Process states

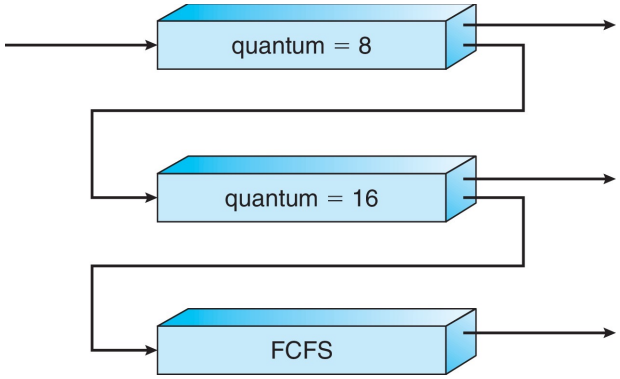
Process execution cycle

CPU	I/O	CPU	I/O

2. Can a process change states inside a same CPU burst?

3. If yes, what are the states in which a process can be while been in the same CPU burst?

Summary on scheduling



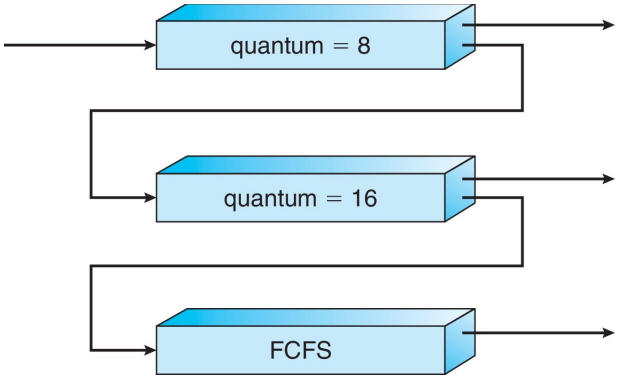
4. If process P has CPU burst of 12 units of time, in which ready queue process P goes once it is de-scheduled by the RR of the first queue?

5. For its next CPU burst, in which queue process P goes after completing its last 4 units of its current CPU burst time?

6. In which queue process P goes if it performs an I/O after executing for 6 units of time?

7. In which ready queue process P returns once it has completed its I/O?

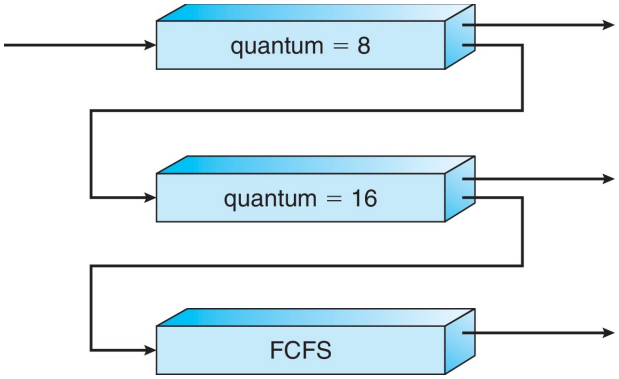
Summary on scheduling



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

- In multilevel scheduling, usually each queue has a different scheduling algorithm. On the left, 3 queues, 2 RR with different quantum time and a FCFS
- However, implicitly, queues represent different priorities, inside a same CPU burst
- In Solaris, queues have an explicit priority and different quantum times

Summary on 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

8. In Solaris, if a process does not complete it CPU burst inside the quantum time of the queue, does it priorities increases or decreases?

9. If a process P is in queue with priority 25, and its CPU burst is 150 units, in which queue it goes once it has exhausted the quantum time of queue 25?

10. If a process P is in queue with priority 25, and its CPU burst is 70 units, in which queue it goes once its I/O is completed?

11. If a process is given a fixed priority, can it be scheduled in a multilevel feedback queue?

End of Section 5