# Scheduling (CS-351)

# Agenda

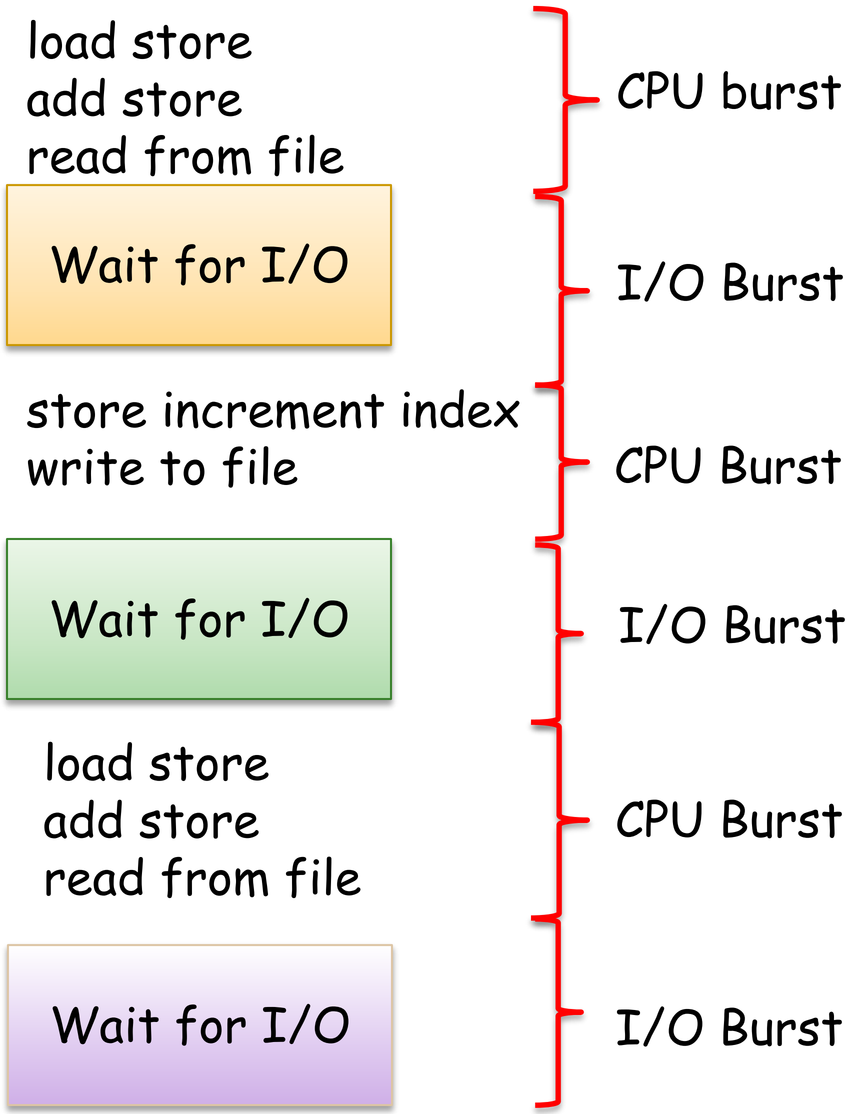
- Introduction to CPU scheduling.
- CPU Scheduling Algorithms: FCFS, SJF, SRT, RR, Priority queue, Multi-level queue
- Pthreads API

# Basic Concepts: CPU Scheduling

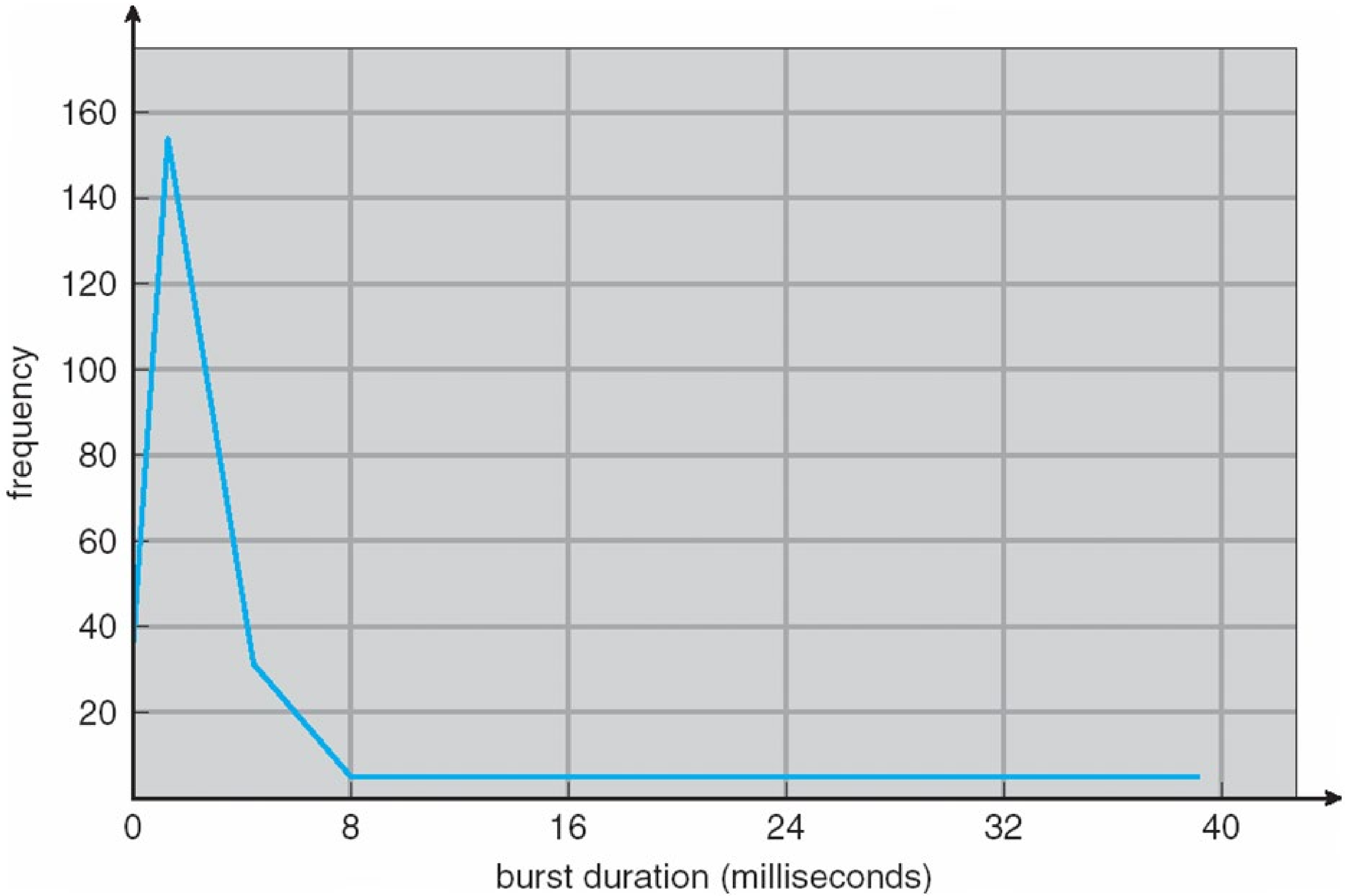
- CPU is one of the primary computer resources.
- CPU scheduling: selecting the next process for execution on the CPU once the current process leaves the CPU idle.
- The goal of CPU scheduling is to maximize the degree of multiprogramming i.e., having some process running at all times.
- The success of CPU scheduling depends on an observed property of the processes:
    - CPU execution
    - I/O waiting

# CPU-I/O Burst Cycle

- Processes alternate between CPU bursts (i.e. executing on the CPU) and I/O bursts (performing I/O).



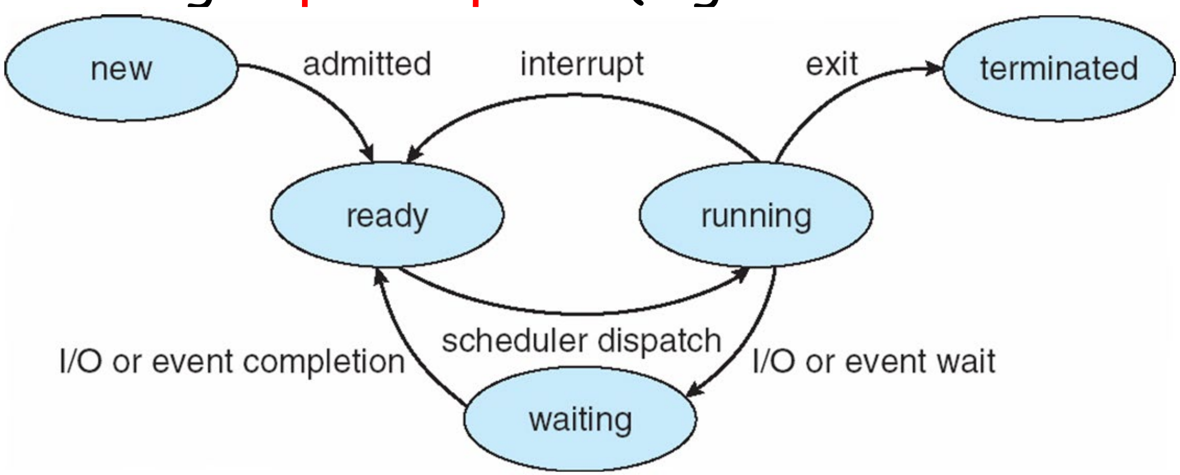
# Histogram of CPU-burst Times




- Large number of short CPU bursts and a small number of long CPU bursts.

# The CPU Scheduler (a.k.a. The Short-Term Scheduler):

- Selects a process from the processes in memory that are ready to execute and allocates the CPU to the process.
- CPU scheduling decisions occur when:
  - 1. When a process switches from the running to the waiting state.
  - 2. When a process switches from the running to the ready state.
  - 3. When a process switches from the waiting state to the ready state.
  - 4. When a process terminates.
- Scheduling under 1 and 4 is nonpreemptive (e.g. Windows 3.11)
- All other scheduling is preemptive (e.g. Windows 95 and up and MAC OSX).



# Preemptive vs Non-preemptive Scheduling

- Preemptive scheduling: process executing on the CPU can be interrupted in order to make way for another process.
- Non-preemptive scheduling: once a process gets the CPU, it runs to completion and cannot be interrupted.

# Dispatcher

- A module that gives control of the CPU to the process selected by the scheduler.
  - Switches the context
  - Switches to user mode
  - Jumps to the proper location in the program to restart that program.
- Invoked during every process switch.
- Dispatcher latency: the time it takes for the dispatcher to stop one process and start another – should be minimized!

# Scheduling Criteria

- Performance metrics of scheduling algorithms:
  - CPU utilization: keeping the CPU as busy as possible.
  - Throughput: the amount of processing that can be performed per unit of time.
  - Waiting time: the amount of time the process spends waiting in the **ready** queue.
  - Response time: amount of time it takes from when a request was **submitted** until the **first** response is produced.
  - Running time: the amount of **execution** time of the process
  - Running time = CPU burst time
  - Turnaround time: how long does it take to execute a process?
  - Turnaround time = waiting time + running time

# Scheduling Algorithms: First-Come First-Served

- The process that requests the CPU first is allocated the CPU first.
- Example:

| Process | Burst Time (millisecs) |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Suppose the processes arrive in the order: $P_1$, $P_2$, $P_3$:
  - Gantt chart (shows starting and ending times):


  - Waiting times: 0 for $P_1$, 24 for $P_2$, and 27 for $P_3$.
  - Average waiting time: (0 + 24 + 27) / 3 = 17 ms

# Scheduling Algorithms: FCFS-cont.

- Average Turnaround Time (ATT) for the three processes using FIFO scheduling is 27.

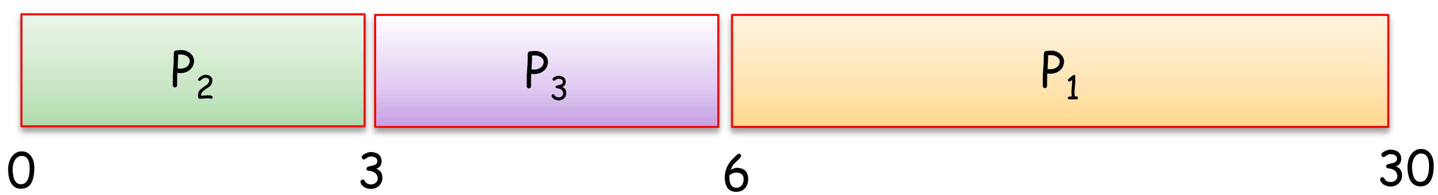
| | turnaround time | ATT( average turnaround time) |
|---|---|---|
| P1 | 0+24=24 | (24+27+30)/3 = 27 |
| P2 | 24+3=27 | |
| P3 | 27+3=30 | |

# Scheduling Algorithms: First-Come First-Served

- The process that requests the CPU first is allocated the CPU first.
- Example:

| Process | Burst Time (millisecs) |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Suppose the processes arrive in the order: $P_2$, $P_3$, $P_1$:
  - Gantt chart (shows starting and ending times):




  - Waiting times: 0 for $P_2$, 3 for $P_3$, and 6 for $P_1$.
  - Average waiting time: (6 + 0 + 3) / 3 = 3 ms
  - ATT?

# Scheduling Algorithms: First-Come First-Served

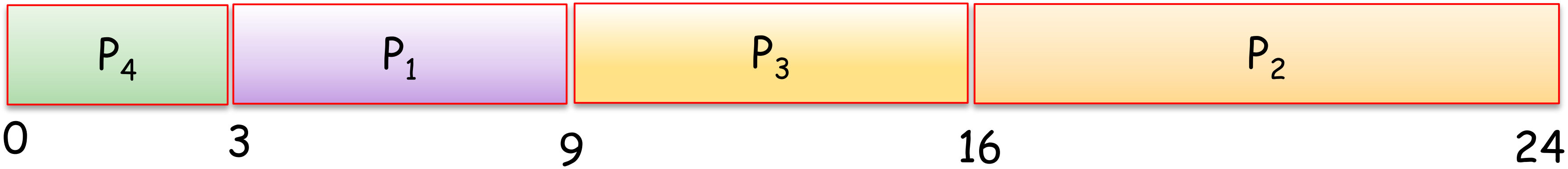
- Advantages: simple to implement and understand.
- Disadvantages:
  - Average waiting time is often quite long.
  - Convoy effect: short process waiting behind a long process.
  - Non-preemptive: once the CPU is allocated to a process, that process keeps the CPU until it requests I/O or terminates.

# Scheduling Algorithms: Shortest-Job-First

- Associates with each process the length of its next CPU burst. Uses these lengths to schedule the process with the shortest time. Can either be preemptive or nonpreemptive
- Example (non-preemptive):

| Process | Burst Time (millisecs) |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

- Gantt chart:




- Waiting time: 3 for $P_1$, 16 for $P_2$, 9 for $P_3$, 0 for $P_4$.
- Average waiting time: (3 + 16 + 9 + 0) / 4 = 7ms (10.25 ms with FCFS).
- ATT?

# Scheduling Algorithms: Shortest-Job-First

- Unlike nonpreemptive, will interrupt the currently executing process. Also known as shortest-remaining-time-first (SRT).
- Example (preemptive):

| Process | Arrival Time | Burst Time (millisecs) |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |


- Waiting time: [(10-1) + (1-1) + (17-2) + (5-3)]/4 = 26/4 = 6.5 ms
- ATT?

# Scheduling Algorithms: Shortest-Job-First

- Advantage: is optimal – gives minimum average waiting time for a given set of processes.
- Disadvantages: we can only estimate the length of the next CPU request – a difficult task!
- One way to estimate the CPU burst of the process is to use exponential averaging.
- What if an emergency job but very long?

# Student Participation: Determining the remaining time.

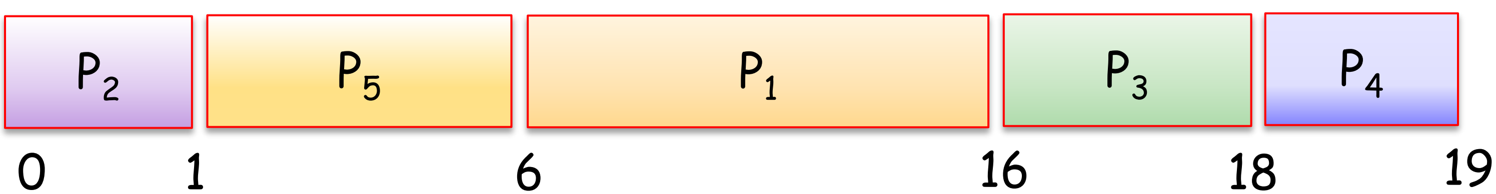
- The remaining time of a process depends on __________.
- 1) arrival time
  - True
  - False
- 2) total CPU time
  - True
  - False
- 3) real time in system
  - True
  - False
- 4) attained CPU time
  - True
  - False

# Scheduling Algorithms: Priority Scheduling

- Priority is associated with each process. The CPU is allocated to the process with the highest priority. Can be preemptive or nonpreemptive.
- Example (smaller number = higher priority):

| Processes | Burst Time (millisecs) | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |

- All processes arrive at the same time.
- Gantt chart:



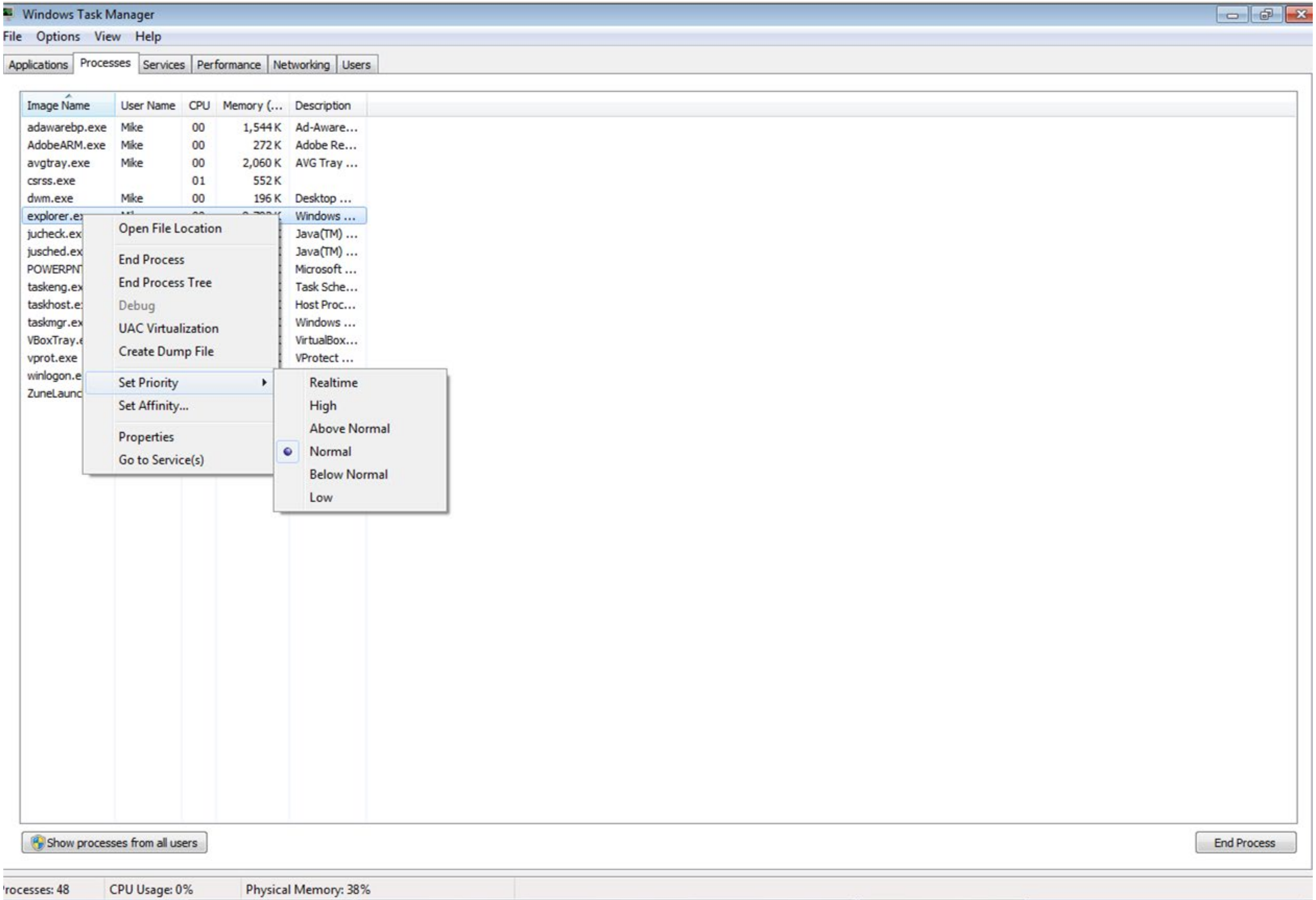

- Average waiting time: 8.2 ms
- ATT?

# Scheduling Algorithms: Priority Scheduling

- Preemptive priority scheduling will preempt the CPU if the newly arrived process has a higher priority than the currently running process.
- Starvation: low-level priority processes may have to wait indefinitely.
  - Example (rumor): when MIT shutdown the IBM 7094 in 1973, they found a low-priority process that has been submitted in 1967 and had not yet run!
  - Solution: aging: gradually increase the priority of processes that wait in the system for a long time.

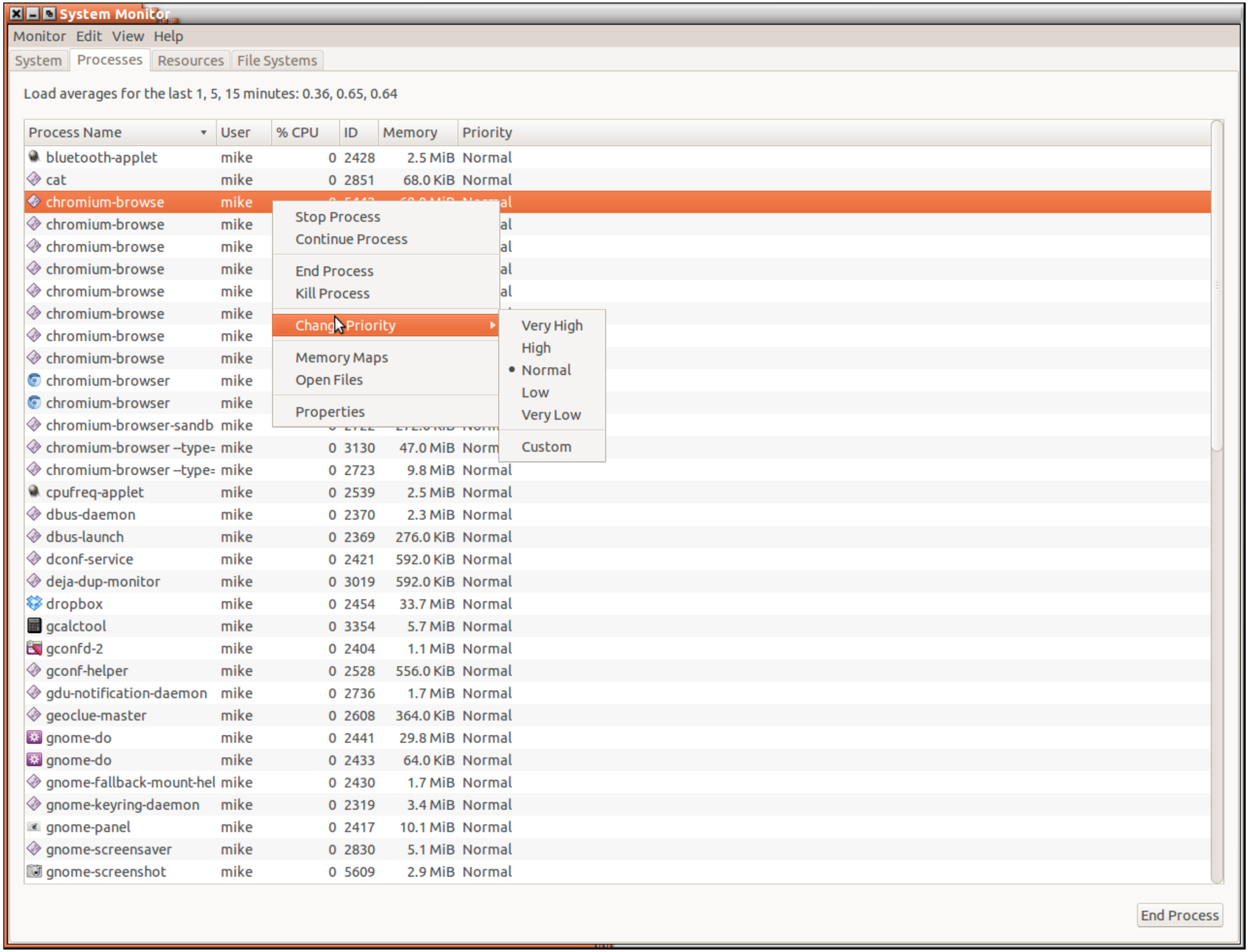
# Scheduling Algorithms: Priority Scheduling

- Process priorities in Windows:



# Scheduling Algorithms: Priority Scheduling

- Process priorities in Ubuntu Linux:




| Process Name | User | % CPU | ID | Memory | Priority |
|---|---|---|---|---|---|
| bluetooth-applet | mike | 0 | 2428 | 2.5 MiB | Normal |
| cat | mike | 0 | 2851 | 68.0 KiB | Normal |
| chromium-browse | mike | | | | |
| chromium-browse | mike | | | | |
| chromium-browse | mike | | | | |
| chromium-browse | mike | | | | |
| chromium-browse | mike | | | | |
| chromium-browse | mike | | | | |
| chromium-browse | mike | | | | |
| chromium-browse | mike | | | | |
| chromium-browser | mike | | | | |
| chromium-browser | mike | | | | |
| chromium-browser-sandb | mike | | | | |
| chromium-browser --type= | mike | 0 | 3130 | 47.0 MiB | Norm |
| chromium-browser --type= | mike | 0 | 2723 | 9.8 MiB | Normal |
| cpufreq-applet | mike | 0 | 2539 | 2.5 MiB | Normal |
| dbus-daemon | mike | 0 | 2370 | 2.3 MiB | Normal |
| dbus-launch | mike | 0 | 2369 | 276.0 KiB | Normal |
| dconf-service | mike | 0 | 2421 | 592.0 KiB | Normal |
| deja-dup-monitor | mike | 0 | 3019 | 592.0 KiB | Normal |
| dropbox | mike | 0 | 2454 | 33.7 MiB | Normal |
| gcalctool | mike | 0 | 3354 | 5.7 MiB | Normal |
| gconfd-2 | mike | 0 | 2404 | 1.1 MiB | Normal |
| gconf-helper | mike | 0 | 2528 | 556.0 KiB | Normal |
| gdu-notification-daemon | mike | 0 | 2736 | 1.7 MiB | Normal |
| geoclue-master | mike | 0 | 2608 | 364.0 KiB | Normal |
| gnome-do | mike | 0 | 2441 | 29.8 MiB | Normal |
| gnome-do | mike | 0 | 2433 | 64.0 KiB | Normal |
| gnome-fallback-mount-hel | mike | 0 | 2430 | 1.7 MiB | Normal |
| gnome-keyring-daemon | mike | 0 | 2319 | 3.4 MiB | Normal |
| gnome-panel | mike | 0 | 2417 | 10.1 MiB | Normal |
| gnome-screensaver | mike | 0 | 2830 | 5.1 MiB | Normal |
| gnome-screenshot | mike | 0 | 5609 | 2.9 MiB | Normal |

# Scheduling Algorithms: Priority Scheduling

- Priorities in Linux: each process has a nice value specifying its priority:
  - Nice values range between -20 and 19.
  - Lower values indicate higher priority.

- Setting nice values:
  - Example: start program ls with nice value of 19:
    - nice –n 19 ls
  - Example: change nice value of already running process with id 1234, to 15.
    - renice 15 -p 1234

# Scheduling Algorithms: Round-Robin (RR)

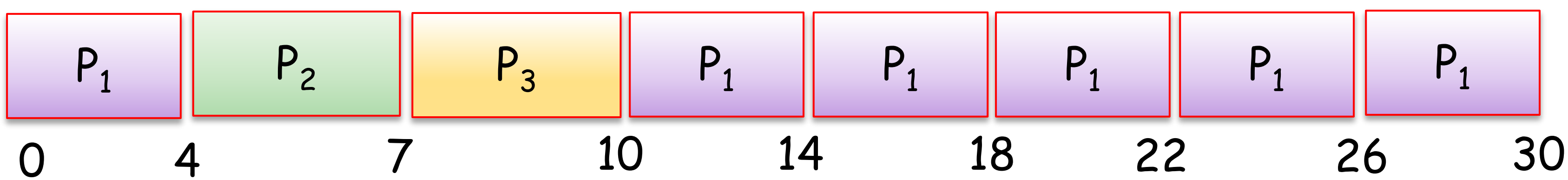
- Each process gets a small unit of CPU time i.e. time quantum. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Given n processes and time quanta of q, Each process waits no longer than (n – 1) * q time units.
- Performance is sensitive to the choice of the time quantum:
  - Very large: RR degenerates to first-come first-serve
  - Very small: q must be large in comparison to context switch latency, otherwise overhead is too high.

# Scheduling Algorithms: Round-Robin (RR)

- Example (quantum = 4 millisecs):

| Process | Burst Time (millisecs) |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

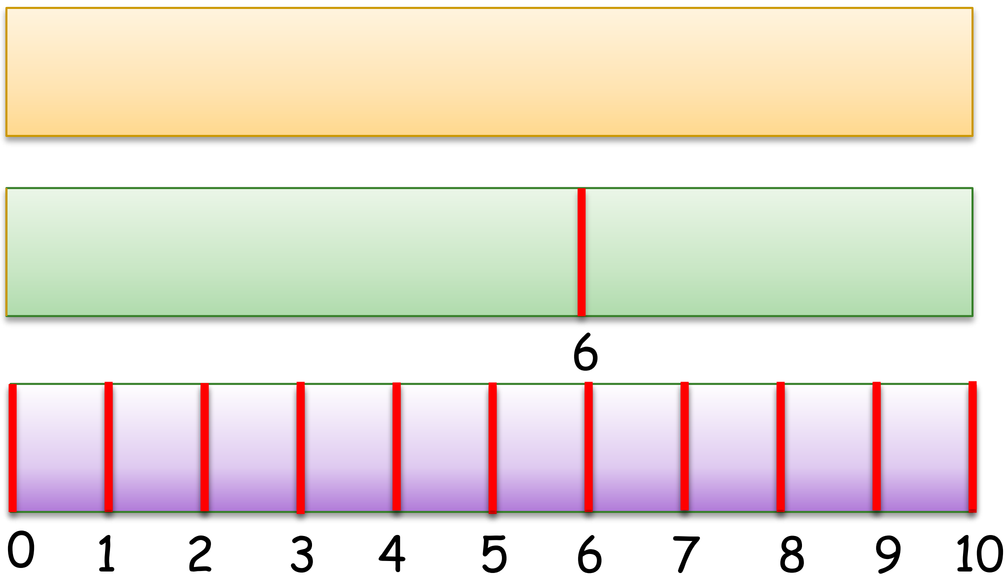
- Gantt chart:




- Waiting time: $P_1$ waits for 6 (10-4), $P_2$ waits for 4, and $P_3$ waits for 7.
- Average waiting time: 17/3 = 5.66
- ATT?

# Scheduling Algorithms: Round-Robin (RR)

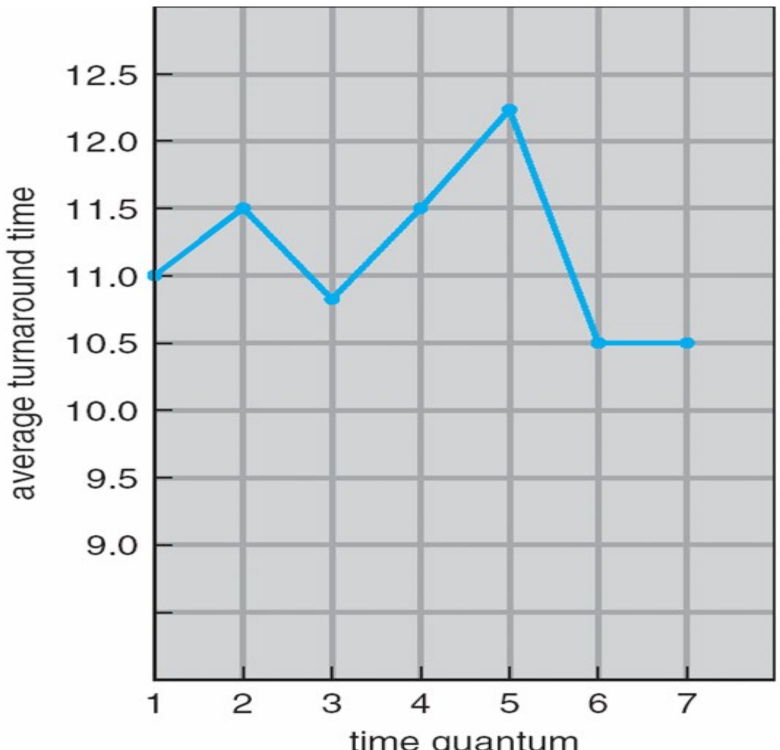
- How smaller quantum increases context switches:
  - Process time = 10




| Quantum | Context Switches |
| --- | --- |
| 12 | 0 |
| 6 | 1 |
| 1 | 9 |

# Scheduling Algorithms: Round-Robin (RR)

- How turnaround time varies with the time quantum:

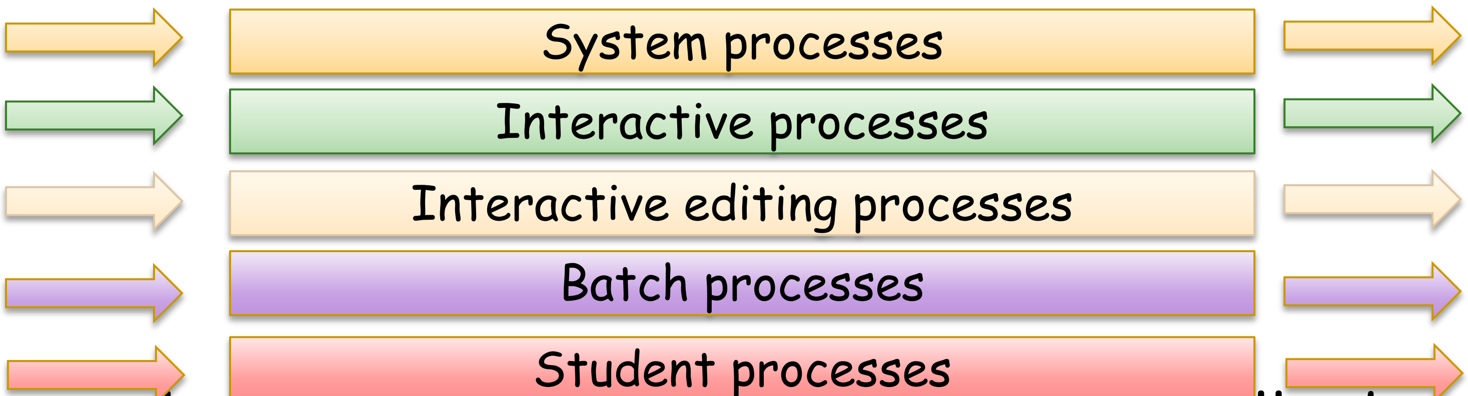



| Process | Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

- Generally, the average turnaround time can be improved if most processes finish their next CPU burst in 1 quantum.
  - Example: given 3 processes of 10 time units each and a quantum of 1, the avg. turnaround time is 29. Increasing the quantum to 10, reduces the avg. turnaround time to 20

# Scheduling Algorithms: Multilevel Queue

- Partitions the ready queue into several separate queues. Processes are permanently assigned to queues, generally based on some property of the process e.g. memory size, priority, or type.
- Each queue has its own scheduling algorithm.
- Example:



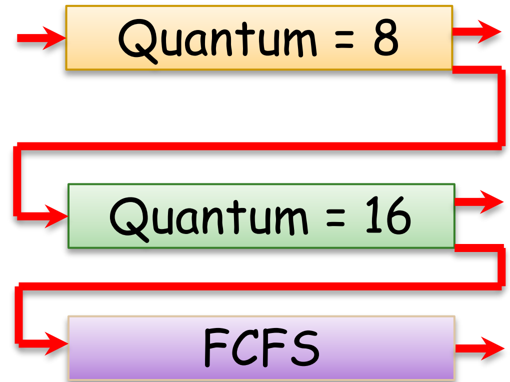

- Also, need to schedule among the queues (usually done using fixed-priority preemptive scheduling algorithm).
  - I.e., from which queue should the next be selected?

# Scheduling Algorithms: Multilevel Feedback Queue

- Similar to multilevel queue scheduling, but the process can move between queues.
- The scheduler is 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.

# Scheduling Algorithms: Multilevel Feedback Queue

- Example: 3 queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – First-Come First-Served




- Scheduling:
  - A new process enters queue $Q_0$ which uses round robin with quantum of 8.
  - When it gets the CPU, the job receives 8 milliseconds of running time.
  - If it does not finish in 8 milliseconds, the job is moved to queue $Q_1$ which uses round robin with quantum of 16.
  - At $Q_1$, the job receives additional 16 milliseconds.
  - If it still does not complete, it is preempted and moved to queue $Q_2$ which uses FCFS scheduling.

# Thread Scheduling: Pthreads

- User-level and kernel-level threads are scheduled differently.
- Many-to-one and many-to-many models, thread libraries schedule user-level threads to run on a light weight process (LWP) – a virtual processor on which the thread can be scheduled to run (or a kernel thread).
- Process-contention scope (PCS): User thread of a process competes for execution on a LWP (or kernel thread) with other user threads of the same process.
- System-contention scope (SCS): Kernel thread executing user threads of a particular process compete for execution on the physical CPU with other kernel threads executing user threads of the same process.

# Thread Scheduling: Pthreads

- Pthreads API enables the developers to specify either PCS or SCS during thread creation:
  - PTHREAD_SCOPE_PROCESS: schedules threads using PCS scheduling i.e. each thread is bound to an available LWP.
  - PTHREAD_SCOPE_SYSTEM: schedules threads using SCS scheduling i.e. on many-to-many systems will create and bind an LWP for each user-level thread.

# Thread Scheduling: Pthreads: Contention Scope

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM THREADS];
    pthread_attr_t attr;
    // Get the default attributes
    pthread_attr_init(&attr);
    //Set the scheduling algorithm to
    //PROCESS or SYSTEM
    pthread_attr_setscope(&attr,
    PTHREAD_SCOPE_SYSTEM);
    // Set the scheduling policy - FIFO, RT,
    or OTHER
    pthread_attr_setschedpolicy(&attr,
    SCHED_OTHER);
    // Create the threads
    for (i = 0; i < NUM _THREADS; i++)
            pthread_create(&tid[i],&attr,runn
    er,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)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```

# Thread Scheduling: Pthreads: Scheduling Policy

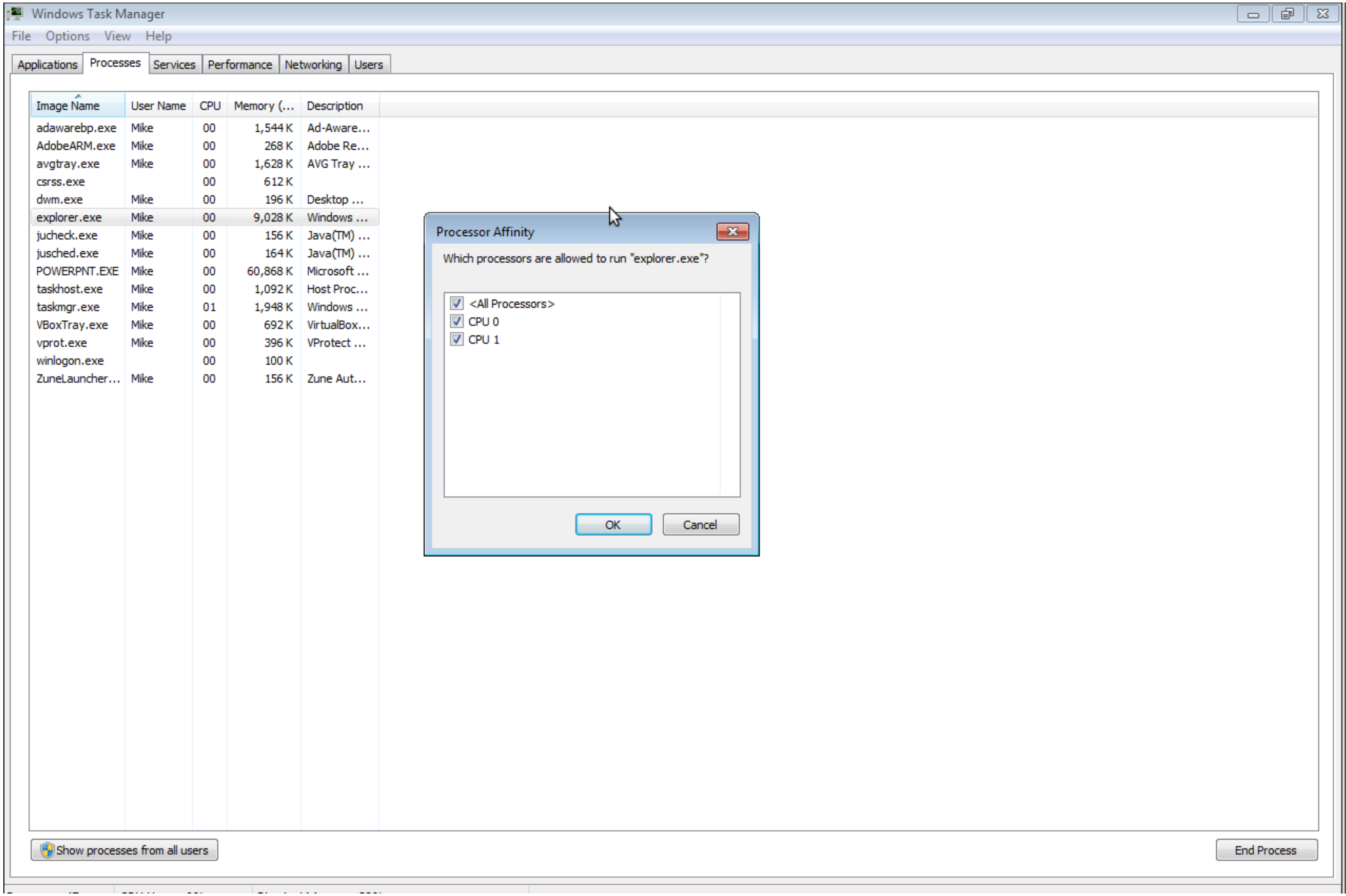
```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling policy - FIFO, RT,
    or OTHER */
    pthread_attr_setschedpolicy(&attr,
    SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)

        pthread_create(&tid[i],&attr,runn
    er,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)
{
    printf("I am a thread\n");
    pthread_exit(0);
}
```

# Multi-processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.
- Homogeneous processors within a multiprocessor.
- Asymmetric multiprocessing:
  - One processor handles all scheduling, I/O processing, etc. All other processors only execute user code.
  - Advantage: only one processor accesses the system data structures – don't need to worry about data sharing issues.
- Symmetric multiprocessing (SMP): each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes.
- Processor affinity: process has affinity for processor on which it is currently running.
  - soft affinity: a process can migrate between processors.
  - hard affinity: a process cannot migrate between processors

# Multi-processor Scheduling

- Process affinities in Windows:



# CPU Scheduling on NUMA Systems

- Non-Uniform memory access systems (NUMA): comprises combined CPU and memory boards. The CPUs on the board can access the memory on that board with less latency than on the other boards:
  - Scheduling goal: schedule a process on the CPU attached to the memory bank containing the process data.