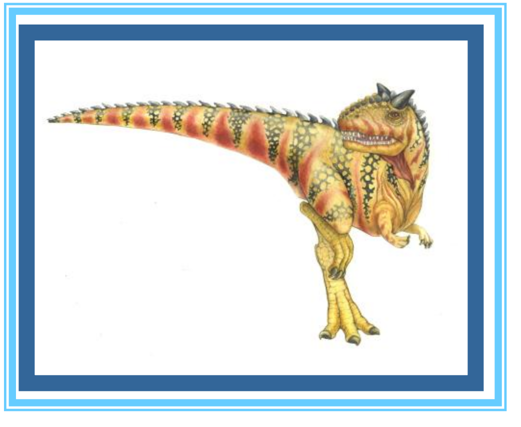
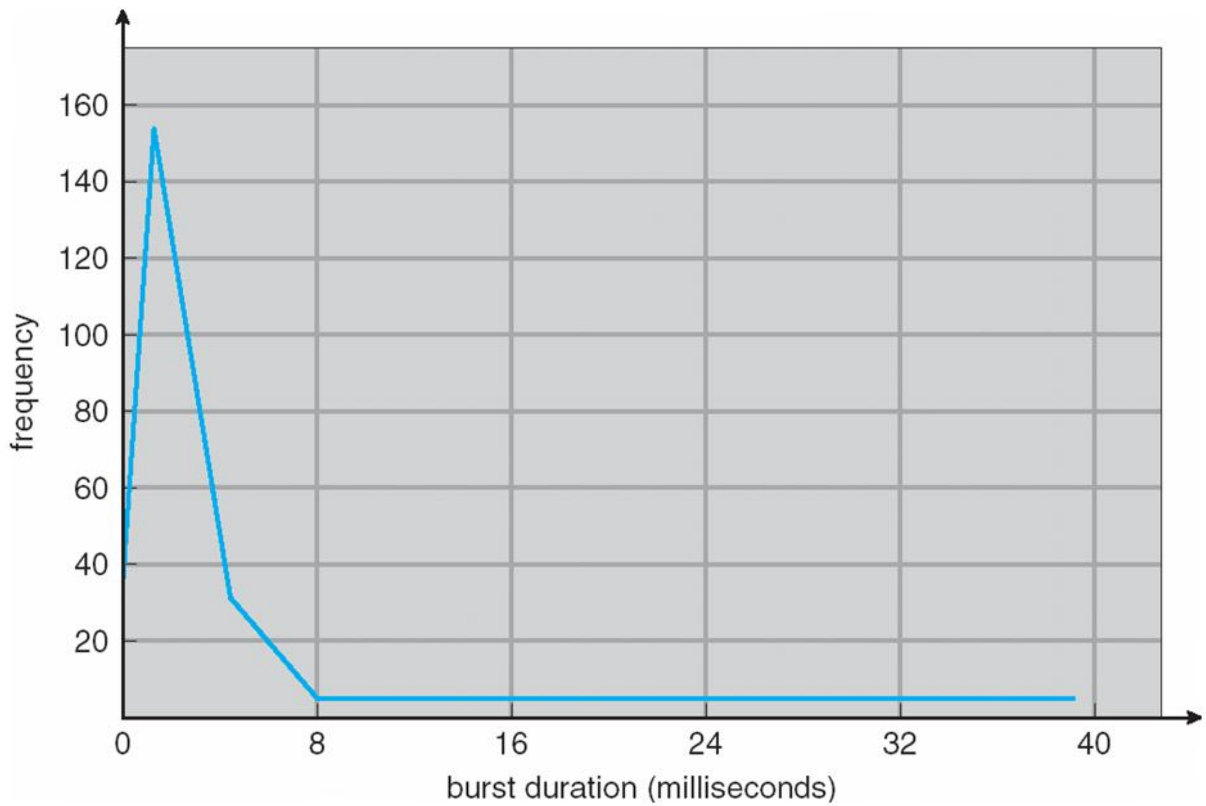
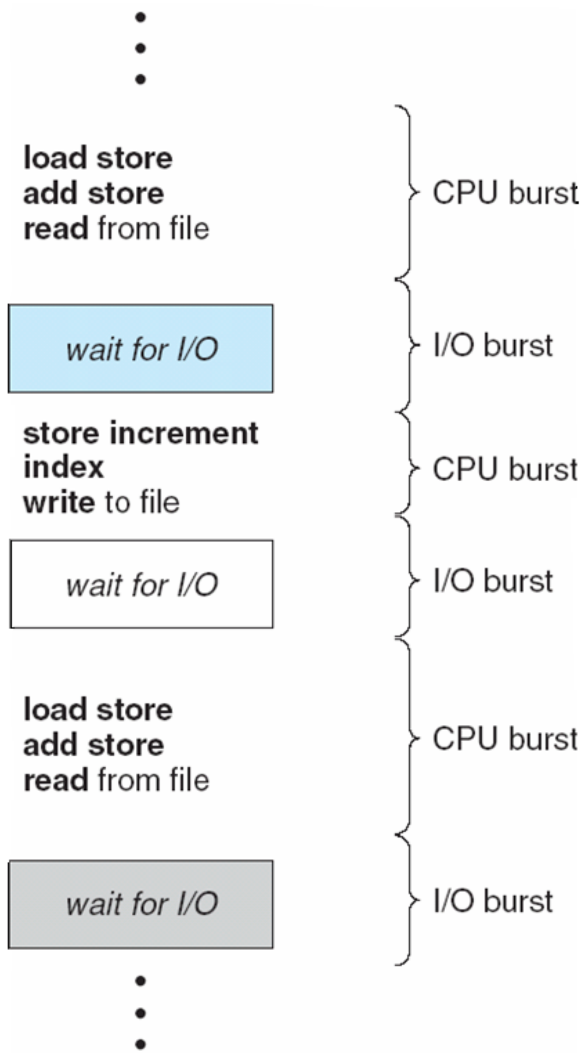
Chapter 6: CPU Scheduling



Histogram of CPU-burst Times



Alternating Sequence of CPU And I/O Bursts



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other 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

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible → Max
- **Throughput** – # of processes that complete their execution per time unit → Max
- **Turnaround time** – amount of time to execute a particular process → Min
- **Waiting time** – amount of time a process has been waiting in the ready queue → Min
- **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) → Min

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:

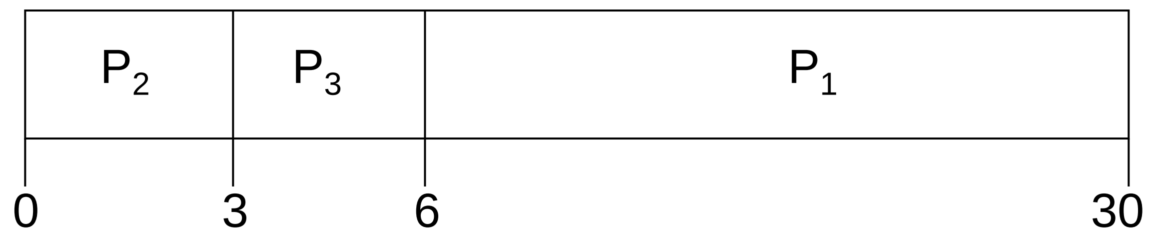
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

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

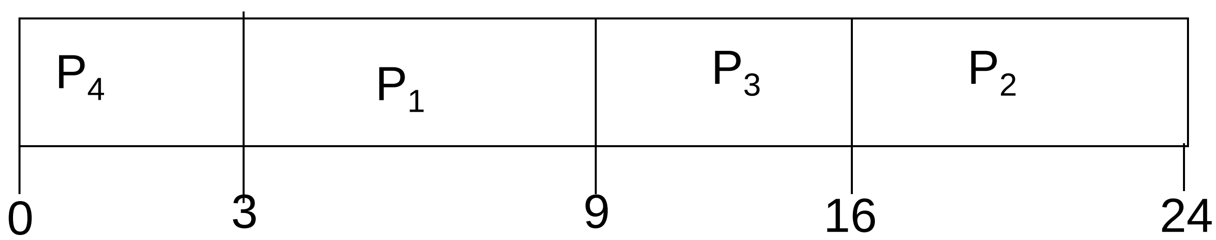
Shortest-Job-First (SJF) Scheduling

- 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 length of the next CPU request

Example of SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart



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

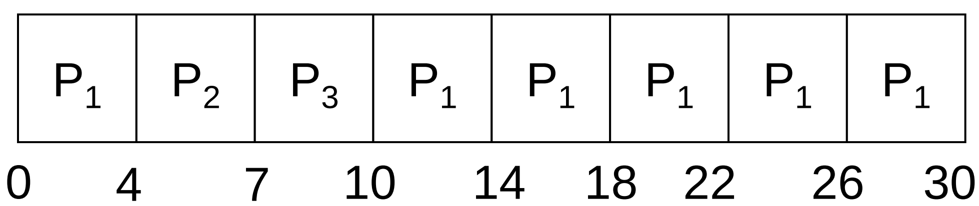
Round Robin (RR)

- 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 $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

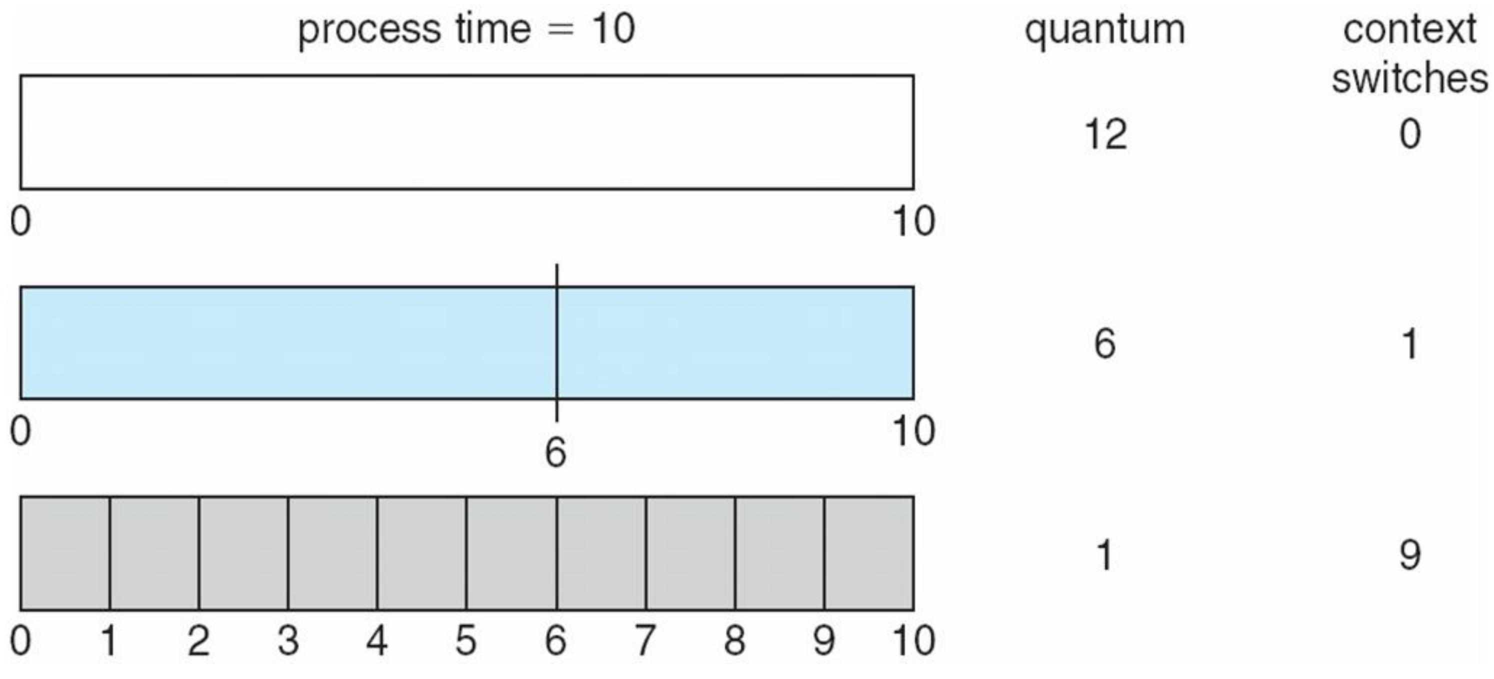
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

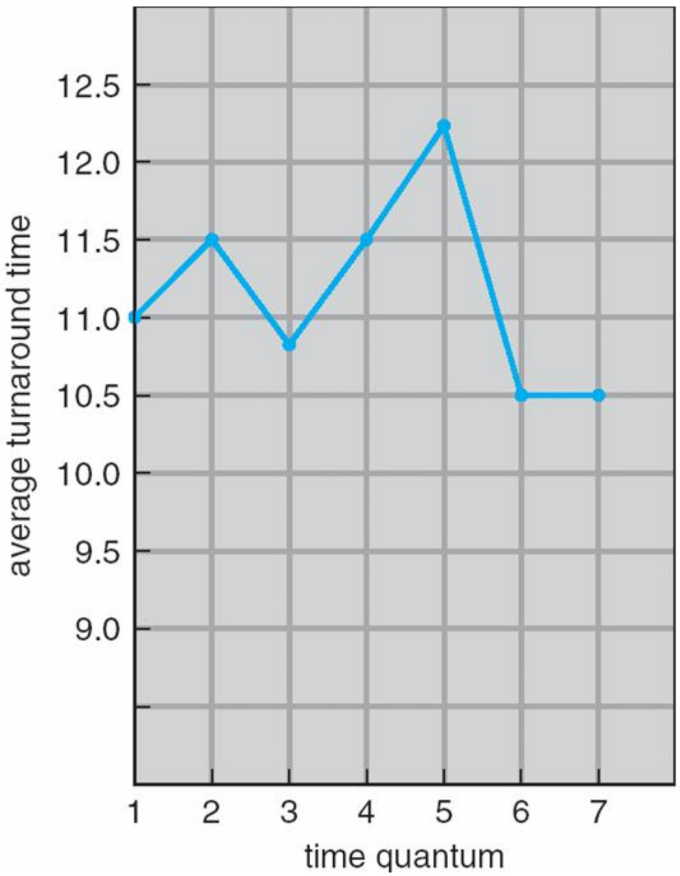


- Typically, higher average turnaround than SJF, but better *response*

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

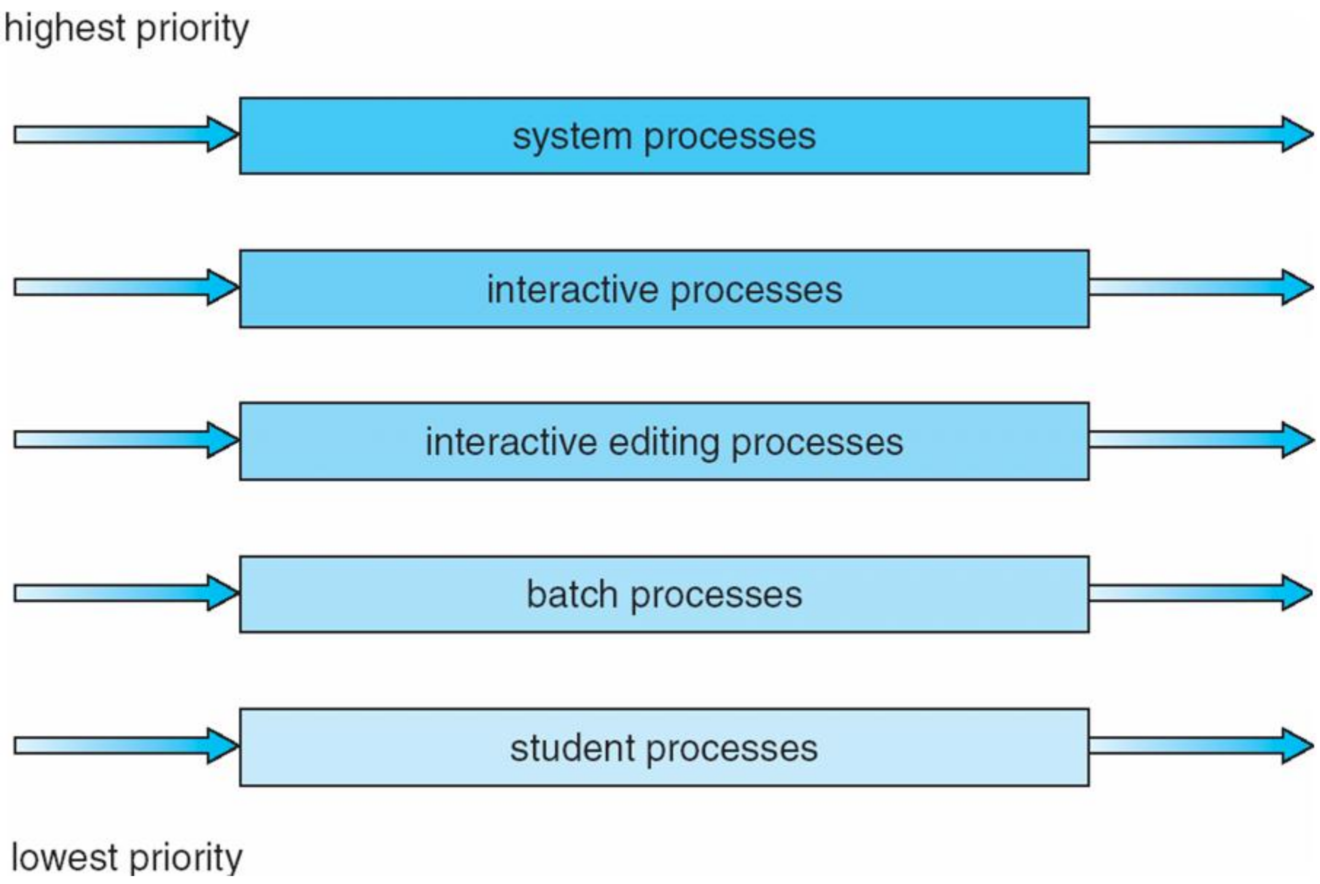


process	time
P_1	6
P_2	3
P_3	1
P_4	7

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; (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;
    - 80% to foreground in RR
    - 20% to background in FCFS

Multilevel Queue Scheduling

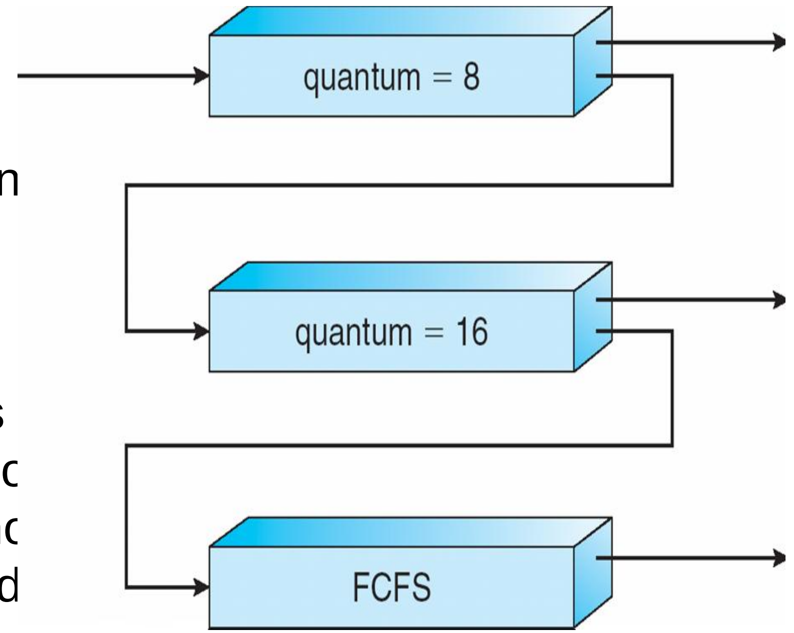


Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- 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

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 millisecon
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, jc receives 8 milliseconds. If it does nc finish in 8 milliseconds, job is moved 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.



Thread Scheduling

- Distinction between user-level and kernel-level 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

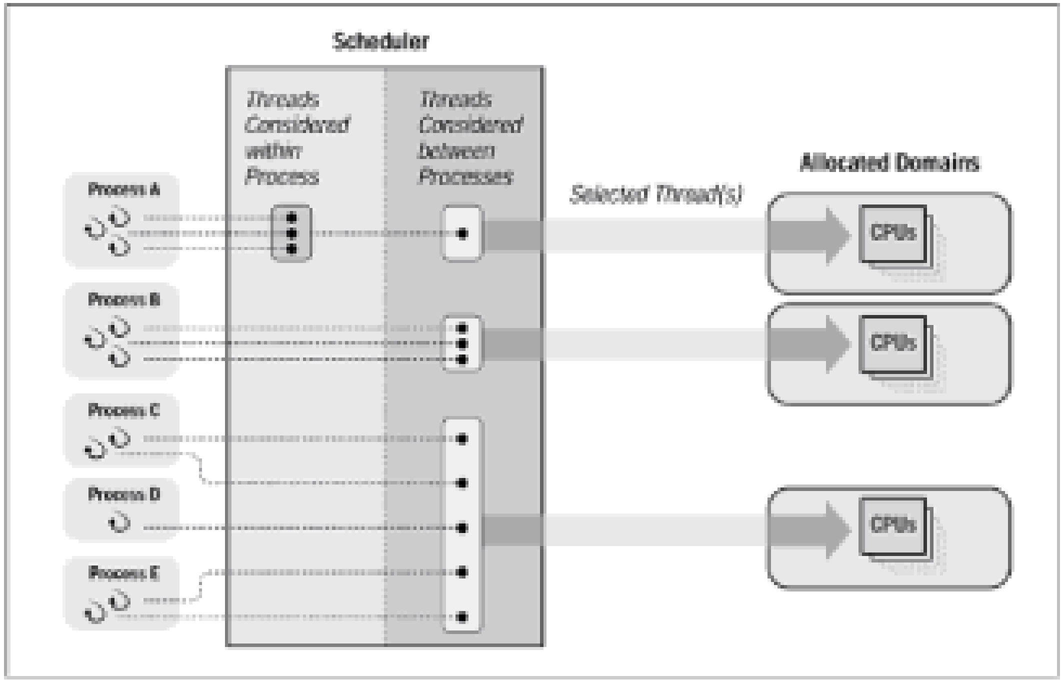
```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
     int i;
    pthread t tid[NUM THREADS];
    pthread attr t attr;
    /* get the default attributes */
    pthread attr init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    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,runner,NULL);
```

Pthread Scheduling API

```
        /* 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);
}
```

- Many threaded programs have no reason to interfere with the default behavior of the system's scheduler. Nevertheless, the Pthreads standard defines a thread-scheduling interface that allows programs with real-time tasks to get involved in the process.
- Scheduling priority
  - A thread's scheduling priority, in relation to that of other threads, determines which thread gets preferential access to the available CPUs at any given time.
- Scheduling policy
  - A thread's scheduling policy is a way of expressing how threads of the same priority run and share the available CPUs.

- Scheduling scope determines how many threads≈and which threads≈a given thread must compete against when it's time for the scheduler to select one of them to run on a free CPU.
- When scheduling occurs in *process scope*, threads are scheduled against only other threads in the same program. When scheduling occurs in *system scope*, threads are scheduled again stall other active threads systemwide.



- SCHED_FIFO: This policy (first-in first-out) lets a thread run until it either exits or blocks. As soon as it becomes unblocked, a blocked thread that has given up its processing slot is placed at the end of its priority queue.
- SCHED_RR: This policy (round robin) allows a thread to run for only a fixed amount of time before it must yield its processing slot to another thread of the same priority. This fixed amount of time is usually referred to as a *quantum*. When a thread is interrupted, it is placed at the end of its priority queue.
- The Pthreads standard defines an additional policy, SCHED_OTHER, and leaves its behavior up to the implementors. On most systems, selecting SCHED_OTHER will give a thread a policy that uses some sort of time sharing with priority adjustment. By default, all threads start life with the SCHED_OTHER policy.

- A real-time application designer would typically first make a broad division between those tasks that must be completed in a finite amount of time and those that are less time critical. Those threads with real-time tasks would be given a SCHED_FIFO policy and high priority. The remaining threads would be given a SCHED_RR policy and a lower priority. The scheduling priority of all of these threads would be set to be higher than those of any other threads on the system. Ideally the host would be capable of system-scope scheduling.

End of Chapter 6