Example of Shortest-remaining-time-first i.e. Preemptive SJF

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

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Preemptive SJF Gantt Chart**

- Average waiting time = $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5$ msec
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)
  - Can be preemptive
  - Can be 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

- There is a rumor that when the IBM 7094 at MIT was shut down in 1973 that a low-priority process from 1967 had not yet been run.
## Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time (ms)</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$P_5$</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Priority scheduling Gantt Chart assuming all arrive at time 0

![Gantt Chart](image)

- Average waiting time = \( \frac{0+1+6+16+18}{5} = 8.2 \text{ msec} \)
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum \( q \)), 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.

- Timer interrupts every quantum to schedule the next process.

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

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>24</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>3</td>
</tr>
</tbody>
</table>

- The Gantt chart assuming all processes arrive at time 0 is:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| $P_1$| $P_2$| $P_3$| $P_1$| $P_1$| $P_1$| $P_1$| $P_1$|     |
| 0    | 4    | 7    | 10   | 14   | 18   | 22   | 26   | 30   |
```

- Typically, higher average turnaround than SJF, but better response
- $q$ should be large compared to context switch time
- $q$ usually 10ms to 100ms, context switch < 10 $\mu$sec
Time Quantum and Context Switch Time

process time = 10

<table>
<thead>
<tr>
<th>Quantum</th>
<th>Context Switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Turnaround Time Varies With The Time Quantum

Given a time quantum of 1, where does 11.0 come from?

80% of CPU bursts should be shorter than q
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- 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; i.e., 80% to foreground in RR
  - 20% to background in FCFS
Multilevel Queue Scheduling

highest priority

- system processes

- interactive processes

- interactive editing processes

- batch processes

- student processes

lowest priority
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 milliseconds
  - $Q_2$ – FCFS

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

```c
#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);
}