Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Thread Scheduling
- Operating Systems Examples
- Algorithm Evaluation
The idea of multiprogramming:
- Keep several processes in memory. Every time one process has to wait, another process takes over the use of the CPU

CPU-I/O burst cycle: Process execution consists of a cycle of CPU execution and I/O wait (i.e., CPU burst and I/O burst).
- Generally, there is a large number of short CPU bursts, and a small number of long CPU bursts.
- An I/O-bound program would typically have many very short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.
Alternating Sequence of CPU And I/O Bursts

- load store
- add store
- read from file

**wait for I/O**

- store increment
- index
- write to file

**wait for I/O**

- load store
- add store
- read from file

**wait for I/O**
Histogram of CPU-burst Times

![Histogram of CPU-burst Times](image-url)
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
Preemptive Issues

- Inconsistent cases may occur: preemptive scheduling incurs a cost associated with access to shared data.

- Affect the design of OS kernel: What happens if the process is preempted in the middle of critical changes (for instance, I/O queues) and the kernel (or the device driver) needs to read or modify the same structure?
  - Unix solution: waiting either for a system call to complete or for an I/O block to take place before doing a context switch.
  - However, weak in supporting real-time computing and multiprocessing.
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
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- 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)
Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
Scheduling Algorithms

- First-come, first-served (FCFS) scheduling
- Shortest-job-first (SJF) scheduling
- Priority scheduling
- Round-robin scheduling
- Multilevel queue scheduling
- Multilevel feedback queue scheduling
First-Come, First-Served (FCFS) Scheduling

<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>

Suppose that the processes arrive in the order: $P_1, P_2, P_3$

The Gantt Chart for the schedule is:

```
0  24  27  30
P_1   P_2   P_3
```

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
Suppose that the processes arrive in the order $P_2, P_3, P_1$

- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th></th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 (SJR) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)

- SJF is optimal – gives minimum average waiting time for a given set of processes
## Example of Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

![Timeline Diagram](image)

- Average waiting time = \( \frac{0 + 6 + 3 + 7}{4} = 4 \)
Example of Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

- Average waiting time = \( \frac{9 + 1 + 0 + 2}{4} = 3 \)
Determining Length of Next CPU Burst

- Frequently used in long-term scheduling
  - A user is asked to estimate the job length. A lower value means faster response. Too low a value will cause timeout.

- **Approximate SJF**: the next burst can be predicted as an **exponential average** of the measured length of previous CPU bursts

\[
\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \quad \text{history}
\]

\[
= \alpha t_n + (1 - \alpha)\alpha t_{n-1} + (1 - \alpha)^2 \alpha t_{n-2} + \ldots
\]

Commonly, \(\alpha = 1/2\)

\[
= \frac{1}{2} t_n + \frac{1}{2}^2 t_{n-1} + \frac{1}{2}^3 t_{n-2} + \ldots
\]
Prediction of the Length of the Next CPU Burst

- CPU burst ($t_i$): 6, 4, 6, 4, 13, 13, 13, ...
- "guess" ($\tau_i$): 10, 8, 6, 6, 5, 9, 11, 12, ...

Graph showing time on the x-axis and $\tau_i$ on the y-axis.
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count

- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n = t_n$
  - Only the actual last CPU burst counts

Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor
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
An Example

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

AWT = (6+0+16+18+1)/5=8.2
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 = 20

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>53</td>
</tr>
<tr>
<td>$P_2$</td>
<td>17</td>
</tr>
<tr>
<td>$P_3$</td>
<td>68</td>
</tr>
<tr>
<td>$P_4$</td>
<td>24</td>
</tr>
</tbody>
</table>

- The Gantt chart is:

```
  0  20  37  57  77  97  117  121  134  154  162
P_1 | P_2 | P_3 | P_4 | P_1 | P_3 | P_4 | P_1 | P_3 | P_3 |
```

- Typically, higher average turnaround than SJF, but better *response*
Time Quantum and Context Switch Time

process time = 10

<table>
<thead>
<tr>
<th></th>
<th>quantum</th>
<th>context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-6</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6-10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1-9</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
Turnaround Time Varies With The Time Quantum

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
</tbody>
</table>

$p_1p_2p_3p_4p_1p_2p_4p_1p_2p_4p_1p_4p_1p_4p_4$ 

1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7
Multilevel Queue

- Ready queue is partitioned into separate queues: foreground (interactive) and 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; 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 $Q_2$. 
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS
Multiple-Processor Scheduling

- Only **homogeneous** systems are discussed here

- **Symmetric multiprocessing**
  - Each processor is self-scheduling
  - All processes may be in a common ready queue or each processor have its own private queue

- Asymmetric multiprocessing: all system activities are handled by a processor, the others only execute user code (allocated by the master), which is far simple than **symmetric multiprocessing**

- **Processor affinity**: a process has an affinity for the processor on which it is currently running
  - Keep a process running on the same processor
SMT: Symmetric Multithreading

- SMP: Allow several threads to run concurrently by providing multiple physical processors (PPs).
- SMT: providing multiple logical processors (LPs) on the same PP.
- Each LP has its own architecture state, including general-purpose and machine-state registers.
  - LP is responsible for its own interrupt handling
- SMT is a feature provided by H/W (state, interrupt handling), not S/W.
  - Certain performance gain are possible if OS is aware that.
  - e.g., Consider a system with two PPs, both are idle. The scheduler should first try scheduling separate threads on each PP rather than on separate LPs on the same PP.
A Typical SMT Architecture

logical CPU  logical CPU

physical CPU

system bus

logical CPU  logical CPU

physical CPU
Real-Time Scheduling

- **Hard real-time** systems – required to complete a critical task within a guaranteed amount of time

- **Soft real-time** computing – requires that critical processes receive priority over less fortunate ones
Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP

- Global Scheduling – How the kernel decides which kernel thread to run next
Thread Scheduling

- User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads are ultimately mapped to an associated kernel-level thread, or LWP.

  - **Process local scheduling**: Thread scheduling is done local to the application. The threads library schedules user-level threads to run on an available LWP.

  - **System global scheduling**: The kernel decides which kernel thread to schedule.
#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);
}

5.38 Silberschatz, Galvin and Gagne ©2005
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling
Solaris 2 Scheduling

- **Global Priority**: Highest to lowest
- **Scheduling Order**: First to last
- **Class-Specific Priorities**: Real Time, System, Interactive & Time Sharing
- **Scheduler Classes**: Kernel threads of real-time LWPs, Kernel service threads, Kernel threads of interactive & time-sharing LWPs
Solaris Scheduling

- Four classes of scheduling: real-time -> system -> interactive -> time sharing.
- A process starts with one LWP and is able to create new LWPs as needed. Each LWP inherits the scheduling class and priority of the parent process. Default: time sharing (multilevel feedback queue)
- Inverse relationship between priorities and time slices: the high the priority, the smaller the time slice.
- Interactive processes typical have a higher priority; CPU-bound processes have a lower priority.
Solaris Scheduling

- Uses the system class to run kernel processes, such as the scheduler and paging daemon. The system class is reserved for kernel use only. User process running in kernel mode are not in the system class.

- Threads in the real-time class are given the highest priority to run among all classes.

- There is a set of priorities within each class. However, the scheduler converts the class-specific priorities into global priorities. (round-robin queue)
## Solaris Dispatch Table (for interactive and time-sharing threads)

<table>
<thead>
<tr>
<th>priority</th>
<th>time quantum</th>
<th>time quantum expired</th>
<th>return from sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>15</td>
<td>160</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>10</td>
<td>52</td>
</tr>
<tr>
<td>25</td>
<td>120</td>
<td>15</td>
<td>52</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
<td>20</td>
<td>53</td>
</tr>
<tr>
<td>35</td>
<td>80</td>
<td>25</td>
<td>54</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>30</td>
<td>55</td>
</tr>
<tr>
<td>45</td>
<td>40</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>40</td>
<td>58</td>
</tr>
<tr>
<td>55</td>
<td>40</td>
<td>45</td>
<td>58</td>
</tr>
<tr>
<td>59</td>
<td>20</td>
<td>49</td>
<td>59</td>
</tr>
</tbody>
</table>
Dispatch Table

- Priority: a higher number indicates a higher priority
- Time quantum: the lower priority has the higher time quantum
- Time quantum expired: the new priority of a thread that has used its entire time quantum without blocking. Such threads are considered CPU-intensive. Lower priority of these threads.
- Return from sleep: the priority of a thread that is returning from sleeping (such as waiting for I/O). Its priority is boosted to between 50 and 59.
## Windows XP Priorities

<table>
<thead>
<tr>
<th>Priority</th>
<th>Real-time</th>
<th>High</th>
<th>Above Normal</th>
<th>Normal</th>
<th>Below Normal</th>
<th>Idle Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-critical</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>above normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>below normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Windows XP Scheduling

- Using a priority-based, preemptive scheduling algorithm
- There are 32-level priority which are divided into two classes
  - The variable class contains priorities from 1 to 15
  - The real-time class contains threads from 16 to 31
- Within each of the priority classes is a relative priority
- The priority of each thread is based on the priority class it belongs to and its relative priority within that class.
Linux Scheduling – preemptive & priority

- Version 2.5: support SMP, Load balancing & Processor affinity
- Time-sharing (100-140) and Real-time (0-99)
- Higher priority with longer time quanta
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first
## The Relationship Between Priorities and Time-slice length

<table>
<thead>
<tr>
<th>numeric priority</th>
<th>relative priority</th>
<th>time quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>highest</td>
<td>200 ms</td>
</tr>
<tr>
<td>•</td>
<td></td>
<td>real-time tasks</td>
</tr>
<tr>
<td>•</td>
<td></td>
<td>other tasks</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The diagram illustrates the relationship between numeric priorities and time-slice lengths. Higher priorities, indicated by lower numbers, receive shorter time slices. For example, tasks with a priority of 0 receive the shortest time slice of 200 ms, while those with a priority of 140 receive the longest time slice of 10 ms.
List of Tasks Indexed According to Priorities

<table>
<thead>
<tr>
<th>active array</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td>priority</td>
<td>task lists</td>
</tr>
<tr>
<td>[0]</td>
<td><img src="image1" alt="Task List" /></td>
</tr>
<tr>
<td>[1]</td>
<td><img src="image3" alt="Task List" /></td>
</tr>
<tr>
<td></td>
<td><img src="image5" alt="Task List" /></td>
</tr>
<tr>
<td></td>
<td><img src="image7" alt="Task List" /></td>
</tr>
<tr>
<td>[140]</td>
<td><img src="image9" alt="Task List" /></td>
</tr>
</tbody>
</table>
Algorithm Evaluation

- Criteria to select a CPU scheduling algorithm may include several measures, such as:
  - Maximize CPU utilization under the constraint that the maximum response time is 1 second
  - Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time

- Evaluation methods?
  - deterministic modeling
  - queuing models
  - simulations
  - implementation
Deterministic Modeling

- Analytic evaluation
  Input: a given algorithm and a system workload to
  Output: performance of the algorithm for that workload

- Deterministic modeling
  - Taking a particular predetermined workload and defining the performance of each algorithm for that workload.
Deterministic Modeling

**Process** | **Burst Time**
---|---
P1 | 10
P2 | 29
P3 | 3
P4 | 7
P5 | 12

- **FCFS**

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
</table>
P1 | 39 |
P2 | 42 |
P3 | 61 |
P4 | 49 |
P5 | 12 |

**AWT** = 28 ms

- **SJF**

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
</table>
P3 | 3 |
P4 | 32 |
P1 | 61 |
P5 | 20 |

**AWT** = 13 ms

- **RR (q = 10)**

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
</table>
P3 | 10 |
P4 | 23 |
P1 | 30 |
P5 | 40 |
P2 | 50 |
P5 | 52 |
P2 | 61 |

**AWT** = 23 ms
Deterministic Modeling

- A simple and fast method. It gives the exact numbers, allows the algorithms to be compared.

- It requires exact numbers of input, and its answers apply to only those cases. In general, deterministic modeling is too specific, and requires too much exact knowledge, to be useful.

Usage

- Describing algorithm and providing examples
- A set of programs that may run over and over again
- Indicating the trends that can then be proved
Queuing Models

- Queuing network analysis
  - Using
    - the distribution of service times (CPU and I/O bursts)
    - the distribution of process arrival times
  - The computer system is described as a network of servers. Each server has a queue of waiting processes.
- Determining
  - utilization, average queue length, average waiting time, and so on
Queuing Models

- Little's formula (for a stable system):
  \[ n = \lambda \times W \]

  - \( n \): average queue length
  - \( \lambda \): average arrival rate
  - \( W \): average waiting time

  Example:
  - 14 persons in queue
  - 7 arrives/second
  - 2 seconds waiting

  Result:
  \[ 14 = 7 \times 2 \]

Queuing analysis can be useful in comparing scheduling algorithms, but it also has limitations.

- Queuing model is only an approximation of a real system. Thus, the result is questionable.
  - The arrival and service distributions are often defined in unrealistic, but mathematically tractable, ways.
  - Besides, independent assumptions may not be true.
Simulations

- Simulations involve programming a model of the system. Software data structures represent the major components of the system.
- Simulations get a more accurate evaluation of scheduling algorithms.
  - expensive (several hours of computer time).
  - large storage
  - coding a simulator can be a major task
- Generating data to drive the simulator
  - a random number generator.
  - trace tapes: created by monitoring the real system, recording the sequence of actual events.
Evaluation of CPU Schedulers by Simulation
Implementation

- Put data into a real system and see how it works.
- The only accurate way
  - cost is too high
  - environment will change (All methods have this problem!)
  - e.g., To avoid moving to a lower priority queue, a user may output a meaningless character on the screen regularly to keep itself in the interactive queue.
Home Works

- 2, 3, 5, 7, 9
- Due: Nov. 12
- First exam.: Nov. 12 (19:00 ~ 20:20)
End of Chapter 5