Chapter 5: Process Scheduling

By Worawut Srisukkham  Updated By Dr. Varin Chouvatut
Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating System Examples
- Algorithm Evaluation
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
Basic Concepts

- Maximum CPU utilization is obtained with multiprogramming
- CPU-I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these 2 states.
- **CPU-burst** distribution
Histogram of CPU-burst Times
Alternating Sequence of CPU and I/O Bursts

- load store
- add store
- read from file

- store increment index
- write to file

- wait for I/O

- load store
- add store
- read from file

- wait for I/O
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 state
  4. Terminates
- Scheduling schemes under circumstances 1 and 4 are nonpreemptive.
- All other schemes are preemptive.

**nonpreemptive:** ไม่สามารถแทรกการทำงานกลางคันขณะที่ CPU กำลังประมวลผลโปรเซส
**preemptive:** แทรกการทำงานกลางคันขณะที่ CPU กำลังประมวลผลโปรเซส
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler (or the CPU scheduler); this function involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program

- **Dispatch latency** – the time it takes for the dispatcher to stop one process and start another running

Dispatcher: ตัวส่งข่าวสารไปยัง state อื่น ตัวส่งค่อย
Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – the number of processes that are completed 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)
Scheduling Algorithm: Optimization Criteria

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

There are many different CPU-scheduling algorithms:
1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Priority Scheduling
4. Round-Robin Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling
First-Come, First-Served (FCFS) Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time (ms)</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:

|| P_1 | P_2 | P_3 |
|-----|-----|-----|
| 0   |     |     |
| 24  |     |     |
| 27  |     |     |
| 30  |     |     |

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Turnaround time: $P_1 = 24$; $P_2 = 27$; $P_3 = 30$
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: $(6 + 0 + 3)/3 = 3$
- Turnaround time: $P_1 = 30; P_2 = 3; P_3 = 6$
- Much better than previous case
- A *Convoy effect* – short processes stand behind a 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 first**

- 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
  - The difficulty is knowing the length of the next CPU request
### Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- **SJF scheduling chart**

<table>
<thead>
<tr>
<th></th>
<th>$P_4$</th>
<th>$P_1$</th>
<th>$P_3$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>9</td>
<td>16</td>
<td>24</td>
</tr>
</tbody>
</table>

- **Average waiting time** = \(\frac{3 + 16 + 9 + 0}{4} = 7\)

- **Turnaround Time**:
  - $P_1 = 9$
  - $P_2 = 24$
  - $P_3 = 16$
  - $P_4 = 3$
Example of nonpreemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>2.0</td>
<td>8</td>
</tr>
<tr>
<td>P₃</td>
<td>4.0</td>
<td>7</td>
</tr>
<tr>
<td>P₄</td>
<td>5.0</td>
<td>3</td>
</tr>
</tbody>
</table>

- **SJF scheduling chart**: แบบ nonpreemptive ไม่สามารถแทรกการทำงานกลางคันได้

```
<table>
<thead>
<tr>
<th>P₁</th>
<th>P₄</th>
<th>P₃</th>
<th>P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>9</td>
<td>16</td>
</tr>
</tbody>
</table>
```

- **Average waiting time** = \((0 + 14 + 5 + 1) / 4 = 20/4 = 5\) ms
# 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>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2.0</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4.0</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1.0</td>
<td>3</td>
</tr>
</tbody>
</table>

**SJF scheduling chart**: แบบ preemptive แทรกการทำงานกลางคันได้

![SJF scheduling chart diagram](chart.png)

- **Average waiting time** = \(\frac{3 + 14 + 5 + 0}{4} = \frac{22}{4} = 5.5\) ms

* ติค Arrival time ด้วย*
Determining Length of Next CPU Burst

เนื่องจากว่า SJF เหมาะกับการจัด Schedule แบบ Long-Term Scheduling จะไม่สามารถนำมาใช้กับ Short-Term Scheduling เพราะไม่สามารถที่จะรู้ช่วงเวลาถัดไปที่ CPU Burst จึงเกิดวิธีการต่อไปนี้ขึ้น

- Can only estimate the length
- 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. \( \tau_{n+1} = \) predicted value for the next CPU burst
3. \( \alpha, \ 0 \leq \alpha \leq 1 \)
4. Define \( \tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n \)

\( \alpha \) is constant or as an overall system average
Prediction of the Length of the Next CPU Burst

\[ \alpha = \frac{1}{2} \quad \text{and} \quad \tau_0 = 10 \]

<table>
<thead>
<tr>
<th>CPU burst ( t_i )</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ( \tau_i )</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  $$
  \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots
  + (1 - \alpha) t_{n-j} + \ldots
  + (1 - \alpha)^{n+1} \tau_0
  $$
- 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 be executed
- Solution ≡ Aging – as time progresses, increase the priority of the process
Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$P_5$</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

- **priority scheduling chart**

<table>
<thead>
<tr>
<th>P2</th>
<th>P4</th>
<th>P1</th>
<th>P3</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>11</td>
<td>17</td>
<td>24</td>
</tr>
</tbody>
</table>

- Average waiting time = \((11 + 0 + 17 + 8 + 24)/5 = 12\) ms
- Turnaround Time: $P_1 = 17; P_2 = 8; P_3 = 24; P_4 = 11; P_5 = 33$
Round-Robin (RR) Scheduling

- 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 \( \frac{1}{n} \) of the CPU time in chunks of at most *q* time units. No processes wait longer than \( (n - 1) \times q \) time units.

- Performance
  - *q* large \( \Rightarrow \) FCFS
  - *q* small \( \Rightarrow \) *q* must be large with respect to the context-switch time, otherwise overhead is too high.

*time quantum*: ส่วนแบ่งเวลา
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 is:

```
0  4  7  10  14  18  22  26  30
|1|2|3|P_1|P_2|P_3|P_1|P_1|P_1|
```

- average waiting time: \( \frac{6 + 4 + 7}{3} = 5.67 \) ms

- Turnaround time: \( P_1 = 30 \); \( P_2 = 7 \); \( P_3 = 10 \);

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

Showing how a smaller time quantum increases context switches
Turnaround time also depends on the size of the quantum time
Multilevel Queue Scheduling

- 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
  - Commonly implemented as fixed-priority preemptive 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
  - and 20% to background in FCFS
An example of a multilevel queue scheduling algorithm with 5 queues, listed in order of priority.

- Highest priority: System processes
- Interactive processes
- Interactive editing processes
- Batch processes
- Student processes

(lowest priority)
Multilevel Feedback Queue Scheduling

- A process can move between various queues; aging can be implemented this way to prevent starvation.
- Multilevel-feedback-queue scheduler is generally defined by the following parameters:
  - the number of queues
  - the scheduling algorithm 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: เพิ่มศักดิ์ขึ้น
demote: ลดศักดิ์ลง
Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$ – RR with time quantum of 8 milliseconds
  - $Q_1$ – RR with time quantum of 16 milliseconds
  - $Q_2$ – FCFS

- Scheduling
  - A new job enters queue $Q_0$. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to the tail of queue $Q_1$.
  - Only when queue $Q_0$ is empty will the scheduler execute processes in queue $Q_1$. At $Q_1$ job receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue $Q_2$.
  - Processes in queue $Q_2$ are run on an FCFS basis but are run only when queues $Q_0$ and $Q_1$ are empty.
Multilevel Feedback Queues

Queue ถ้าไม่สามารถเริ่มทำงานได้ หาก Queue ก่อนหน้าทำงานไม่เสร็จ หรือยังไม่empty หรือยังไม่หมดช่วงเวลาที่กำหนดให้ (time quantum)
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 same process

- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
- System using one-to-one models such as Windows XP, Solaris, and Linux schedule thread using only SCS
In thread creation with Pthreads, the POSIX Pthread API allows specifying either PCS or SCS during thread creation.

Pthreads identifies the following contention scope values:

- `PTHREAD_SCOPE_PROCESS` schedules threads using PCS scheduling
- `PTHREAD_SCOPE_SYSTEM` schedules threads using SCS scheduling.

**PCS:** process-contention scope

**SCS:** system-contention scope
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 (PCS) or SYSTEM (SCS)*/
    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);
} /* end main */

/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread_exit(0);
}
Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **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** – a process has an affinity for the processor on which it is currently running
  - **soft affinity**: process may migrate between processors
  - **hard affinity**: process must not migrate to other processors

---

Homogeneous: แบบเดียวกัน เช่น cpu เป็น Intel เหมือนกัน
Heterogeneous: หลายแบบ เช่น cpu เป็น Intel, AMD, ultra spark, power Mac
Alleviating: แบ่งเบาภาระ
migrate: ย้ายการทำงาน
affinity: เกี่ยวพันกัน
NUMA and CPU Scheduling

NUMA : Non-Uniform Memory Access

สถาปัตยกรรมที่มีการใช้ NUMA จะทำให้:

A CPU has faster access to some parts of main memory than to other parts.

![Diagram of NUMA and CPU Scheduling](image)
Recent trend is to place multiple processor cores on the same physical chip.

SMP systems that use Multicore processors are Faster and consume Less power than systems in which each processor has its own physical chip.

Multiple threads per core are also growing.

- Takes advantage of memory stall to make progress on another thread while memory retrieval happens.

SMP : Symmetric Multiprocessing
memory stall cycle: ช่วงเวลาที่ CPU ต้องรอการนำข้อมูลที่ไม่ได้อยู่ในหน่วยความจำให้ถูก load มาไว้ในหน่วยความจำ เช่น a cache miss (ข้อมูลที่ต้องการเข้าถึง ไม่ได้อยู่ใน cache memory)
Multithreaded processor cores in which 2 (or more) hardware threads are assigned to each core. Thus, if one thread stalls while waiting for memory, the core can switch to another thread.
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux scheduling
## Solaris Dispatch Table

<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>
Solaris Scheduling

- Interrupt threads
- Realtime (RT) threads
- System (SYS) threads
- Fair share (FSS) threads
- Fixed priority (FX) threads
- Timeshare (TS) threads
- Interactive (IA) threads
## Windows XP Priorities

<table>
<thead>
<tr>
<th></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>
Linux Scheduling

- Constant order $O(1)$ scheduling time
- Two priority ranges: time-sharing (or multitasking) and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- See example picture on next slide
### 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></td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lowest</td>
<td>10 ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>real-time tasks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>other tasks</td>
<td></td>
</tr>
</tbody>
</table>
### List of Tasks Indexed According to Priorities

<table>
<thead>
<tr>
<th>active array</th>
<th>expired array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>priority</strong></td>
<td><strong>task lists</strong></td>
</tr>
<tr>
<td>[0]</td>
<td>[1]</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>·</td>
<td>·</td>
</tr>
<tr>
<td>[140]</td>
<td>·</td>
</tr>
</tbody>
</table>

**active array:** เก็บ task ที่ทำงานอยู่
**expired array:** เก็บ task ที่หมดเวลา
Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.
  
  Ex. Define all processes running in FCFS, SJF, RR and then find out the result of minimum waiting time.

- Queueing models – what can be determined is the distribution of CPU and I/O bursts. Knowing arrival rate and service rates, we can compute utilization, average queue length, average wait time, and so on.

- Implementation – the only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the OS, and see how it works.
Evaluation of CPU schedulers by Simulation

- Actual process execution
  - CPU 10
  - I/O 213
  - CPU 12
  - I/O 112
  - CPU 2
  - I/O 147
  - CPU 173

- Trace tape

- Simulation
  - FCFS
  - Performance statistics for FCFS

- Simulation
  - SJF
  - Performance statistics for SJF

- Simulation
  - RR (q = 14)
  - Performance statistics for RR (q = 14)
End of Chapter 5