CHAPTER 5: PROCESS SCHEDULING
Chapter 5: Process Scheduling

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

- Maximum CPU utilization obtained with multiprogramming

- CPU–I/O Burst Cycle — Process execution consists of a cycle of CPU execution and I/O wait

- CPU burst distribution
Alternating Sequence of CPU And I/O Bursts
Histogram of CPU-burst Times
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 non-preemptive

- 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

- 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
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P_1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>30</td>
</tr>
</tbody>
</table>

- 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:

  \[
  \begin{array}{c|c|c|c}
  \hline
  \text{P}_2 & \text{P}_3 & \text{P}_1 \\
  0 & 3 & 6 \\
  \hline
  \end{array}
  \]

- 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

- 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 [Shortest-Remaining-Time-First (SRTF)]
    - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.

- 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)**

Average waiting time $= (0 + 6 + 3 + 7)/4 = 4$
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 = $(9 + 1 + 0 + 2)/4 = 3$
Determining Length of Next CPU Burst

- 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: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$. 
Prediction of the Length of the Next CPU Burst

![Graph showing the prediction of CPU burst lengths]

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

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

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

- If we expand the formula, we get:
  \[
  \tau_{n+1} = \alpha \tau_n + (1 - \alpha)\alpha \tau_{n-1} + \ldots + (1 - \alpha)^{i+1} \alpha \tau_i + (1 - \alpha)^{n+1} \tau_0
  \]

- 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 $\equiv$ highest priority)
  - Preemptive
  - Nonpreemptive

- SJF is a priority scheduling where priority is the predicted next CPU burst time
Priority Scheduling (cont.)

- **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

<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:

```
0  4  7  10  14  18  22  26  30
P_1 P_2 P_3 P_1 P_1 P_1 P_1 P_1
```
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:

```
<table>
<thead>
<tr>
<th></th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th>P_4</th>
<th>P_1</th>
<th>P_3</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20</td>
<td>37</td>
<td>57</td>
<td>77</td>
<td>97</td>
<td>117</td>
<td>121</td>
<td>134</td>
<td>154</td>
<td>162</td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>process time = 10</th>
<th>quantum</th>
<th>context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
```

- Process time = 10
- Quantum: 12, 6, 1
- Context switches: 0, 1, 9
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>
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; 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 Queues

- 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
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

```c
#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);
```
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
Thread Scheduling

- **Local Scheduling**
  - How the threads library decides which thread to put onto an available LWP
  - **Process-contention-scope (PCS)**

- **Global Scheduling**
  - How the kernel decides which kernel thread to run next
  - **System-contention-scope (SCS)**
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.
#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);

    /* 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);
}
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available

- Homogeneous processors within a multiprocessor

- Load sharing

- 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
Multiple-Processor Scheduling

- Processor Affinity
  - Avoid migration of processes from one processor to another.
  - Soft affinity (mostly, but not guaranteed)
  - Hard affinity (no migration)
Non-uniform memory access (NUMA) also affect affinity of processor and memory.
Multi-core and multi-threaded

- Recent trend to place multiple processor cores on same physical chip
  - Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens
Real-Time Scheduling

- **Hard real-time systems**
  - required to complete a critical task within a guaranteed amount of time
  - Special-purpose software and hardware

- **Soft real-time computing**
  - requires that critical processes receive priority over less fortunate ones
  - Real-time processes
    - The priority must not degrade over time.
    - The dispatch latency must be small.
Real-Time Scheduling (cont.)

- Most versions of UNIX
  - wait system call to complete.
  - Dispatch latency can be long.

- To allow system calls to be preemptible.
  - Insert preemption points
  - Make the entire kernel preemptible (Solaris 2)
    - Use synchronization mechanisms to protect kernel data structure.
Real-Time Scheduling (cont.)

- What happen if the high-priority process needs to read or modify kernel data being access by lower-priority process?

- Problem: priority inversion

- Priority-inheritance protocol
  - The low-priority processes inherit the high priority until the resources is released.
Operating System Examples

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

- Global priority: highest at the top, lowest at the bottom.
- Scheduling order: first at the top, last at the bottom.
- 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
- Run queue:
  - Interrupt threads
  - Realtime (RT) threads
  - System (SYS) threads
  - Fair share (FSS) threads
  - Fixed priority (FX) threads
  - Timeshare (TS) threads
  - Interactive (IA) threads

Diagram illustrates the scheduling process with priorities and classes.
Solaris Dispatch Table for time-sharing and interactive 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>

A higher priority value means higher priority.
Windows XP

- A soft real-time OS.
- Priorities are divided into two classes:
  - The variable class: 1~15
  - The real-time class: 16~31

- The base priority + the relative priority
  - Low the thread priority when it is interrupted (time).
  - Boost the priority when a thread is released from “wait”.
  - Give good response time to interactive threads.
  - The foreground process: time slice X 3.
## 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

- Two algorithms: time-sharing and real-time

- 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 runnable processes have credit = 0, recrating occurs
    - Based on factors including priority and history
    - E.g. Credits = credits/2 + priority

- 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>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>lowest</td>
<td>10 ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other tasks</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>priority</td>
<td>task lists</td>
</tr>
<tr>
<td>[0]</td>
<td>●</td>
</tr>
<tr>
<td>[1]</td>
<td>● ●</td>
</tr>
<tr>
<td></td>
<td>● ● ●</td>
</tr>
<tr>
<td>[140]</td>
<td>● ● ● ●</td>
</tr>
</tbody>
</table>
Algorithm Evaluation

- **Deterministic modeling**
  - takes a particular predetermined workload and defines the performance of each algorithm for that workload
  - Too specific

- **Queueing models**
  - The mathematics of complicated algorithms can be difficult to work with.
  - May not be accurate.
Algorithm Evaluation (cont.)

- **Simulation**
  - According to probability distributions.
  - Trace tapes: recording the sequence of actual events.
  - Can be expensive

- **Implementation**
  - Put the actual algorithm in the real system
  - The cost is high
  - The environment will change
Deterministic modeling

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>39</td>
<td>42</td>
<td>49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P₃</th>
<th>P₄</th>
<th>P₁</th>
<th>P₅</th>
<th>P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>10</td>
<td>20</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
<th>P₄</th>
<th>P₅</th>
<th>P₂</th>
<th>P₅</th>
<th>P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>20</td>
<td>23</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>52</td>
</tr>
</tbody>
</table>
5.15

actual process execution

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