**CS 575 Scheduling Algorithms Simulator**

**By Nik Yusof and Farjad Yousufi**

We investigate the performance of Job or Process Scheduling strategies by creating a simulator of an operating system that runs random jobs, we create certain strategies all utilizing various Memory Allocation strategies and a CPU Scheduling strategy on an operating system class.

**Theoretical Concepts**

Explain First Fit

Explain Best Fit

Explain Shortest Job

Explain Contiguous Allocation

Explain Pre-emption and Non-Pre-emptive Scheduling

Explain Multi-programming Concept

**Overall Design of Our Program**

Our program consists simply of 6 classes to set up the operating system, processes and the memory within it to simulate how the scheduling will work. We then create separate execution classes to run the various scheduling strategies starting from 1…to n where n is the number of different strategies implemented. These strategies are discussed above.

Operating System Class:

Initialized by the “strategy” we are running. It takes in the status of the processes and if the processes are ready to be output on the screen for monitoring by the user. Each strategy will take in the processes and the allocated memory and start scheduling based on the strategy which we will go into detail on two different parallel processor utilizing round robin strategy. We start with 20 processes.

Process Class:

A process consists of the following variables:

a. Process identification number  
b. Memory request (in MB)  
c. Execution time request  
d. Memory assigned (segment number)  
e. Execution time remaining (initially same as item c.)

f. Job status: Waiting, Ready, Running, & Finished

Our Data Structure:

We utilize a Queue data structure that consists of an array of Processes. We have about 10 processes that we utilize to give a good indication of how the scheduling strategies run. This Data Structure has a number of methods that provide information on the various processes based on the type of scheduling strategy that are being run. We set the instance variables of the process class in this class utilizing the Queue.

A process generator class:

This uses the custom Queue data structure to set the time requested, memory requested, id, and remaining time for a process.

Memory Modules Class:

This is the essential component of our main memory class and consists of information on which segment the memory is, the size of the modules in MB, whether the segment is currently in use as a Boolean, and the process number being run on that particular module.

Main Memory Class:

We create a main memory class that will have the memory scheduled on it. This is essentially an array of the memory modules that we created above. We have the number of the modules that we have as the size of the array within this class, it also includes the particular size of each memory module as they are all different sizes to show a good simulation of the First-Fit and Best-fit strategies.

a. Segment number  
b. Size in kilobytes (MB)  
c. Whether the segment is in use or not

d. Wasted space (MB)

Illustration of Memory Component

**Fixed-Size Memory Segments**

*Segment Number* *Size in MB*

0 64

1 96

2 48

3 32 4 128

5 96

6 48

**Algorithms Utilized:**

Case One-FCFS First Fit

Jobs are moved from the job queue to the ready queue in **First-Come First-Served** order and

using the **First-Fit** allocation policy. The jobs in the ready queue are executed on the two processors in a **round robin** fashion.

Case Two-FCFS Best Fit

Jobs are moved from the job queue to the ready queue in **First-Come First-Served** order and

using the **Best-Fit** allocation policy. The jobs in the ready queue are executed on the two processors in a **round robin** fashion.

Case Three-SJFS Best-Fit

Jobs are moved from the job queue to the ready queue in **Shortest-Job First-Served** order (shortest means smallest execution time) and using the **Best-Fit** allocation policy. The jobs in the ready queue are executed on the two processors in a **round robin** fashion.

Case Four-SJFS First-Fit

Jobs are moved from the queue to the ready queue in Shortest Job First served order (smallest execution time) and then we use the first-fit allocation policy. All processes use the round robin method to implement on our two processors.

The computer system that runs this OS has **two** identical processors in the main memory Class so that two ready processes can execute truly in parallel per time unit of which each case runs 30. The OS uses multiprogramming to multiplex the two processors amongst all the processes in memory. The physical memory space is divided into several segments of varying sizes and the allocation strategy uses **contiguous allocation** of memory space to the waiting processes. Each user program runs in a single segment. Any unused space of an allocated segment is considered wasted.

Memory space requirement and CPU time requirement of all the processes will be randomly generated within the memory module and memory classes and this uses Java.random(): this data essentially constitutes the “**Process Queue**” or “**Input Queue**”. The OS allocates memory to as many processes as possible based on the scheduling and allocation strategies. It then starts their execution. These processes remain in memory until completion (**no preemption**.)

When one or more jobs finish, the HOS brings in additional jobs from the job queue based on the same scheduling and allocation strategies.

The input for the simulation will be the memory request and execution time request of a sequence of jobs. To generate input, we use the random number generator Math.random() or Random class provided by Java. We use calls to the random number generator to input a job. Assume that memory requests (in MB) are randomly distributed in the range (24 - 100) and time requests are in the range (2 - 10). We generate 20 jobs, all assumed to have an arrival time of 0 (zero). For job identification, we use the number the jobs 0 to 19, in order, as they are generated as the job ID. We assume that the simulation will run for 30 time units.

Output of a Sample Run of the first Case:

Executing the OS with case 1.

=========================================================================

Time Unit Number: 1

=========================================================================

Time ID Segment Mem Request Time Remain Messages

---- -- ------- ----------- ----------- --------

7 0 0 51 7 Running

4 1 1 62 4 Running

4 2 4 84 4 Ready

9 3 5 60 9 Ready

8 4 -1 59 8 Waiting

9 5 -1 74 9 Waiting

5 6 -1 91 5 Waiting

3 7 2 40 3 Ready

9 8 3 28 9 Ready

2 9 -1 79 2 Waiting

2 10 -1 66 2 Waiting

8 11 -1 59 8 Waiting

2 12 -1 70 2 Waiting

3 13 -1 61 3 Waiting

2 14 -1 81 2 Waiting

2 15 -1 96 2 Waiting

3 16 -1 55 3 Waiting

8 17 -1 59 8 Waiting

2 18 -1 82 2 Waiting

2 19 6 40 2 Ready

=========================================================================

Waiting: 13

Total Memory Wasted: 147

=========================================================================

**Assumptions**

For the sake of simplicity, we assumed that all execution times are in multiples of one time slice unit. We will assume true parallel processing of jobs and thus, during each time slice, take one time unit off the time remaining for two of the active jobs. To be fair to all the jobs in the ready queue, you need to schedule the ready jobs in a **round robin** fashion.

**Conclusion Section**