CSCI 4210 — Operating Systems
CSCI 6140 — Computer Operating Systems
Project 3 (document version 1.1)
Round Robin Scheduling and Contiguous Memory Allocation

Overview

- This project is due by 11:59:59 PM on Monday, November 23, 2015. Projects are to be submitted electronically.
- This project will count as 13% of your final course grade.
- This project is to be completed either **individually** or in a **team of two**. Do not share your code with anyone else.
- You **must** use one of the following programming languages: C, C++, Java, or Python.
- Your program **must** successfully compile and run on Ubuntu v14.04.3 LTS or v15.04 (consider using `http://c9.io`).

Project Specifications

In this third project, you will extend the first two projects and complete our study of CPU scheduling algorithms, in particular, round robin (RR) scheduling. Further, you will incorporate a contiguous memory allocation scheme into your simulation.

As with the first two projects, processes may be waiting to use the CPU or blocked on I/O. In this current project, processes may also be suspended due to memory defragmentation. Further, processes enter and exit the system potentially with non-zero arrival times.

Also, each process will have specific memory requirements to be met by the various contiguous memory allocation schemes, i.e., the first-fit, next-fit, and best-fit algorithms.

The I/O subsystem will not be covered in this project.

Conceptual Design

A **process** is defined as a program in execution. Expanding the set of states from the previous projects, processes in this project are in one of the following five states: (a) waiting to be admitted into the system (i.e., memory allocation); (b) ready to use the CPU; (c) actively using the CPU; (d) blocked on (or performing) I/O; and (e) exiting the system (i.e., memory deallocation).

Processes in state (a) are admitted to the system in a first-come-first-served (FCFS) manner. If possible, a process is allocated a contiguous partition of memory (depending on the placement algorithm) and changes its state to state (b).

Processes in state (b) reside in the ready queue. The scheduler selects processes from this queue to hand off to the dispatcher, which then performs the necessary context switch to get each process running with the CPU and therefore into state (c).

From state (c), processes may move back to state (b) due to a preemption, move to state (d) to perform I/O, or move to state (e) to terminate.

For this project, the scheduling algorithms are shortest remaining time (SRT) and round robin (RR). SRT was already implemented in our previous projects. When you run your simulation, it must simulate both of these scheduling algorithms in conjunction with memory allocation and deallocation.

Shortest Remaining Time (SRT)

The SRT algorithm selects the process with the lowest CPU burst time from among all processes in the queue. The queue is therefore often implemented as a priority queue ordered by CPU burst time. Note that this is essentially a priority scheduling algorithm in which the priority measure is based on CPU burst time.

The SRT algorithm is a preemptive algorithm, i.e., when process `A` arrives and is to be added to the queue, its CPU burst time is compared with the remaining time of currently running process `B`. If the burst time of `A` is less than that of `B`, then a preemption occurs—i.e., process `B` is forced to relinquish its time with the CPU (and is returned to the ready queue) such that process `A` uses the CPU immediately (after a context switch). In such a case, process `A` is not added to the queue.

Round Robin (RR)

The RR algorithm is essentially the FCFS algorithm with a predefined time slice `t_slice`. Each process is given `t_slice` amount of time to complete its CPU burst. If the time slice expires, the process is preempted and added to the end of the ready queue.

If a process completes its CPU burst before a time slice expiration, the next process on the ready queue is context-switched into the CPU.

For your simulation, if a preemption occurs and there are no processes on the ready queue, do **not** perform a context switch.

Contiguous Memory Management

As noted above, each process has specific memory requirements to be met by various contiguous memory allocation schemes. The algorithms to simulate are the first-fit, next-fit, and best-fit algorithms, each of which is described in more detail in the next section.

For main memory, use a data structure of your choosing to represent a memory that contains a configurable number of equally sized "memory units." Note that we will only use a dynamic partitioning scheme for this project, meaning that your data structure needs to maintain a list containing: (1) where processes are allocated; (2) how much contiguous memory they use; and (3) where and how much free memory is available (i.e., where the free partitions are).

As an example, memory may be represented as shown below (also see examples in the in-class notes). When you display your simulated memory, use the format below, showing 32 memory units per line. Also, as a default, simulate a memory consisting of 256 memory units (i.e., eight output lines).

```
Simulated Memory:
================================
AAAAAAAABBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBB.................
...................DDDDDDDDD....
................................
................................
........HHHHHHHHHHHHHHHHHHHHHHHH
HHH.............................
................................
================================
```

In the above example diagram, four processes are allocated into their four respective partitions. Further, there are three free partitions (i.e., between `B` and `D`, between `D` and `H`, and between `H` and the "bottom" of memory.

Placement Algorithms

When a process `P` wishes to enter the system, memory must first be allocated dynamically. More specifically, a dynamic partition must be created from an existing free partition.

For the **first-fit** algorithm, `P` is placed into the first free partition available in scanning from the "top" of memory.

For the **next-fit** algorithm, `P` is placed into the first free partition available in scanning from the end of the most recently placed process.

For the **best-fit** algorithm, `P` is placed into the smallest free partition available into which `P` fits. If a "tie" occurs, use the free partition closer to the top of memory.

For all three of these placement algorithms, the memory scan covers the entire memory. If necessary (i.e., if the scan hits the bottom of memory), the scan continues from the top of memory.

If no suitable free partition is available, then an out-of-memory error occurs, at which point defragmentation is required.

Defragmentation

If a process is unable to be placed into memory, defragmentation occurs. In such cases, processes are relocated as necessary, starting from the top of memory.

Once defragmentation is started, it will run through to completion, at which point, the process that triggered defragmentation will (hopefully) be admitted to the system successfully. If not, your simulation must deny the request and move to the next process.

While defragmentation is running, no other processes may be running (i.e., all processes are essentially placed in a suspended state) until all relocations are complete.

Note that the time to move **one unit** of memory is defined as `t_memmove`.

Given the memory shown previously, the results of defragmentation will be as follows (i.e., processes `D` and `H` move upwards in memory):

```
Simulated Memory:
================================
AAAAAAAABBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBDDDDDDDDDHHHHHHHH
HHHHHHHHHHHHHHHHHH..............
................................
................................
................................
................................
................................
................................
================================
```

Simulation Configuration

As with Projects 1 and 2, define `t_cs` as the time, in milliseconds, that it takes to perform a context switch (use a default value of `13`).

Also define the aforementioned `t_slice` and `t_memmove` values, measured in milliseconds. For `t_slice`, use a default value of `80`. And for `t_memmove`, use a default value of `10`.

Input File

The input file to your simulator specifies the processes to simulate, as well as their memory requirements. This input file is a simple text file that adheres to the following specifications:

- The filename is `processes.txt`.
- Any line beginning with a `#` character is ignored (these lines are comments).
- All blank lines are also ignored.

- Each non-comment line specifies a single process by defining the process identifier (a single character), the arrival time, the CPU burst time, the number of bursts, the I/O wait time, and the memory requirements (i.e., number of required memory units); all fields are delimited by `|` (pipe) characters. Note that times are specified in milliseconds (ms) and that the I/O wait time is defined as the amount of time from the end of the CPU burst (i.e., before the context switch) to the end of the I/O operation.

An example input file is shown below (**note** that the format has changed from the previous projects, so be careful).

```
# example simulator input file for project 3
#
# <proc-num>|<arrival-time>|<burst-time>|<num-burst>|<io-time>|<memory>
#
A|0|168|5|287|24
B|0|385|1|0|16
D|180|97|5|2499|36
C|180|1770|2|822|48
```

Your simulator must read the input file, then, given these initial conditions, simulate the two separate scheduling algorithms (i.e., SRT and RR). Further, for each of these scheduling algorithms, your simulator must simulate the three placement algorithms (i.e., first-fit, next-fit, and best-fit).

Note that after you simulate each of the above, you must reset the simulation back to the initial set of processes and set your elapsed time back to zero. In short, we wish to compare these algorithms with one another given the same initial conditions. Therefore, you will simulate a total of six simulations.

In general, you must add processes to the queue based on the scheduling algorithm or in the process order shown. In the above example input file, processes for RR will initially be on the queue in the order `A`, `B` at time `0`, then at time `180`, processes `D` and `C` will be added with process `D` added before process `C`.

Note that depending on the contents of this input file, there may be times during your simulation in which the CPU is idle, because all processes are either performing I/O or have terminated. When all processes terminate, your simulation ends.

As with previous projects, all "ties" are to be broken using process order (alphabetical). As an example, if processes `Q` and `R` happen to both finish with their I/O at the same time, process `Q` wins this "tie" and is added to the ready queue before process `R`.

Also, do not implement any additional logic for the I/O subsystem. In other words, there are no I/O queues to implement.

CPU Burst, Turnaround, and Wait Times

CPU Burst Time: CPU burst times are to be measured for each process that you simulate. CPU burst time is defined as the amount of time a process is actually using the CPU. Therefore, this measure does not include context switch times. Note that this measure can simply be calculated from the given input data.

Turnaround Time: Turnaround times are to be measured for each process that you simulate. Turnaround time is defined as the end-to-end time a process experiences for a single CPU burst. More specifically, this is the arrival time in the ready queue through to when the CPU burst is completed. As such, this measure includes context switch times.

Wait Time: Wait times are to be measured for each process that you simulate. Wait time is defined as the amount of time a process spends waiting to use the CPU, which equates to the amount of time the given process is in the ready queue. Therefore, this measure does not include context switch times that the given process experiences (i.e., only measure the time the given process is actually in the ready queue).

Required Output

Your simulator should keep track of elapsed time `t` (measured in milliseconds) for each scheduling algorithm. Initially, `t` is set to `0`. As your simulation proceeds based on the input file and the scheduling and memory placement algorithms, `t` advances to each "interesting" event that occurs, displaying a line of output that describes each event.

Your simulator output should be entirely deterministic. To achieve this, your simulator must output each "interesting" event that occurs using the format shown below.

```
time <t>ms: <event-details> [Q <queue-contents>]
```

The "interesting" events are (with sample output to be provided soon):

- Start of simulation (including what scheduling and memory placement algorithms are being simulated)
- Process enters the system and memory is allocated
- Process starts using the CPU
- Process is preempted
- Process completes its CPU burst
- Process starts performing I/O
- Process finishes performing I/O
- System starts performing defragmentation
- System finishes defragmentation
- Process terminates (by finishing its last CPU burst)
- Process exits the system and memory is deallocated
- End of simulation

Sample Output

Output for the "interesting" events listed on the previous page are shown in the examples below. Please match this output to the extent possible. Minor variations are allowed.

Note that you may assume that allocating memory to a process upon its arrival takes zero time.

Example 1

This example corresponds to the sample input file shown above.

```
time 0ms: Simulator started for RR (t_slice 80) and First-Fit
time 0ms: Process 'A' added to system [Q A]
time 0ms: Simulated Memory:
================================
AAAAAAAAAAAAAAAAAAAAAAA........
................................
................................
................................
................................
................................
................................
................................
================================
time 0ms: Process 'B' added to system [Q A B]
time 0ms: Simulated Memory:
================================
AAAAAAAAAAAAAAAAAAAAAAABBBBBBBB
BBBBBBBB........................
................................
................................
................................
................................
................................
................................
================================
time 13ms: Process 'A' started using the CPU [Q B]
time 93ms: Process 'A' preempted due to time slice expiration [Q B A]
time 106ms: Process 'B' started using the CPU [Q A]
time 180ms: Process 'D' added to system [Q A D]
time 180ms: Simulated Memory:
================================
AAAAAAAAAAAAAAAAAAAAAAABBBBBBBB
BBBBBBBBDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDD....................
................................
................................
```

```
................................
................................
................................
================================
time 180ms: Process 'C' added to system [Q A D C]
time 180ms: Simulated Memory:
================================
AAAAAAAAAAAAAAAAAAAAAAAABBBBBBBB
BBBBBBBBDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCC....
................................
................................
................................
................................
================================
time 186ms: Process 'B' preempted due to time slice expiration [Q A D C B]
time 199ms: Process 'A' started using the CPU [Q D C B]

...
```

Example 2

This example briefly shows the next-fit algorithm and a process completing its CPU burst.

```
time 0ms: Simulator started for RR (t_slice 80) and Best-Fit

...

time 2000ms: Process 'N' added to system [Q H L K M J N]
time 2000ms: Simulated Memory:
================================
HHHHHHHHHHHHHHHH..............JJ
JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
JJJJJJJJJJJJJJJJJJJJJJJ..KKKKK
KKKKKKKKKKKKKKKKKKKKKKKKKKKKLL
LLLLLLLLLLLLLLLLLLLLLNNNNNNNN..
MMMMMMMMMMMMMMMM..............
................................
................................
================================
time 2013ms: Process 'H' started using the CPU [Q L K M J N]
time 2093ms: Process 'H' preempted due to time slice expiration [Q L K M J N H]
time 2106ms: Process 'L' started using the CPU [Q K M J N H]
time 2138ms: Process 'L' completed its CPU burst [Q K M J N H]
time 2138ms: Process 'L' performing I/O [Q K M J N H]
time 2151ms: Process 'K' started using the CPU [Q M J N H]

...
```

Example 3

This example shows defragmentation.

```
time 0ms: Simulator started for SRT and Next-Fit

...

time 2989ms: Process 'K' started using the CPU [Q J H L N M S]
time 3000ms: Process 'R' unable to be added; lack of memory
time 3000ms: Starting defragmentation (suspending all processes)
time 3000ms: Simulated Memory:
================================
HHHHHHHHHHHHHHHH..............JJ
JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
JJJJJJJJJJJJJJJJJJJJJJJ..KKKKK
KKKKKKKKKKKKKKKKKKKKKKKKKKKKLL
LLLLLLLLLLLLLLLLLLLLLNNNNNNNN..
MMMMMMMMMMMMMMMM..............
.....SSSSSSSSSSSSSSSSSSSSSS...
..............................
================================
time 4670ms: Completed defragmentation (moved 167 memory units)
time 4670ms: Simulated Memory:
================================
HHHHHHHHHHHHHHHHJJJJJJJJJJJJJJJ
JJJJJJJJJJJJJJJJJJJJJJJJJJJJJJ
JJJJJJJJJJKKKKKKKKKKKKKKKKKKKK
KKKKKKKKKKKKKLLLLLLLLLLLLLLLLL
LLLLLLNNNNNNNNMMMMMMMMMMMMMMMS
SSSSSSSSSSSSSSSSSSSS.........
..............................
..............................
================================
time 4670ms: Process 'R' added to system [Q J H L N M S R]
time 4699ms: Process 'K' completed its CPU burst [Q J H L N M S R K]

...
```

Required Analysis

As a final output of your simulation, please generate a `simout.txt` file that contains statistics for each of the simulated algorithms. Show the same statistics as in the previous project. Also calculate and show the amount of simulation time (in both milliseconds and as a percentage) that the system spent performing defragmentation.

Submission Instructions

To submit your project, please create a single compressed and zipped file (using `tar` and `gzip`). Please include only source and documentation files (i.e., do not include executables or binary files!).

To package up your submission, use `tar` and `gzip` to create a compressed `tar` file using one of your team member's RCS userid, as in `goldsd.tar.gz`, that contains your source files (e.g., `main.c` and `file2.c`); include a `readme.txt` file only if necessary.

Here's an example showing how to create this file:

```
bash$ tar cvf goldsd.tar main.c file2.c readme.txt
main.c
file2.c
readme.txt
bash$ gzip -9 goldsd.tar
```

Submit the resulting `goldsd.tar.gz` file via the corresponding project submission link available in LMS (`http://lms9.rpi.edu`). The link is in the Assignments section.

For team-based submissions, only one student is required to submit the code. Be sure all team member names are included at the top of each source file in a comment.