CS 446/646 SIMULATOR PROJECT

INTRODUCTION

This assignment has been developed to provide you with a quality experience of the design and operational decisions made by persons developing an Operating System. However, it also incorporates the real world (i.e., advanced academia and/or industry) conditions of managing a slightly larger scale project as well as reading code during the grading component of each phase.

The simulator project will be run in three phases. Each of these phases will be specified in this document although some small changes may be made as the project progresses. The Instructor is open to changes recommended by students as long as the entire project is completed on or before 8 May (Preparation Day).

The simulator must be programmed in C or C++. 5% extra credit will be provided to all students (i.e., graduate and undergraduate) for each program phase that is implemented in C. This means it must be completely C and the make operation must compile it with **gcc** instead of **g++**.

CALENDAR/SCHEDULE OF ASSIGNMENTS

23 February: PA02 program assigned (Phase I)
27 February: PA01 grading turned in (from previous assignment)
22 March: PA02 program turned in (to Week 6 folder assignment)
23 March: PA03 program assigned (Phase II)
27 March: PA02 grading turned in (to Michael's office)
3 April: PA03 program turned in (to Week 9 folder assignment)
6 April: PA04 program assigned (Phase III)
10 April: PA03 grading turned in (to Michael's office)
24 April: PA04 program turned in (to Week 11 folder assignment)
1 May: PA04 grading turned in (to Michael's office)

The schedule above is provided to help make assignments and due dates clear. Make sure you correctly turn in each assignment to the right place by the right time.

GENERAL PROGRAMMING AND DEVELOPMENT EXPECTATIONS

Specific rubrics will be provided for grading each program. However, the following are general expectations of programmers in this 400-/600- level course:

- since you will have an overview of all of the programs, it will be worth your time to consider the subsequent phases as you develop the first program(s); if you have an overlying strategy from the beginning, extending each program will not be difficult

- you may work with any number of fellow students to develop your program design, related data structures, algorithmic actions, and so on for each phase. If you do, you must note which students with whom you worked in your upload text on WebCampus; this is for your protection

- that said, once you begin coding each phase, you may not discuss the project, the strategy(s), debugging, and so on, or work with anyone else under any circumstances; it will behoove you to make sure you have a high-quality design developed prior to beginning your coding process

- all programs must be eminently readable, meaning any reasonably competent programmer should be able to sit down, look at your code, and know how it works in a few minutes. As has been noted in class, this does not mean a large number of comments are necessary; your code itself should read clearly

- the program must demonstrate all the software development practices expected of a 400- (or 600-) level course. For example, all potential user or file interaction failures must be resolved elegantly, any screen presentation must be of high quality, any data structures or management must demonstrate high quality, and so on. If there is any question about the level of these standards, check with the Instructor

- for each programming assignment, each student will upload the program files using his or her own secret ID. The file for each student must be tarred and zipped in Linux as specified below, and must be able to be unzipped on any of the ECC computers include any and all files necessary for the operation of the program. Any extraneous files such as unnecessary library or data files will be cause for credit reduction. As was implemented in PA01 the file must be named **PA0X_<ID Code>.tar.gz** where **X** represents the specific project number, or as an example, **PA03_12345.tar.gz**. The programs must be uploaded at or before 3:30 pm on the date for each specific programming project/phase and delivered to the Instructor's office before 3:30 pm for each grading component. Dates are found previously in this document

- Each individual call to an I/O operation must be conducted by a unique, individual thread. No credit will be earned for the programming part of the assignment if threads are not used for these actions.

CREATING THE PROGRAM META-DATA

The program meta-data components are as follows:

S – operating **S**ystem, used with **`start`** and **end**
A – Program **A**pplication, used with **`start`** and **end**
P – **P**rocess, used with **run**
I – used with **I**nput operation descriptors such as **`hard drive`**, **`keyboard`**
O – used with **O**utput operation descriptors such as **`hard drive`**, **`monitor`**

The program meta-data descriptors are as follows:

end, hard drive, keyboard, monitor, run, start

The cycle times are applied as specified here:

The cycle time represents the number of milliseconds per cycle for the program. For example, if the hard drive has a 50 msec/cycle time, and it is supposed to run for 10 cycles, the hard drive operation must run for 500 msec. You must use an onboard clock interface of some kind to manage this, and the precision must be to the microsecond level. The cycle time in the meta-data program must be set to zero when it is not applicable to the operation, but it is always required as part of the meta-data code.

The form for each meta-data operation is as follows:

<component letter>(<operation>)<cycle time>; <successive meta-data operation>; . . . <last operation>.

Creating example test programs:

The program **programgenerator.cpp** has been developed to support testing and work with this assignment. It can generate test program meta-data with varying parameters. It can also be modified if you wish to use a different operations-generating algorithm(s).

RUNNING THE SIMULATOR

The simulator will input a configuration file that is accepted from the command line, as follows:

```
./simulator config_1.txt
```

Differing configuration files will be used for various testing purposes.

Phase I – Single Program Management

DESCRIPTION

This phase will require the creation of a simple, one-program OS simulator named **OS_Phase_1**, using three states (Enter/Start, Running, Exit). It will accept the meta-data for one program, run it, and end the simulation. Each unique I/O operation must be conducted with its own unique thread

THE CONFIGURATION FILE

Contents of the configuration file are as follows:

Start Simulator Configuration File
Version/Phase: <number>
File Path: <complete file path and name of program file>
Processor cycle time (msec/cycle): <time>
Monitor display time (msec/cycle): <time>
Hard drive cycle time (msec/cycle): <time>
Printer cycle time (msec/cycle): <time>
Keyboard cycle time (msec/cycle): <time>
Log: <Log to Both> OR <Log to Monitor> OR <Log to File>
Log File Path: <complete file path and name of log file name>
End Simulator Configuration File

A specific example of the configuration file is shown here:

Start Simulator Configuration File
Version/Phase: 1.0
File Path: c:\users\billsmith\simproj1.simfile
Processor cycle time (msec): 10
Monitor display time (msec): 25
Hard drive cycle time (msec): 50
Printer cycle time (msec): 500
Keyboard cycle time (msec): 1000
Log: Log to Both
Log File Path: c:\users\billsmith\logfile1.logfile
End Simulator Configuration File

Example meta-data programs:

Balanced Processing and I/O:

Start Program Meta-Data Code:
S(start)0; A(start)0; P(run)14; I(hard drive)13; O(hard drive)15;
P(run)12; O(hard drive)11; P(run)5; I(hard drive)12; O(hard drive)12;
P(run)5; O(monitor)10; P(run)12; O(monitor)10; A(end)0; S(end)0.
End Program Meta-Data Code.

I/O Bound

Program Meta-Data Code:
S(start)0; A(start)0; I(hard drive)10; P(run)13; O(monitor)15;
I(keyboard)9; O(monitor)5; I(hard drive)6; O(monitor)12; P(run)9; I(keyboard)10;
O(hard drive)7; P(run)10; I(keyboard)7; A(end)0; S(end)0.
End Program Meta-Data Code.

CPU Bound

Program Meta-Data Code:
S(start)0; A(start)0; I(hard drive)12; O(hard drive)7; I(keyboard)15;
O(hard drive)11; P(run)10; P(run)9; I(hard drive)14; P(run)11; O(monitor)13;
P(run)10; O(monitor)6; P(run)15; A(end)0; S(end)0.
End Program Meta-Data Code.

SIMULATOR OPERATIONS AND OUTPUT

The simulator will conduct the following operations:

It will load and store the configuration file as needed, it will create a Process Control Block (PCB) for the one program, it will then put the process into a running mode, and it will report on all of its activities. The program must be able to print the output to the screen, to a file, or both, as provided in the configuration file. Contrived* example using previously provided “Balanced Processing and I/O” example input and configuration file:

```
0.000000 - Simulator program starting
0.000003 – OS: preparing process 1
0.000005 – OS: starting process 1
0.000006 – Process 1: start processing action
0.140006 – Process 1: end processing action
0.140007 – Process 1: start hard drive input
0.640007 – Process 1: end hard drive input
0.640008 – Process 1: start processing action
0.730008 – Process 1: end processing action
.
.
.
1.001003 – OS: removing process 1
1.001010 – Simulator program ending
```

*Contrived:
- the times are made up but use the cycles given (in the program meta-data) and the ms per cycle data given (in the configuration file)

Phase II – Batch Processing

DESCRIPTION

This phase will require the creation of a batch program OS simulator named **OS_Phase_2**, using four states (Enter/Start, Ready, Running, Exit). It will accept the meta-data for several programs (i.e., potentially unlimited number), run the programs one at a time, and then end the simulation.

Undergraduate: Must be configurable for, and implement, First In – First Out (FIFO) or Shortest Job First (SJF) CPU scheduling algorithms.

Graduate, or undergraduate extra credit: In addition to FIFO, and SJF, must also be configurable for, and implement, Shortest Remaining Time First - Non Preemptive (SRTF) CPU scheduling algorithms; SRTF must analyze which of the remaining processes is the shortest after the completion of each process, then select the shortest available remaining process.

Note the addition of the CPU Scheduling option in the configuration file below (highlighted in red), and that this is now version 2.0.

THE CONFIGURATION FILE

Contents of the configuration file are as follows:

Start Simulator Configuration File
Version/Phase: <number>
File Path: <complete file path and name of program file>
CPU Scheduling: <scheduling code>
Processor cycle time (msec/cycle): <time>
Monitor display time (msec/cycle): <time>
Hard drive cycle time (msec/cycle): <time>
Printer cycle time (msec/cycle): <time>
Keyboard cycle time (msec/cycle): <time>
Log: <Log to Both> OR <Log to Monitor> OR <Log to File>
Log File Path: <complete file path and name of log file name>
End Simulator Configuration File

A specific example of the configuration file is shown here:

Start Simulator Configuration File
Version/Phase: 2.0
File Path: c:\users\billsmith\simproj1.simfile
CPU Scheduling: FIFO
Processor cycle time (msec): 10
Monitor display time (msec): 25
Hard drive cycle time (msec): 50
Printer cycle time (msec): 500
Keyboard cycle time (msec): 1000
Log: Log to Both
Log File Path: c:\users\billsmith\logfile1.logfile
End Simulator Configuration File

Example meta-data programs:

Balanced Processing and I/O:

Start Program Meta-Data Code:
S(start)0; A(start)0; I(keyboard)5; O(hard drive)14; P(run)11;
P(run)11; I(hard drive)14; A(end)0; A(start)0; P(run)8; O(monitor)8;
I(keyboard)7; P(run)13; O(hard drive)6; A(end)0; A(start)0; I(hard drive)5;
P(run)5; O(hard drive)5; I(keyboard)15; O(hard drive)10; A(end)0; A(start)0;
P(run)7; P(run)15; I(hard drive)14; O(monitor)5; P(run)13; A(end)0; A(start)0;
I(keyboard)10; O(monitor)11; I(hard drive)10; O(monitor)13; I(hard drive)14;
A(end)0; S(end)0.
End Program Meta-Data Code.

SIMULATOR OPERATIONS AND OUTPUT

The simulator will conduct the following operations:

It will load and store the configuration file as needed, it will create a Process Control Block (PCB) for each program, it will then sequentially put each process into a running mode, and it will report on all of its activities. The program must be able to print the output to the screen, to a file, or both, as provided in the configuration file. Contrived* example using previously provided “Balanced Processing and I/O” example input and configuration file:

```
0.000000 - Simulator program starting
0.000002 – OS: preparing all processes
0.000003 – OS: selecting next process
0.000005 – OS: starting process 1
0.000005 – Process 1: start keyboard input
5.000006 – Process 1: end keyboard input
5.000007 – Process 1: start hard drive input
5.700007 – Process 1: end hard drive input
5.700007 – Process 1: start processing action
5.810008 – Process 1: end processing action
.
.
.
12.001003 – OS: removing process 1
12.001004 – OS: selecting next process
12.001007 – OS: starting process 2
12.001007 – OS: start processing action
12.081007 – OS: end processing action
12.081008 – OS: start monitor output
12.208008 – OS: end monitor output
.
.
.
30.553023 – OS: removing process 5
30.553023 – Simulator program ending
```

*Contrived:
- the times are made up but use the cycles given (in the program meta-data) and the ms per cycle data given (in the configuration file)

Phase III – Multi-Programming

DESCRIPTION

This phase will require the creation of a multiprogramming OS simulator named **OS_Phase_3**, using five states (Enter/Start, Ready, Running, Blocked/Waiting, and Exit). It will accept the meta-data for several programs (i.e., potentially unlimited number), run the programs concurrently using a multi-programming strategy, and then end the simulation.

Undergraduate: Must conduct Round Robin (RR) management with a specified Quantum (i.e., number of cycles) time. Must be configurable for and implement a First In – First Out with pre-emption (FIFO-P) CPU scheduling algorithm.

Graduate, or undergraduate extra credit: Must include the undergraduate requirements. In addition to FIFO-P, the simulator must also be configurable for, and implement the Shortest Remaining Time First – Preemptive (SRTF-P) CPU scheduling algorithm; SRTF-P must analyze which of the remaining processes is the shortest at the end of each interrupted process whether the Quantum time was completed or not, then select and run the shortest available remaining process.

All students: For this phase, when an I/O process is called (i.e., the I/O thread is deployed), the running process must be placed in the Blocked/Waiting state. When the I/O process (i.e., thread) has completed, it must interrupt the processor at the end of the presently running process cycle* causing the OS to move the process back into its appropriate place in the Ready state.

*Example of presently running process cycle: For the circumstance that the presently running process is programmed to run for 10 cycles and each processor cycle is configured to be 10 ms long, if an I/O process completes its actions and sends an interrupt signal to the processor while the presently running cycle is half-way through its seventh cycle, the process must complete the seventh cycle but then be moved to the Ready queue while the OS uses processing time to re-enqueue BOTH the process returning from the Running state AND the process being returned from the Blocked/Waiting state. As usual, the simulator must display the actions it is taking to handle these processes. Another side note to this is that the system must keep track of completed – and not completed – process cycles when processes are interrupted out of the Running state

Note the addition of the Quantum time in the configuration file below, and note that this is now version 3.0.

THE CONFIGURATION FILE

Contents of the configuration file are as follows:

Start Simulator Configuration File
Version/Phase: <number>
File Path: <complete file path and name of program file>
CPU Scheduling: <scheduling code>
Quantum time (cycles): <number of cycles for quantum>
Processor cycle time (msec/cycle): <time>
Monitor display time (msec/cycle): <time>
Hard drive cycle time (msec/cycle): <time>
Printer cycle time (msec/cycle): <time>
Keyboard cycle time (msec/cycle): <time>
Log: <Log to Both> OR <Log to Monitor> OR <Log to File>
Log File Path: <complete file path and name of log file name>
End Simulator Configuration File

A specific example of the configuration file is shown here:

Start Simulator Configuration File
Version/Phase: 3.0
File Path: c:\users\billsmith\simproj1.simfile
CPU Scheduling: FIFO-P
Quantum time (cycles): 6
Processor cycle time (msec): 10
Monitor display time (msec): 25
Hard drive cycle time (msec): 50
Printer cycle time (msec): 500
Keyboard cycle time (msec): 1000
Log: Log to Both
Log File Path: c:\users\billsmith\logfile1.logfile
End Simulator Configuration File

Example meta-data programs:

Balanced Processing and I/O:

```
Start Program Meta-Data Code:
S(start)0; A(start)0; I(keyboard)5; O(hard drive)14; P(run)11;
P(run)11; I(hard drive)14; A(end)0; A(start)0; P(run)8; O(monitor)8;
I(keyboard)7; P(run)13; O(hard drive)6; A(end)0; A(start)0; I(hard drive)5;
P(run)5; O(hard drive)5; I(keyboard)15; O(hard drive)10; A(end)0; A(start)0;
P(run)7; P(run)15; I(hard drive)14; O(monitor)5; P(run)13; A(end)0; A(start)0;
I(keyboard)10; O(monitor)11; I(hard drive)10; O(monitor)13; I(hard drive)14;
A(end)0; S(end)0.
End Program Meta-Data Code.
```

SIMULATOR OPERATIONS AND OUTPUT

The simulator will conduct the following operations:

It will load and store the configuration file as needed, it will create a Process Control Block (PCB) for each program, it will then put each process into the Ready queue, use the appropriate strategy to select and run the first process, place it into Running mode, and then watch for an interrupt that indicates that either: 1) the Quantum time or 2) the I/O action have been completed. From that point, the OS must use the configured algorithm to select the next process to be run, and place it in the Running mode. Each process must be removed as soon as it has completed, and the simulator must shut down after the last process has been completed and removed. As before, the simulator must report on as many actions as reasonably necessary to show what the system is doing at any given moment, and then be able to print the output to the screen, to a file, or both, as provided in the configuration file.

Contrived* (selected) example components:

```
<time to microsecond precision> - Simulator program starting
<time to microsecond precision> – OS: selecting next process
<time to microsecond precision> – OS: preparing process 1
<time to microsecond precision> – OS: starting process 1
<time to microsecond precision> – Process 1: start keyboard input
<time to microsecond precision> – OS: starting keyboard input
<time to microsecond precision> – OS: process 1 to Blocked
<time to microsecond precision> – OS: selecting next process
<time to microsecond precision> – OS: starting process 2
<time to microsecond precision> – Process 2: start processing action
<time to microsecond precision> – OS: quantum time interrupt
<time to microsecond precision> – OS: process 2 to Ready
<time to microsecond precision> – OS: selecting next process
<time to microsecond precision> – OS: starting process 3
<time to microsecond precision> – Process 3: start hard drive input
<time to microsecond precision> – OS: starting hard drive input
<time to microsecond precision> – OS: process 3 to Blocked
<time to microsecond precision> – OS: selecting next process
<time to microsecond precision> – OS: starting process 4
.
.
.
<time to microsecond precision> – OS: process 1 end
<time to microsecond precision> – OS: process 1 removed
<time to microsecond precision> – OS: selecting next process
<time to microsecond precision> – OS: starting process 5
.
.
.
<time to microsecond precision> – OS: process 3 end
<time to microsecond precision> – OS: process 3 removed
<time to microsecond precision> – Simulator program ending
```

*Contrived:
- This will look different every time it is run, even with the same program meta-data; the key requirement is that any time a system change is made, the OS is recognized as making the appropriate change, and the programs continue on from their appropriate queue