CS 480 SIMULATOR PROJECT

INTRODUCTION

This assignment has been developed to provide students 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 larger scale project as well as reading code during the grading component of each phase.

The simulator project will be run in at least four 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, including grading, is completed on or before 6 December 2019 (course term end).

The simulator must be programmed in C. This means it must be written completely in C and the ***make*** operation must compile it with **`gcc`** (i.e., as opposed to **`g++`**). All programs are required to use a ***make*** file with the **`-f`** switch, and all files must be compiled with the **`-Wall`** switch, the **`-std=c99`** switch, and the **`-pedantic`** switch and use a standard make file naming protocol. The make file names are the project or program names with a ***_mf*** appended, and are used as follows: ***make -f myprog_mf***. Students are also required to demonstrate effective modularity by breaking the various functions out into appropriately organized files. More information about programming standards and operations is provided later in this document.

CALENDAR/SCHEDULE OF ASSIGNMENTS

~~26 Aug: PA01 assigned in Week 1 folder~~
~~13 Sep: PA01 program due in Week 1 folder (3 weeks)~~
16 Sep: PA02 assigned in Week 4 folder
20 Sep: PA01 grading due on paper to Michael's office (under door)
04 Oct: PA02 program due in Week 4 folder (3 weeks)
07 Oct: PA03 program assigned in Week 7 folder
11 Oct: PA02 grading due on paper to Michael's office (under door)
01 Nov: PA03 program due in Week 7 folder (4 Weeks)
04 Nov: PA04 program assigned in Week 11 folder
08 Nov: PA03 grading due on paper to Michael's office (under door)
29 Nov: PA04 program due in Week 11 folder (~4 Weeks)
06 Dec: PA04 grading due on paper to Michael's office (under door)

The schedule above is provided to help make assignments and due dates clear. Make sure each assignment is correctly turned in to the right place by the right time; loss of some or all credit may occur for incorrect assignment preparation and/or uploading management.

Also note that due to time limitations, it is unlikely that any deadlines will be changed.

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-level course:

- since students will have an overview of all of the programs, be sure to consider the subsequent phases as the first programs are developed; an overlying strategy from the beginning will significantly support extending and/or expanding each program

- students may work with any number of fellow students to develop the program design, related data structures, algorithmic actions, and so on for each phase. Students who do work together must note which students with whom they worked in the upload text on BlackBoard Learn; this is for the students' protection

- that said, once a student begins coding each phase, s/he may not discuss or work with anyone on the development, coding, and/or debugging process. Strategy(s) may still be discussed but without specific Instructor permission, no student may view or be involved with the code of another. It will be a good idea to make sure a high-quality design has been developed prior to beginning the coding process

- all programs must be eminently readable, meaning any reasonably competent programmer should be able to sit down, look at the code, and know how it works in a few minutes. This does not mean a large number of comments are necessary; the code itself should read clearly. Refer to the Programming Standards document for best practices and requirements; this document will be used as the final reference during the grading phases

- the program must demonstrate all the software development practices expected of a 400-level course. For example, all potential file failures must be resolved elegantly, any screen presentation must be of high quality, any data structures or management must demonstrate high quality, supporting actions and components must demonstrate effective modularity with the use of functions, there may not be any global or single-letter variables, and so on. If there is any question about these standards, check with the Instructor

- one example of clean modularity is that no functions other than the main function may be in the main driver file. All simulator actions, utility functions, and any other support code must be in other files with file names that clearly indicate what kind of support code will be contained within. It is expected that for the final assignment, there will be at least five separate C files, but in most cases, no more than ten to twelve

- as stated previously in this document, students must use the C programming language exclusively for these projects; use of any other programming language will likely result in no credit earned for a given project

- students may use any of the C libraries specified in this paragraph as needed, but may not use any other libraries, and may not use pre-developed data structures, tools, or programs that students are expected to write for this project. Examples of appropriate libraries would be: ***sys/time.h***, ***math.h***, ***stdio.h***, ***stdlib.h***, ***pthread.h***, ***time.h***, and ***string.h***. Note however that any utility functions that are to be used must be written by the students. Examples of utility functions are any functions that start with "***str***" (e.g., ***strcpy***, ***strcat***, ***strtok***, etc.), or functions that implement conversions such as ***atoi***, ***atof***, etc.; any of these or other functions that conduct utility actions must be written by the individual student using them. Students who want to use other libraries or have questions about utility functions must check with the Instructor for approval. If a given function or library other than mentioned in this paragraph is approved, the approval will be shared with all students in the class. The use of unapproved headers/libraries and/or utility functions will cause a reduction in credit.

- In addition, students must use POSIX/pthread operations to manage the I/O operations but may not use previously created threads such as timer threads (e.g., ***sleep***, ***msleep***, ***usleep***, etc.). Note that this means that the program must be developed in a Linux environment. If there are any questions on this, ask the Instructor so your grade is not harmed by an incorrect choice.

- all programs must compile without errors or warnings, and run on the CEFNS linux system (i.e., **`linux.cefns.nau.edu`**). All programs must also be tested for any memory issues using the Valgrind/Memcheck software product, and this must be tested on the CEFNS system as well. Individual students may develop their programs in any environment they choose* but – as stated – the program must compile and run, and pass the Valgrind tests, on the CEFNS system. It will be a good idea to check individual programs on this system well before the program is due, which will probably include during the development time.

*While this specification allows for the use of MS/Windows tools, there will come a point in the development process – very likely in PA02 – that you will have to use Linux to implement threading operations to meet the program requirements. You are advised to jump right into the Linux environment and get through the initial struggles during your development of the first program as it will be the easiest assignment

- for each programming assignment, each student will upload the program files using his or her own secret ID which will be generated and provided to students in the BBLearn grading columns; note that this is NOT the NAU student 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 Linux computer. Any and all files necessary for the operation of the program must be included. Any extraneous files such as unnecessary library, data files, or object files will be cause for credit reduction. The file must be named ***PA0X_<ID Code>.tar.gz*** where **X** represents the specific project number, and the students secret ID code is placed in the <ID CODE> location. An example would be ***PA01_123456.tar.gz***. The programs must be uploaded at or before 6:00 pm on the date for each specific programming project/phase, and delivered to the Instructor's office before 3:00 pm for each grading component. Dates are found previously in this document.

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`**
M – **M**emory operations
I – used with **I**nput operation descriptors such as **`hard drive`**, **`keyboard`**
O – used with **O**utput operation descriptors such as **`hard drive`**, **`printer`**, **`monitor`**

The program meta-data descriptors are as follows:

access, allocate, end, hard drive, keyboard, printer, 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 a device has a 50 msec/cycle time, and it is supposed to run for 10 cycles, the device operation (i.e., the timer for that device) must actually run for 500 msec. An onboard clock interface of some kind must be used 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. To repeat, the simulator must represent real time; if the operations take 10 seconds, the simulator must take 10 seconds.

A support file **`simtimer.c`** and its header file will be provided for student consideration. It is not required for students to use this code, however timer displays used for each of the assignments must correctly show the time at microsecond precision (i.e., 0.000001 sec) as specified previously. The microsecond display is demonstrated in the PA01 demonstration program, which will also be provided.

The form for all meta-data operations other than memory management is as follows (each of these components is called an "op code" for purposes of this program):

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

The form for memory management operations is as follows:

To initially allocate memory for a program,

M(allocate<requested memory **B**ase + requested memory **A**llocation, as long integer>; (e.g., M(allocate)BBBBBAAAA;) where the base number is a location in memory in kilobytes (KB), the amount allocated is in kilobytes (KB), and the sum, or total requested, cannot go over 100 MB (102400 KB)

To access memory within a program,

M(access)<requested memory **B**ase + requested memory **A**ccess, as long integer> (e.g., M(access)BBBBBAAAA;) where the base number is a location within the specified segment of memory in kilobytes (KB), and the memory access requested is in kilobytes (KB)

Creating example test programs:

The program **`proggen.c`** has been developed to support testing and work with this assignment. It can generate test program meta-data with varying parameters, although it does not generate memory access or allocation op codes as these need to be uniquely created. It can also be modified as needed to use different operations-generating algorithm(s). Besides using this program for its intended purpose, students can also observe expected programming practices especially as relates to readability. As noted previously in this document, comments are allowed but not expected; program code should be eminently readable by the use of self-documenting identifiers. That said, this code is significantly commented to support learning.

RUNNING THE SIMULATOR

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

```
./sim0x config_y.cnf
```

*x is the project number (1-4), and y is the number of a given configuration file

Note that the program must work in this form. The use of any console input actions for the configuration or meta data files will be cause for significant credit reduction. The configuration file must be used as a command-line argument, and the meta data file must be opened after acquiring the meta data file name from the configuration file. Any deviation from this requirement will cause a reduction of credit.

Also note that differing configuration files will be used for various testing purposes.

Phase I (PA01) – Input Data Management

DESCRIPTION

This phase – which is a review of data structures, implemented in C – will require the creation of two data-acquisition operations that upload and store two sets of data: the Simulator configuration file, and the Simulator meta-data file.

While this is a stand-alone project, students are wise to assess the next steps in the project so they can consider the requirements and develop their code to be modular components of the larger system. The last project or two will be pretty complicated but will not be difficult to develop as long as the base components have been developed well.

CONFIGURATION FILE

As mentioned, the overlying Simulator project will use two input files. The first file will be a configuration file that supports general information that the Simulator needs to accomplish its tasks. The following is an example configuration file, and is provided separately for use in the implementation of this assignment.

All programs must be able to input and store the contents of this file:

```
Start Simulator Configuration File
Version/Phase: 1.0
File Path: Test_3.mdf
CPU Scheduling Code: NONE
Quantum Time (cycles): 55
Memory Available (KB): 12000
Processor Cycle Time (msec): 10
I/O Cycle Time (msec): 20
Log To: Monitor
Log File Path: logfile_1.lgf
End Simulator Configuration File.
```

The first "Start Simulator . . . " line is ignored. The following lines may appear in any order:

Version/Phase: This line will have a version number such as 1.25, 2.3, 3.44, etc. Note that the version/phase will be different for each assignment and will be floating point values; in many cases, student programs are likely to have evolving fractional version numbers as the programs are developed. Specification: 0.0 ≤ V/P ≤ 10.0

File Path: This line must contain the file path where the meta-data will be found. While it could include directories, the assignment requirement is that the data be in the same directory as the program

CPU Scheduling Code: This line will hold any of the following: ***NONE***, ***FCFS-N***, ***SJF-N***, ***SRTF-P***, ***FCFS-P***, ***RR-P***. If **NONE** is encountered, it should default to ***FCFS-N***. No other code names are allowed, and if any are found, the data access must be aborted, and the configuration function must signal failure to the calling function. Note that the configuration input function should not display any output – this will be discussed later.

Quantum Time: This line will hold an integer specifying the quantum time for the Simulator. For the first couple of projects, this will be zero and/or will be ignored by the program although it must still be stored in the data structure. Specification: 0 ≤ Q ≤ 100

Memory Available: This line will hold an integer specifying the system memory that will be available. For the first couple of projects this may also be zero and/or ignored although it must still be stored in the data structure. Specification: 0 ≤ MA ≤ 102400 (100 MB, in KB form)

Processor Cycle Time (msec): This line will hold an integer cycle time that will specify the number of milliseconds each processor cycle will consume.
1 ≤ PCT ≤ 1,000

I/O Cycle Time (msec): This line will also hold an integer cycle time like the processor cycle time.
1 ≤ IOCT ≤ 10,000

Log To: This line will hold one of three terms, being ***Monitor***, ***File***, or ***Both***. No other code names are allowed, and if any are found, the data access must be aborted, and the configuration input function must signal failure to the calling function

Log File Path: This line will hold the file path of the log file, which is used if "Log To:" has selected either ***File*** or ***Both***. It must still hold some string quantity even if "Log To:" is set to ***Monitor*** (e.g. no logfile, or none)
Finally, the last "End Simulator . . . " is ignored

Most failure issues such as missing file, corrupted file data, or incomplete data must stop the function and elegantly respond. This includes closing the input file if it is open, halting any other processing, file I/O, or file management, and providing an indication to the calling function as to what went wrong. Remember that the function must communicate the error to the calling function; error messages must all be printed from the main function.

All data input from the file must be stored in some kind of data structure(s), and be available to the calling function, or subsequently called functions, as random-access data. You must upload data in small segments (e.g., one configuration item or one meta data operation at a time). If you try to upload all of the data at once, you may bring down your program, especially when you are using Valgrind. One of your tasks for this assignment is to select the best data structure(s) for this data, develop the data structure, and implement it to safely satisfy the specified requirements.

IMPORTANT: As mentioned previously, no processing function should ever display an output. The configuration input function is a good example. If there is a failure in the function, it should provide some form of messaging back to the calling function so the calling function can manage the issue, which may include displaying an error message and/or shutting down the program. Any processing functions (i.e., functions not specifically focused on I/O actions other than its specifications) that conduct any I/O will experience a significant reduction of credit. As a note, the simulator function's task is to display simulated operations, so it is acceptable for that function, along with its subordinate functions, to display or store output.

META-DATA FILE

The second file will hold meta-data, which specifies program actions in a coded text form such as the one that follows. See the previous reference in this document to identify the meta-data components.

```
Start Program Meta-Data Code:
S(start)0; A(start)0; M(allocate)5121024 I(hard drive)14; P(run)7;
O(hard drive)5; I(keyboard)8; O(hard drive)10; I(keyboard)15;
P(run)10; O(monitor)11; I(keyboard)14; M(access)5120256; O(printer)8;
A(end)0; S(end)0.
End Program Meta-Data Code.
```

Students will be provided a meta-data file example. While the data acquisition program must upload ***any*** meta-data file of ***any*** size, ***any*** number of actions, ***any*** number of programs, etc., all student programs must work correctly on the given meta-data file. As mentioned previously in this document, the program **`proggen.c`** is provided that generates programs of varying complexity, although it does not generate the memory commands (for reasons that will become clearer later).

The function that opens and acquires data from the meta-data file must carefully screen the command operations so that it captures any corrupted data, aborts the input process, and signals failure back to the calling function. When the data is stored, it should contain the command letter, the operation string, and the cycle time (or memory size) value. The following is an example taken from the meta-data set shown previously in this document:

For the file data: **`I(hard drive)14`**, the command letter ('I'), the operation string ("hard drive") and the cycle time (14) must be stored for that specific action. A memory example would be for **`M(allocate)12250812`**, the command letter ('M'), the operation string ("allocate"), and the memory value as specified previously must be stored for that specific action. Students should review the example file provided and the video tutorials for further reference.

ASSIGNMENT

As specified in the description, students are to develop modules that, when called, input and store the Simulator configuration data and the Simulator meta-data. By definition, these must be modular so students can insert them into the expanding Simulator project phases as they are assigned.

Once the modules are developed, they must be executed in a driver program and tested with varying data to prove they are working correctly.

IMPORTANT: It will not be enough to hack together a program that seems to work. All programs must be eminently readable since each program will be graded by one of your peers in the class in a double-blind anonymous system. Even if your program works – or seems to work – correctly, it will not receive full credit if it is difficult to read and/or understand. Refer to the programming standards provided for this class as well as the example program code provided. While these standards are not an absolute requirement, the intent (readability) of the standards is a requirement. Also review each assignment rubric early in your development process so you will know how your program will be graded. To repeat: All code must be eminently readable. Use of single-letter variables, lack of white space, lack of curly braces used for every selection or iteration statement, etc. will be cause for loss of credit.

IMPORTANT (again): As mentioned previously in this document, the programming quality of a 400-level course is expected here. While this Simulator project is much easier than working with a real operating system, the programming is still non-trivial. It is strongly recommended that students start on each of the Simulator assignments as soon as they are posted; late starts and last-minute programming attempts will not be successful.

As a final note, specifically for this assignment, 1) you must have a file holding only the main driver function, 2) an upload files file with the code for implementing the two file uploads, and 3) you may have one or two utility files (e.g., you may have one string utility file and another general utility file, or you may use one file for all your utilities). These will provide appropriate modularity for the current program, and will also provide support for your work and grading with the next succeeding assignments.

The meta-data file, which is provided for this assignment, is shown here.

```
Start Program Meta-Data Code:
S(start)0; A(start)0; P(run)9; I(hard drive)27;
O(printer)75; I(keyboard)120; O(printer)45; P(run)8; O(printer)45;
I(keyboard)110; O(printer)45; P(run)14; A(end)0; S(end)0;
End Program Meta-Data Code.
```

The resulting output of the simulator must be displayed as shown on the next page, or in an equivalent way. You may add presentation components to this if they assist with simulator operation clarity. Note that the "Loading" displays must actually represent the loading operations (i.e., output the text as each is done). The times must be displayed as calculated, and as previously mentioned, after Phase 1, the simulator must take roughly the correct amount of time for each operation.

Here is an example output product of this program.

```
Config File Upload Component
============================

Config File Display
===================
Version                : 1.05
Program file name      : metadata0.mdf
CPU schedule selection : FCFS-N
Quantum time           : 55
Memory Available       : 111
Process cycle rate     : 10
I/O cycle rate         : 20
Log to selection       : Both
Log file name          : logfile_1.lgf
```

Continues on next page but is all one contiguous display output.

```
Meta Data File Upload Component
===============================

Meta-Data File Display
======================
Op code letter: S
Op code name  : start
Op code value : 0

Op code letter: A
Op code name  : start
Op code value : 0

Op code letter: P
Op code name  : run
Op code value : 9

Op code letter: I
Op code name  : hard drive
Op code value : 27

Op code letter: O
Op code name  : printer
Op code value : 75

Op code letter: I
Op code name  : keyboard
Op code value : 120

Op code letter: O
Op code name  : printer
Op code value : 45

Op code letter: P
Op code name  : run
Op code value : 8

Op code letter: O
Op code name  : printer
Op code value : 45

Op code letter: I
Op code name  : keyboard
Op code value : 110

Op code letter: O
Op code name  : printer
Op code value : 45

Op code letter: P
Op code name  : run
Op code value : 14
```

```
Op code letter: A
Op code name  : end
Op code value : 0

Op code letter: S
Op code name  : end
Op code value : 0
```

Phase 2 (PA02) – Single Program Simulator

DESCRIPTION

This phase will begin your real simulation work by processing several programs (processes) in one simulator run. The simulator must conduct all of the required operations of Sim01, and include the extensions of Sim02 specified here:

- the simulator must manage, process, and display the simulation of a single program with multiple operation commands. The number of operation commands will not be known in advance.

- the simulator must output the simulation results a) to the monitor, b) to a file (without displaying to the monitor) with the name specified in the configuration file, or c) both; it is important to note again that the monitor display must occur as the operation commands occur in real time with the appropriate time quantities. The selection of monitor, file, or both must be made in the configuration file. In addition, the output to file operation must all be conducted at one point, after the simulation has been run. This means that all the displayed operation statements with the times, process numbers, operation descriptions, etc. must be stored line by line until the simulation has finished, at which time the data is output to a file.

- the simulator must be initially configured for First Come First Served – Non-preemptive (FCFS-N). This means that if FCFS-N is shown in the configuration file, the simulator will progress through the processes as they were found in the meta-data file.

- the simulator must show one of four states that the process is in: new, ready, running, or exiting

- the simulator must now use a POSIX thread to manage the cycle time for all I/O operations. This is not required for run operations; these may be run as normal functions or threads at the student's discretion. It also does not apply to the memory management operations which may optionally run for a time as a place holder. These operations will be handled differently in future simulation projects. Note that students are required to create their own timer threads; as mentioned previously in this document, no threads created in, or found in, available libraries, such as ***sleep***, ***usleep***, ***nanosleep***, etc. may be used. For purposes of this assignment, the simulator does not support a multi-tasking (multi-programming) environment. For that reason, the simulator must still wait for each I/O operation to complete before moving on to the next operation command.

- the system must report at least each of the following operation actions:
    - system start and end
    - any state change of any of the processes (e.g., ready, running, etc.)
    - any start or end of any operation command (e.g., hard drive input or output, keyboard input, monitor output, run process actions, etc.)

An example config, metadata, and output file will be provided in BBLearn

Phase 3 (Sim03) – Batch Program Simulator with Memory Management

(Advance Description – Draft)

DESCRIPTION

This phase will offer you the opportunity to learn about memory management by creating your own Memory Management Unit (MMU). You will also be extending the batch processing operations by implementing two different CPU scheduling strategies. The simulator must conduct all of the required operations of the previous (Sim02) simulator, with the addition of the following specifications:

- the simulator must now be configurable for either First Come First Served – Non-preemptive (FCFS-N) or Shortest Job First – Non-preemptive (SJF-N). This means that if FCFS-N is shown in the configuration file, the simulator will progress through the processes as they were found in the meta-data file. However, if SJF-N is shown in the configuration file, the simulator will progress through the processes in such a way that the jobs are run in order by their total operation times from shortest operation times to longest operation times. Note that this does not mean the shortest number of operations or cycles; the actual running times for all operations must be calculated and compared. Also note that if two or more processes have the exact same running times, they are to be scheduled as FCFS. This is an unlikely scenario but must be considered and managed.

- the simulator must continue to use a POSIX thread to manage the cycle time for all I/O operations. This is an option but not a requirement for the run operations which must also still be simulated using the clock times as in Sim02. It also does not apply to the memory management operations which will be handled as specified in the next topics. Note that students are still required to create their own timer threads; no previously created threads such as ***sleep***, ***usleep***, ***nanosleep***, etc. may be used. For purposes of this assignment, the simulator still does not support a multi-tasking (multi-programming) environment. For that reason, the simulator must still wait for each run and I/O operation to complete before moving on to the next operation command.

- the simulator must show one of four states that each current process is in: new, ready, running, or exiting

- for the memory management, the following specifications must be followed:

  - the total memory authorized for a given process will be placed in the configuration file, as previously specified

  - the process will allocate a segment of memory using the M<allocate>BBBBBAAAA operation command provided previously. The MMU must first check that the amount of memory requested is not larger than that specified in the configuration file. Then the MMU must place the Process ID number, the segment number, the requested base, and the requested memory in its own table for future reference

    * if the same or another process attempts to allocate the same memory block, this must throw a segmentation fault. Example: if the original allocation was 15100150 (memory base 1510 KB, and memory offset 150 KB) and the same or another process attempts to allocate 16150100 (memory base 1615 KB and memory offset 100 KB), this must throw a segmentation fault, end that specific process, and report the issue

  - the process will request memory using the M<access>BBBBBAAAA operation command provided previously. The MMU must then check if the process requesting this memory has access to this data within the segment originally allocated. Examples:

    * if the original allocation was M<allocate>10100225 (i.e., memory base 1010 KB, and memory allocated 225 KB) and the request is M<access>10130155 (i.e., memory base 1013 KB, and the memory requested is 155 KB, the memory is available and the system must approve the action

    * if the original allocation was M<allocate>12220175 (i.e., memory base 1222 KB and memory allocated 175 KB), the request M<access>18880125 (i.e., memory base 1888 KB, and memory offset 125 KB) will be 16 kilobytes more than was allocated so the system must show a segmentation fault and stop that specific process. This will not stop the batch processing which should continue with the next appropriate process

  - note that the memory management operation must show its starting and ending time the same as any other operations, but it does not require any specific cycle time

- the system must report at least each of the following operation actions:
    - system start and end
    - any state change of any of the processes (e.g., ready, running, etc.)
    - any selection of a new process as a result of the scheduling requirement
    - any start or end of any operation command (e.g., hard drive input or output, keyboard input, monitor output, run process actions, etc.); note that the ends of all of the I/O operations will occur at times between other scheduled operations

Phase 4 (Sim04) – Multiprogramming Simulator

(Advance Description – Draft)

DESCRIPTION

This phase will mark the culmination of how a multiprogramming operating system works. The program will extend the previous programming assignments in such a way that a user can view an operating system in action. The Sim04 system must effectively demonstrate concurrency with reasonably correct times for running and I/O operations. Threads may be used but are not required as long as the concurrency requirement is met; the program must appear to run the I/O operations in parallel with the run and housekeeping operations and as mentioned, the times for the I/O operations returning from their work (as interruptions) must be pretty close to the correct times. As before, all of the requirements of the previous assignment phases must still be supported, which includes the ability to run one or more programs with FCFS-N (i.e., first come, first served, non preemptive) and SJF-N (i.e., shortest job first, non preemptive) scheduling strategies on any set of given meta-data. In addition, all previous specifications still remain, such as no use of various sleep functions, clean, readable programming code, correct assignment file naming and management, etc. It would be a good idea to review these before attempting this next project. New requirements are specified below.

- as before, threads may be used for all I/O operations as needed. For any of the new strategies, which are FCFS-P, SRTF-P, and RR-P, the threads must be created and the program must move on to the next available operation command. There is likely to be some synchronization management to keep race conditions from occurring; since it is likely the I/O threads will be updating the same data, which must be released to the display when the thread has completed, this is a pretty clear reader/writer problem, and it must be managed as such.

- the FCFS-P (i.e., first come, first served, preemptive) strategy must bring in operation commands in order of the process entry. In addition, when a given process is returned from being blocked, it must be placed back into the scheduling queue in its original order. Example: In a program where there are 8 processes, process operations 0, 1, 2, and 3 might all start with I/O, and are sent out as threads. If process 3 returns first, it must be placed back in the scheduling queue in order (e.g., 3, 4, 5, 6, 7) so that the next operation command of process 3 is run next. Later when process 0 is freed, it must be placed in order (e.g., 0, 3, 5, 6, 7) so that the next operation command of process 0 is run next, and so on

- the SRTF-P (i.e., shortest remaining time first, preemptive) strategy must find the process with the shortest total remaining time before each operation command is run, and run the operation command of that process next

- the RR-P (round robin, preemptive) strategy starts the same as FCFS, however when a process is returned either from running or from being blocked, it simply goes back onto the end of the scheduling queue in the order it was returned

- the P/run operation must stop after each complete individual cycle and check for interrupts that have occurred while it was running that cycle. Example: three I/O operation threads are running when a P/run operation requiring 7 cycles is placed in the processing state, where the processing cycle time is 30 mSec. During the first cycle, no interrupts occur, so the system checks for the interrupts, finds none, and starts the second cycle. At a point 14 mSec into the next run cycle, one I/O thread completes and sends an interrupt signal, and at 22 mSec into the same run cycle, another I/O thread completes and sends an interrupt signal; note that these are concurrent actions. The P/run action must complete its 30 mSec cycle but when it checks for, and finds the two interrupt requests, the P/run process must be placed back into the scheduling queue (appropriately, as specified previously in this document, and with its 7-cycle requirement reduced to 5), and the two I/O actions must be processed (e.g., each I/O completion transaction must be posted, the processes having these I/O operations must be unblocked and appropriately placed back into the scheduling queue, etc.).

Also, if no I/O operation interrupts occur, the P/run operation must still be stopped at the quantum time (e.g., even though it has a 7-cycle requirement, if the quantum is 3, the P/run operation must be stopped after the third cycle, and it must be placed back in the scheduling queue and its 7-cycle requirement must now be reduced to 4

- note that it is likely that two or more I/O operations may finish and drive an interrupt while a P/run operation is being conducted; this will require some kind of queueing management for the waiting interrupt requests

- also note that for I/O-bound programs, many, and possibly all, of the processes may be blocked for periods of time; the program must show the processor idling if there are no Ready-state operation commands to be run

- the system must report at least each of the following operation actions:
    - system start and end
    - any state change of any of the processes (e.g., ready, running, etc.)
    - any selection of a new process as a result of the scheduling requirement
    - any start or end of any operation command (e.g., hard drive input or output, keyboard input, monitor output, run process actions, etc.); note that the ends of all of the I/O operations will occur at times between other scheduled operations

- memory management is not required for this assignment, however extra credit will be earned if memory management is correctly implemented