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 19 November 2020 (course term end).

Development requirements

- The simulator must be programmed and written completely in the C programming language, and run in the Linux environment
- The ***make*** operation must compile all files with **`gcc`** (i.e., as opposed to **`g++`**); the make file must be structured in the same form as the Instructors' provided file
- All programs are required to use a ***make*** file with the **`-f`** switch followed by a make file name (NOT ***makefile***)
- All program files must be compiled with the **`-Wall`** switch, the **`-std=c99`** switch, and the **`-pedantic`** switch
- 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 as well is in the project rubric.

CALENDAR/SCHEDULE OF ASSIGNMENTS

~~11 Jan: Sim01 assigned in Week 1 folder~~
~~28 Jan: Sim01 program due in Week 1 folder (~3 weeks)~~
~~01 Feb: Sim02 assigned in Week 4 folder~~
~~04 Feb: Sim01 grading due in Week 4 folder~~
~~18 Feb: Sim02 program due in Week 4 folder (~3 weeks)~~
22 Feb: Sim03 program assigned in Week 7 folder
25 Feb: Sim02 grading due in Week 7 folder
18 Mar: Sim03 program due in Week 7 folder (3+ Weeks)
22 Mar: Sim04 program assigned in Week 11 folder
25 Mar: Sim03 grading due in Week 11 folder
15 Apr: Sim04 program due in Week 11 folder (3+ Weeks)
22 Apr: Sim04 grading due in Week 15 folder
{22 Apr: Last day of classes}

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.

Posting of Simulator Code to repositories (e.g., github, Bitbucket, SourceForge, etc.)

This project has the potential to be a very powerful representation of your best work and you SHOULD show it off to potential employers. However, according to NAU policy and the policies for this course, sharing this code with other students is a violation of academic integrity. And it should be noted that depending on the circumstances, there is a fair chance you could be charged with an academic integrity violation even after leaving this class. That would be bad since the potential sanctions would be more significant at the University level.

Another issue is the fact that you are given virtually all of the code for the first assignment which is used for all of the others. Representing this code as your own is plagiarism, and potentially, fraud.

So how do you handle this since you would like to show your project to potential employers? Here are acceptable ways to handle this:

1. When showing off your code, make it clear inside the code files as well as in the presentation of the code that the file I/O operations were developed by me. You do not have my permission to represent it as your own code. While I do provide you with the code, I (and your potential employers) expect the credit for development of the code to be given to the appropriate party. Just do that. No harm, no foul. If you actually create the file access code on your own and it is distinctively different from mine, then it is yours and you should represent it as such. Again, no harm, no foul.

2. As for posting the code, you should set it to private status and note in your resume that it will be provided to potential employers on request. They are fine with that. If you are in the process of being interviewed or your potential employer is interested enough in your work to request to see the code, open it up for a brief time (three or four days, but no longer than a week). I am not just okay with that, I will be proud that you did it.

3. Finally, when you graduate, the University does not have as much authority over you and your education anymore so you COULD uncover this code. However, when you graduate, you will be a professional who follows a clear code of conduct in whatever discipline or industry you pursue. It would still not be right for you to set other students up for potential academic integrity violations at that point (Note that this has happened). I would ask that you continue to keep it set to private except for those moments in your career where you wish to show it off. At that point, it is your call.

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 or six separate C files, but in most cases, no more than ten to twelve

- 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.

- Allowed Libraries: ***sys/time.h***, ***math.h***, ***stdio.h***, ***stdlib.h***, ***pthread.h***, ***time.h***, and ***string.h***

- Also allowed: printf family functions, including ***printf***, ***fprintf***, ***sprintf***, ***snprintf***

- Disallowed functions: utility functions are any functions that start with "***str***" (e.g., ***strcpy***, ***strcat***, ***strtok***, etc.)

- Disallowed functions: 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, when specified in the instructions, 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.). 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 Sim02 – 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 their BBLearn grade rows; 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, which would be all the .c, .h, and make files

- In addition, a grading Rubric spreadsheet will be provided with each assignment; this must also be included in the tar/gz file, and must be named **Sim0x_GradingForm_{programmer secret ID}.xlsx** (e.g., **Sim0x_GradingForm_123456.xlsx**)

- Any extraneous files added such as unnecessary library, data files, or object files will be cause for credit reduction

- The file must be named ***Sim0X_<Secret ID>.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 ***Sim01_123456.tar.gz***

- The programs must be uploaded at or before 4:00 pm on the date for each specific programming project/phase, and at or before 4:00 pm for each grading component

- Dates are found previously in this document.

THE PROGRAM CONFIGURATION DATA

All programs must be able to input and store the contents of the file shown next. Note that any of the nine configuration lines may be in any order in the file. However, the "Start Simulator..." and the "End Simulator..." lines will be located at the beginning and end of the file as shown.

```
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 following items specify the expected and allowed data that may be used in the configuration file. Each has a specification of limits or conditions and if any configuration item is outside the specified limits, the uploading process must throw an error.

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: $0.0 \leq V/P \leq 10.0$

File Path: This line must contain the file path where the meta-data will be found. The assignment requirement is that the data must be in the same directory as the program

CPU Scheduling Code: This line will hold any of the following: ***FCFS-N***, ***SJF-N***, ***SRTF-P***, ***FCFS-P***, ***RR-P***. 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: $0 \leq Q \leq 100$

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 ignored although it must still be stored in the data structure. Specification: $1024 \leq MA \leq 102400$ (1 MB to 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. Specification: $1 \leq PCT \leq 1000$

I/O Cycle Time (msec): This line will also hold an integer cycle time like the processor cycle time. Specification: $1 \leq IOCT \leq 10{,}000$

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)

At the end of the configuration file, the last "End Simulator . . . " must be found in the configuration file exactly as shown above.

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.

THE PROGRAM META-DATA

The program meta-data components are as follows:

Commands: **sys**, **app**, **dev**, **cpu**, **mem**

In/Out arguments: **in**, **out**

First string argument (after In/Out where used): **start**, **end process**, **allocate**, **access**, **ethernet**, **hard drive**, **keyboard**, **monitor**, **serial**, **sound signal**, **usb**, **video signal**

First and/or second integer arguments: values as specified in configuration standards.

Sample meta-data

Start Program Meta-Data Code:
sys start
app start, 0
dev in, hard drive, 18
cpu process, 9
cpu process, 9
cpu process, 9
dev out, monitor, 60
app end
app start, 0
dev in, sound signal, 40
mem allocate, 2048, 4096
dev in, hard drive, 30
mem allocate, 2760, 2890
dev in, sound signal, 25

cpu process, 6
mem allocate, 3000, 4000
dev out, usb, 10
mem allocate, 3500, 3700
app end
app start, 0
dev in, video signal, 70
cpu process, 10
dev out, monitor, 70
dev in, hard drive, 18
cpu process, 9
app end
app start, 0
dev in, sound signal, 35
dev out, monitor, 100
dev in, keyboard, 50
cpu process, 9
dev out, video signal, 49
app end
app start, 0
dev in, keyboard, 90
dev out, sound signal, 40
dev out, serial, 32
cpu process, 10
dev in, hard drive, 15
app end
sys end
End Program Meta-Data Code.

GENERAL INFORMATION

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 (found in the configuration file), and it is supposed to run for 10 cycles (found in the meta-data file, 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. To repeat, the simulator must represent real time; if the operations take 10 seconds, the simulator must take 10 seconds.

SUPPORTING PROGRAM CODE

Timing the simulator operations:

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 Sim01 demonstration program, which will also be provided.

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 [-zz] config_y.cnf`

*x is the project number (1-4), y is the number of a given configuration file, and zz is one or more of the three specified command line switches

Note that the program MUST work in this form, and ONLY 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 (Sim01) – 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. It will also provide a display representing the running simulator.

While this is a stand-alone project, students are wise to assess the next three phases of 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.

IMPORTANT: As mentioned previously, no processing function should ever display an output. The configuration and meta-data input operations are a good example. If there is a failure in the operation/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.

MAIN FILE/MAIN METHOD/DRIVER

One file will contain the main file for the simulator program. As mentioned previously in this document, no other functions or operations should be in the main file except the ***main*** function itself, and most of this function's actions will be to call other functions to conduct the necessary operations. The ***main*** function should be developed to upload the configuration and meta-data files, and to conduct the simulation process before any other code is written; this is demonstrated in the tutorial related to this course. If implemented correctly, this function will not change throughout this project.

CONFIGURATION FILE

The configuration file must be uploaded to the system as called by the main function. Any issues with incorrect commands, in/out arguments, and string or other arguments including specified limits for the configuration items must stop the program and report the issue as specifically as possible.

META-DATA FILE

The meta-data file must also be uploaded to the system as called by the main function and again, any incorrect or misspelled commands or string arguments, or out of limit numerical arguments must stop the program, and report the issue as specifically as possible.

As specified above, students will be provided a meta-data file generation program. The meta-data acquisition component must upload ***any*** meta-data file of ***any*** size, ***any*** number of actions, ***any*** number of programs, etc., and all student programs must work correctly on any correctly formed meta-data file.

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. The Sim01 program will also provide a call to the function that runs the simulator. For purposes of this first assignment, the function will simple output "runSim called here" to the monitor to demonstrate that the main driver program is fully operational. Once this part of the assignment is completed, there should be no reason to go back and modify the **main** program.

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 in each project rubric 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.

SIM01 PROCESS/DISPLAY

The following command-line call using the switch **-dc** (display config) is provided here:

```
>OS_SimDriver_6 -dc config0.cnf

Simulator Program
=================

Config File Display
-------------------
Version                 : 1.05
Program file name       : testfile.mdf
CPU schedule selection  : SRTF-P
Quantum time            : 3
Memory Available        : 11100
Process cycle rate      : 10
I/O cycle rate          : 20
Log to selection        : Both
Log file name           : logfile_1.lgf


Simulator Program End.
```

The following command-line call using the switch **-dm** (display meta-data) is provided here:

```
>OS_SimDriver_6 -dm config0.cnf

Simulator Program
=================

Meta-Data File Display
----------------------

Op Code: /pid: 0/cmd: sys/io: NA
         /arg1: start/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: start/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: hard drive/arg 2: 18/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 9/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 9/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 9/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: out
         /arg1: monitor/arg 2: 60/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: end/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: start/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: sound signal/arg 2: 40/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: mem/io: NA
```

```
         /arg1: allocate/arg 2: 2048/arg 3: 4096/op end time:
0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: hard drive/arg 2: 30/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: mem/io: NA
         /arg1: allocate/arg 2: 2760/arg 3: 2890/op end time:
0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: sound signal/arg 2: 25/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 6/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: mem/io: NA
         /arg1: allocate/arg 2: 3000/arg 3: 4000/op end time:
0.000000

Op Code: /pid: 0/cmd: dev/io: out
         /arg1: usb/arg 2: 10/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: mem/io: NA
         /arg1: allocate/arg 2: 3500/arg 3: 3700/op end time:
0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: end/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: start/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: video signal/arg 2: 70/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 10/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: out
         /arg1: monitor/arg 2: 70/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: in
```

```
         /arg1: hard drive/arg 2: 18/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 9/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: end/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: start/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: sound signal/arg 2: 35/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: out
         /arg1: monitor/arg 2: 100/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: keyboard/arg 2: 50/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 9/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: out
         /arg1: video signal/arg 2: 49/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: end/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: start/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: in
         /arg1: keyboard/arg 2: 90/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: out
         /arg1: sound signal/arg 2: 40/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: dev/io: out
         /arg1: serial/arg 2: 32/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: cpu/io: NA
         /arg1: process/arg 2: 10/arg 3: 0/op end time: 0.000000
```

```
Op Code: /pid: 0/cmd: dev/io: in
         /arg1: hard drive/arg 2: 15/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: app/io: NA
         /arg1: end/arg 2: 0/arg 3: 0/op end time: 0.000000

Op Code: /pid: 0/cmd: sys/io: NA
         /arg1: end/arg 2: 0/arg 3: 0/op end time: 0.000000


Simulator Program End.
```

The following command-line call using the switch **-rs** (run simulator) is provided here:

```
>OS_SimDriver_6 -rs config0.cnf

Simulator Program
=================

runSim called here

Simulator Program End.
```

Finally, note that the three command-line switches may be called in any combination. For example, it might be called with both the **-dc** and **-dm** switches so that it displays both the configuration file and the meta-data file. It could also be called with all three switches (**-dc**, **-dm**, and **-rs**) and all three responses should be correctly displayed.

Phase 2 (Sim02) - Simple Multiple Program Batch 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 program must create a Process Control Block data structure for each and every process. There are no specific requirements for PCB data structures other than every process must have a unique process ID. Other data quantities in the PCB are at the discretion of the programmer

- the simulator must manage, process, and display the simulation of multiple programs with multiple operation commands in a batch or sequential form. The number of programs or 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

    o the selection of monitor, file, or both must be made in the configuration file

    o in addition, the output to file operation must all be conducted at one point, after the simulation has been completed. 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 given program may be run to observe the structure of the logfile

o also, note that under the condition of outputting to the file only, the simulator will not respond for a while during the time the simulation is running. You must show some kind of indication that the simulation is running so the user won't think the program has failed. You may display something like, "Simulator running for output to file only" but this display must only occur under that circumstance

- 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 in the order they were found in the meta-data file. At this point, any other scheduling algorithm must fail over to FCFS-N

- the simulator must show the time remaining when a process is placed into RUNNING mode; this will only happen once in this batch mode but will happen several times in Sim 04

- 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. However, all programs should be tested with a variety of metadata files, at least five to ten of them.

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

DESCRIPTION

This phase will offer you the opportunity to learn about memory management by creating your own software 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, compared, and displayed when the process is placed into the RUNNING state. 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. Finally, note that the metadata/processes must be loaded and numbered in the order they are provided, and from there, they may be scheduled as specified. Do not reorder the processes to satisfy the SJF scheduling.

- Also note that it is not permitted to create separate simulation operations for the separate scheduling. There must be only one simulation master loop that calls for and uses the appropriate scheduling strategy

- 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 paragraphs. 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. Part of the management process for holding this data includes a requirement to create a PCB for each process provided in the previous phase. There is no specific requirement for what is to be stored in each PCB but the data it holds must be specifically pertinent to the process it represents.
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. Up to 100 MB (102400 KB) may be specified in the configuration file

- the process will allocate a segment of memory using the **`mem allocate, 3000, 4000`** operation command provided previously.

- For an allocation request:

- the system displays the request attempt
  - e.g., for op code: **`mem allocate, 3000, 4000`**
  - **`Display: 0.000037, Process: 0, attempting mem allocate request`**

- the MMU must first check that the amount of memory requested is not larger than that specified in the configuration file (i.e., the base plus the offset is not greater than 102400 KB). For this example, the base is 3000 KB and the offset is 4000 KB meaning the request to allocate is between 3000 and 7000 KB inclusive

- the MMU must then check to see if the base plus the offset does not overlap with any established memory segment within the current process or any other process in the system

- Note that it will be important to manage and store the lower and upper limits of each memory request. However, there is no value in actually creating memory so this is not allowed. You must simply manage and monitor the limits provided

- if it does not overlap (success), then the system will:

    - Display the message:
        - **`0.000042, Process: 0, successful mem allocate request`**

    - Continue the process with the next op code

- if it does overlap (failure), then the system will:

    - Display the error message:
        - **`0.000042, Process: 0, failed mem allocate request`**

    - Stop the process with a segmentation fault message:
        - **`0.000045, OS: Process 0 experiences segmentation fault`**
        - **`0.000051, OS: Process 0 ended and set in EXIT state`**

    - Set a new process in the running state, and continue

Note that times are examples for one given test run

- For an access request:

- the system displays the request attempt
    - e.g., for op code: **`mem access, 3000, 4000`**
    - **`Display: 5.270116, Process: 0, attempting mem access request`**

- the MMU must check that the amount of memory to be accessed is within the limits of the specified process (inclusive, i.e., the memory access request could access all of the memory allocated to that process). For the given example, it would be acceptable to access the memory from base 3000, up to and including 7000 (allocated above), considering the offset of 4000

- if the memory access request is within the allocated memory for the specified process (success), then the system will:
    - Display the message:
        - **5.270120, Process: 0, successful mem access request**
    - Continue the process with the next op code
- if the memory access request is outside the allocated memory for the specified process (failure), then the system will:
    - Display the message:
        - **5.270132, Process: 0, failed mem access request**
    - Stop the process with a segmentation fault message:
        - **5.270145, OS: Process 0 experiences segmentation fault**
        - **5.270149, OS: Process 0 ended and set in EXIT state**
    - Continue the process with the next op code

- finally, 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, along with the display of the time remaining for that process
    - 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
    - if there are questions on any of the above conditions, the example program provided is the reference

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 interrupts) 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.

- threads may optionally be used for all I/O operations as needed; extra credit will be earned for correct use of threads. For any of the new strategies, which are FCFS-P, SRTF-P, and RR-P, the threads may 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.

- if threads are not used, simulated concurrency must still be represented and displayed as if threads were used

- 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 with their correct return times, and 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 are likely to 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