CSC3210 – Week 12 Project: Process Simulator
Project Details and Instructions

**Introduction**

A CPU on a computer system executes a stream of instructions. In doing so, it does not "know" what workload it is executing. While certain hardware mechanisms (e.g., system mode) can be utilized for isolation and protection, it is the operating system that is creating abstractions such as processes to simplify the concept of multiple processes all running at the same time. The operating system does the necessary bookkeeping and context switching adjust the CPU instruction stream to give the appearance of multiple processes executing at the same time. The algorithm used by the operating system for process scheduling is a source of overhead so using an efficient scheduler has direct effect to the overall efficiency of the computer system. In this assignment, your job will be to simulate a processes scheduler in an operating system using different parameters. Your simulation will need to also keep track of statistics that can be used to evaluate the efficiency of the scheduling implementation.
Work on the assignment is to be done ***individually***. You are welcome to collaborate with class members, but the project must be your own work. Your implementation must use the C programming language and utilize system calls and library functions as appropriate.

**Background and References**

For this assignment you will be implemented a process scheduling simulator.
As a review, it might be useful to recall the following scheduling policies:

- Preemption
  - Non-preemptive - process are allowed to run to completion
  - Preempting - operating system interrupts processes that aren't finished to allow another process to run
- Waiting Process Ordering
  - Ordered by arrival time
  - Ordered by CPU time needed
  - Ordered by priority

Selecting different policies results in several algorithms (among others):

- First come, first served (FCFS)
- Shortest job next (SJN) / Shortest job first (SJF)
- Round Robin
- Shortest remaining time (SRT)

**Project Description**

While it can be proven that that most optimal scheduling algorithm in terms of reducing wait time is shortest remaining time (SRT), however it is often impractical or even impossible to know how much CPU time a process needs. For this assignment, we will assume that processes do not know how much CPU time they need to complete, but their execution will be set by various events.
Your goal will be to simulate the execution of one or more process while recording statistics about the run time for each process and the system as a whole.

- The system consists of a single CPU. The CPU can execute only a single instruction stream at a time.
- The system has 10 I/O devices identified by their I/O device ID - the ID has values 0 through 9 (inclusive)
- Processes will start in the system by specifying their priority
    - When a process starts it is set to be ready to run
    - Only ready processes can be executed on the CPU
    - When a process is selected to run, the instructions for its text are set to execute on the CPU
    - During a processes execution it may make a request for input/output (I/O). Doing so makes the process not ready to run. It must be placed on a waiting queue and another ready process must be selected to run.
- A process priority is a number between 0 and 9 (inclusive)
    - A higher number is a higher priority
    - e.g. 0 is the lowest priority and 1 is a lower priority than 8.
- The scheduling policy must follow the rule that the highest priority process that is ready to run gets to run next.
    - A larger number indicates higher priority (e.g. 10 is a higher priority than 1)
- An additional parameter will specify whether the scheduling policy is preemptive or not

Given these constraints a process can be in one of 3 states:

- Running - running on the CPU
- Ready - able to run on the CPU but currently not running
- Blocked - waiting for I/O on a resource

**Development Requirements**

Your program must simulate the system until all processes have completed. Simulation parameters will be given to you in a text file. The path to this file will be given on the ***command line***. For example:

**./procsim sample1_input.txt**

This file will define additional scheduling policies, when processes start, when processes request I/O, when I/O operations complete, and when processes end.
The format for the text file will be as follows:

- The first line consists of a single number either 0 or 1
    - 0 indicates that the system does ***NOT*** use preemptive scheduling
    - 1 indicates that the system ***does*** use preemptive scheduling
- The next lines indicate events for the processes in the system
    - Each line will start with a single number indicating the time which the event occurred
    - The next number in the line indicate the event type

- The rest of the line consists of additional numbers specific to the event

***NOTE:*** All numbers in the input file will be integers > 0 and separated by spaces in the lines
The following table describes the events:

| Operation Type | Operation | Description | Additional Values |
|---|---|---|---|
| 1 | Process Start | Start a process | The process priority |
| 2 | I/O Request | The current running process requests an I/O operation | Identifier of the device |
| 3 | I/O End | The request for an I/O operation completed | Identifier of the device |
| 4 | Process End | The current running process ends | None |

- Times are specified in terms of unit-less time values. The system starts at time zero (0) and ends when all processes have ended.
- You can assume that the events in the file are coherent
  - Numbers in the parameter file will always be integers
  - Events will never be out of order
  - The time of events will be distinct - no two events will ever happen at the same time
  - A process will never end if it is not running
  - An I/O request for a resource will never end before it is requested
  - I/O will never be requested if no processes are able to run
  - A process will never have a priority outside the valid range
    - A process will never have a negative priority - 0 is the lowest priority
    - A process will never have a priority greater than 9
  - A process will never wait for an I/O device outside the ID range
    - A process will never wait on a negative I/O device
    - A process will never wait on a device greater than 9
- There may be a duration of time in which there are no processes eligible to run. In this case, the system is idle and no processes are executed.
- If the system is preempting, the currently running process is removed from the CPU (and queued) if another ***higher priority*** process becomes ready to run. The higher priority process is then allowed to run on the CPU.
  - If a process with the ***same or less*** priority as the currently executing process becomes ready, the current process continues to run.

- If the system is non-preempting, the currently running process continues to run, even if a higher priority process becomes ready, until the running process exits or requests I/O

***NOTE:*** More than one process may wait on the same I/O device

- If this happens, when the I/O request completes, ***ALL*** waiting processes will be set to ready
- Processes become ready must be ordered in the order of highest priority first
- If processes have the same priority, the processes should be ordered in order of the time they requested the I/O device (earliest first).

Here is an example parameter file and execution explanation:

**0**
**2 1 1**
**3 1 2**
**4 2 1**
**6 4**
**7 3 1**
**10 4**

- The system starts at time 0 and is ***NOT*** using preemption
- At time 2, a process with priority 1 enters the system - P1
- At time 3, a process with priority 2 enters the system - P2
    - Because the system is not preempting process 1 continues to run
- At time 4, the currently running process (P1) requests I/O for device 1
    - Process P2 is ready run so it starts running
- At time 6, the currently running process (P2) ends
    - Process P1 is still waiting for I/O on device 1 so the system is idle
- At time 7, the I/O for device 1 ends
- At time 10, the currently running process (P1) ends
- There are no more processes so the system ends

**Queuing**

You are required to implement and use queues to organize the processes that are not running. The exact number of queues you use and the specific implementation for the queues is up to you.

**Statistics Tracking**

As each process start, the system ***must*** assign it a process identifier starting at 1 and incrementing by 1 for each new process that starts. Do ***NOT*** reuse processes identifiers. The parameter file will never simulate more processes than will fit in an integer.

At the end of the simulation you ***must*** print the following:

- For each process:
  - The ready wait time - the amount of time a process waits when it is ***READY*** to run
  - Do ***NOT*** include the time a process waits for I/O in this value
  - The I/O wait time - the amount of time a process waits for an I/O device
- The total time the system was idle

***NOTE:*** Since you are using multiple processes or threads in this assignment, the output will be deterministic. In other words, the output for a given input file should always be the same for every run of your program.

**Output Requirements**

For the above example, here is a sample output:

**Simulation started: Preemption: False**
**2: Staring process with PID: 1 PRIORITY: 1**
**2: Process scheduled to run with PID: 1 PRIORITY: 1**
**3: Staring process with PID: 2 PRIORITY: 2**
**4: Process with PID: 1 waiting for I/O device 1**
**4: Process scheduled to run with PID: 2 PRIORITY: 2**
**6: Ending process with PID: 2**
**7: I/O completed for I/O device 1**
**7: Process scheduled to run with PID: 1 PRIORITY: 1**
**10: Ending process with PID: 1**

**Simulation ended at time: 10**
**System idle time: 3**

**Process Information:**
**PID: 2, PRIORITY: 2, READY WAIT TIME: 1, I/O WAIT TIME: 0**
**PID: 1, PRIORITY: 1, READY WAIT TIME: 0, I/O WAIT TIME: 3**

Information for each event ***and*** each context switch is printed. The value before the colon (:) indicates the event time for each action.

***NOTE:*** Your program should output the same information. The exact formatting is not necessary

- Output should include current event time and an explanation of each event
- A status message for each context switch indicating the PID and priority of the new executing process

- A summary of the simulation including:
    - System idle time
    - Process ready wait time and I/O wait time for ***each*** process - Process can be printed in any order

**Additional Requirements**

- Do ***NOT*** include source files inside each other via #include. The #include preprocessor macro should only be used for header files
- You ***must*** put all source code in a src directory.
- You ***must*** submit a make file in your src directory that will build ***ALL*** executables by just typing make on the command line
- Research make files on how to correctly create more than one executable from a single make file.

**Code Structure**

Code must follow style guidelines as defined in the course material.

**Getting Started**

The following files have been provided for you:

- src/main.c - source code for the main routine
- samples/sample1_input.txt - sample input file
- samples/sample1_output.txt - output for sample1_input.txt - sample given above
- samples/sample2_input.txt - sample input file
- samples/sample2_output.txt - output for sample2_input.txt - multiple processes with different priorities and a preemptive scheduler
- samples/sample3_input.txt - sample input file
- samples/sample3_output.txt - output for sample3_input.txt - multiple processes with different priorities, a preemptive scheduler, with I/O waiting

Feel free to create additional functions and files as needed to run your simulation. Consider dividing up code into separate files. Use header files as needed. Do ***NOT*** forget to use include guards in your header files.
At the top of ***EACH SOURCE FILE*** include a comment block with your name, assignment name, and section number.

***SUBMISSION REQUIREMENTS:*** You must submit with your source code a Makefile which can be used to build your program. Your program must compile without error ***AND*** without warnings. Your Makefile must create a single executable called procsim

**Hints and Tips**

While you are required to use queues in this assignment, the number and implementation is up to you. Just remember that:

- Processes have priority and the highest priority process that is ready to run should run on the CPU next
- Process may wait for an I/O device - however a process can only wait for one I/O device at a time
- Multiple process may wait on the same I/O device - when that I/O device finishes the I/O request ***ALL*** processes waiting on the device become ready to run
    - Processes waiting on an I/O device may have different priority
- There is a limit to the number of priorities on the system
- There is a limit to the number of I/O devices on the system

**Reading in the simulation parameters**

- You are required to read in the parameters via an input file as described above. The name of the file will be provided on the ***command line***.
- You can use this fscanf to read in all the numbers you need from the file. It takes care of all the white space for you. So, to read your file you can start with:

```
#include <stdio.h>

int main(int argc, char* argv[]) {

   FILE* input = fopen(argv[1], "r");

   // Read in the simulation parameters
   int preemptive_scheduler;
   fscanf(input, "%d", &preemptive_scheduler);

   // Read each event
   int event_time;
   int operation_code;
   int priority;
   int io_device;

   // NOTE: fscanf will return a number other than 2 when the end of file is reached
   //       this can be used to indicate that there are no more events
   while(fscanf(input, "%d %d", &event_time, &operation_code) == 2) {
      switch(operation_code) {
```

```
        case 1:
          // Handle process start
          fscanf(input, "%d", &priority);
          break;
        case 2:
          // Handle I/O request
          fscanf(input, "%d", &io_device);
          break;
        case 3:
          // Handle I/O request complete
          fscanf(input, "%d", &io_device);
          break;
        case 4:
          // Process end
          break;
      }
    }

    fclose(input);
}
```

***NOTE:*** The events for the simulation end when the input file contains no data. Using fscanf it is easy to check for this since the return value from fscanf is "number of successfully matched and assigned input items". To read in an event, the event will ***ALWAYS*** start with 2 integers (the event time, and the operation code). When there are no more events in the file, fscanf won't be able to read these two integers and will return something other than 2.

***NOTE:*** the code above reads in additional values depending on the operation. You will need to use these values when handling the event.

**Testing and Debugging**

Feel free to use the samples included in your repository to get you started with testing. Creating a file with simpler parameters (fewer process) might help you debug.

CSC3210 – Week 12 Project: Process Simulator
Project Details and Instructions

***SUBMISSION REQUIREMENT:***

- You must fully test your simulator. As part of your submission, you must create test cases of your own simulation parameter files to show that you adequately tested your program. Don't forget to test small configuration, large configuration, and boundary cases.
- With each test case include documentation describing what it is testing.

**Debugging**
See the course debugging tips for using gdb and valgrind to help with debugging.
Your program must be free of run-time errors and memory leaks.

**Deliverables**
When you are ready to submit your assignment:

- Make sure your name, assignment name, and section number are contained in all files in your submission - in comment block of source file(s)
- Make sure all source and header files are in your src directory ***ONLY***.
- Make sure you cite your sources.
- Make sure your assignment code is commented thoroughly and follows the coding standard.
- Include in your submission, a set of suggestions for improvement and/or what you enjoyed about this assignment.
- Make sure all files are zipped and uploaded to Canvas

**Additional Submission Notes**
If/when using resources from material outside what was presented in class (e.g., Google search, Stack Overflow, etc.) document the resource used in your submission. Include exact URLs for web pages where appropriate.

***NOTE:*** Sources that are not original research and/or unreliable sources are not to be used. For example:

- Wikipedia is not a reliable source, nor does it present original research: https://en.wikipedia.org/wiki/Wikipedia:Wikipedia_is_not_a_reliable_source
- ChatGPT is not a reliable source: https://thecodebytes.com/is-chatgpt-reliable-heres-why-its-not/

You should ensure that your program compiles without warnings (using -Wall) or errors prior to submitting. Make sure your code is well documented (inline comments for complex code as well as comment blocks on functions). Make sure you check your code against the style guidelines.

**Grading Criteria**

- 5 Points) Submitted files follow submission guidelines
  - Only the requested files were submitted
  - Files are contain name, assignment, section
  - Sources outside of course material are cited
- (5 Points) Suggestions
  - List of suggestions for improvement and/or what you enjoyed about this assignment
- (10 Points) Code standard
  - Code compiles without errors or warnings
  - Code is formatted and commented per the course coding standard
- (5 Points) Memory management
  - Code contains no memory leaks
- (15 Points) Submitted test cases are documented and adequately test your program
- (60 Points) Program passes instructors test cases