Assignment 02

Due: Beginning of the class, **Oct 15th**

Late due: Beginning of the class, **Oct 17th**

You must work on this assignment on your own.

Part II Programming: **Scheduling** (80 points)

Task

Assuming a **uniprocessor** environment, write a simulator for short-term process scheduling using the **Shortest Job Next** (SJN) strategy. The scheduling simulation **must** be carried out in a multi-threaded fashion (see **threading** section below).

Important:

- A **single-threaded** implementation will incur a **50% penalty** of the total assignment grade.
- Any hard hardcoding to generate expected output without implementing the required functions will result in a **ZERO** grade.
- We will investigate the similarity of your code against all sources we can find online:
  - You may **NOT** use any existing code for scheduling from online such as github, chatgpt, etc.. Any use of those would result in an **automatic ZERO** for the assignment and will be reported as plagiarism.
  - And as mentioned in the syllabus, posting the assignment, and asking for help on any online platform is considered plagiarism that would result in a **ZERO** for the assignment and academic dishonesty reporting.

Scheduling algorithm

The scheduling algorithm you need to implement is **Shortest Job Next** (SJN) with **exponential averaging estimation** for CPU bursts.

For simplicity:

- **Simulate the non-preemptive** version of the shortest job next algorithm, i.e., once the execution of the current CPU burst starts, it will always complete before determining and executing the next CPU burst.
- **Important**: Each process is accessing a **separate** I/O, when the process enters the next I/O burst, it will execute it immediately, this implies:
  - During the CPU burst (execution) of a process, **all blocked processes** due to I/O would be executing their respective current I/O burst **at the same time**.
- Assume context switch time and other overheads are comparatively negligible.

Note:

- The **turnaround** (completion) time of a process is the duration from the moment of the process admitted to the system through the completion of the last CPU burst of the process.
- Using a three process-state (Running, Blocked, Ready) model, the turnaround time is:
  - **Turnaround time = CPU bursts + I/O bursts + Wait time**

- Note:
  - **CPU** bursts are the time each process spends in a **Running** process state.
  - **I/O** bursts are the time each process spends in a **Blocked** process state.
  - The **wait time** is the wait time each process spends in a **Ready** process state**.**

Exponential averaging for estimating / predicting the next CPU burst

Use the exponential averaging formula below for estimating CPU bursts for the Shortest Job Next scheduling:

$$p(n) = \begin{cases} \mu_{CPU\ burst} & n = 1 \\ \alpha \cdot b(n-1) + (1-\alpha) \cdot p(n-1) & n > 1 \end{cases}$$

- Weight ***alpha*** $\alpha \in (0,1)$
- $p(n)$ – prediction of nth CPU burst
- $\mu_{CPU\ burst}$ - average CPU burst
- $b(n)$ – nth CPU burst

Input

The input file has the process bursts with the following format (below is just an example, test files could be having more (or less) number of processes and more (or less) number of bursts for each process:

4 7 6
3 8 3 1 6
5 2 3

The three lines mean there are three processes **P0, P1, P2** admitted to the system at **the same time in that order**. It may have an arbitrary number of processes. Each line has a series of **actual** burst times for a process (separated by a single white space), **alternating between CPU bursts and IO bursts**, with each burst being a positive integer that represents the length of the burst (e.g., in milliseconds). The bursts of each line (for a process) **must start** with a **CPU** burst, followed by an **I/O** burst, followed by a **CPU** burst, so on and so forth, the series **must end** with a **CPU** burst, which means each line must have an **odd** number of bursts. So, the above example input can be **interpreted** as:

P0: 4ms (CPU), 7ms (I/O), 6ms (CPU)
P1: 3ms (CPU), 8ms (I/O), 3ms (CPU), 1ms (I/O), 6ms (CPU)
P2: 5ms (CPU), 2ms (I/O), 3ms (CPU)

With **exponential averaging** for estimating **CPU** bursts, and using an ***alpha*** **0.5** as **an example**, the estimated CPU bursts will be (refer to the formula above and the lecture slides):

P0: **5.00ms** (CPU), 7.00ms (I/O), **4.50ms** (CPU)
P1: **4.00ms** (CPU), 8.00ms (I/O), **3.50ms** (CPU), 1.00ms (I/O), **3.25ms** (CPU)
P2: **4.00ms** (CPU), 2.00ms (I/O), **4.50ms** (CPU)

Note for simplicity, the CPU burst **average** is calculated by averaging the CPU bursts in each row.

Important:

- If alpha is specified from command line (see below), you **must** use the **estimated CPU bursts** of the processes in the **Ready queue to determine which next CPU burst** is to be executed and **use** the **actual burst** time for simulating how long the burst will be executed.

Output

- All **outputs** should be printed to the **standard output**. Do not print to a file.
  - Use the **given log functions** from the **log.h** and **log.c (or change it to log.cpp for C++)** source code to ensure the format of the output is good for autograding.
- **Before simulation** starts:
  - Print the **process bursts read from the input file**.
  - See the given **sample** output for examples.
- **During** the simulation:
  - **Whenever** some **cpu** burst is executed for a process, print to the standard output with the following information:
    - P0: executed cpu bursts = 4, executed io bursts = 0, time elapsed = 8, enter io
      - This means that process P0 was just executed for some cpu burst time, the **executed cpu bursts so far** for the process is 4, the **executed io bursts so far** is 0, **the total time elapsed** since the beginning of the scheduling is 8, and the P0 cpu execution **stops** as it is entering an I/O burst.
      - More examples:
        - P1: executed cpu bursts = 4, executed io bursts = 7, time elapsed = 15, completed
          - P1 was just executed for an amount of cpu burst and it stopped because P1 has completed executing all its bursts.
      - See the given sample output for more examples.
- **After** the **simulation ends**:
  - Print the scheduling stats (**turnaround time and wait time**) of the processes in the **order of their completion** time. See above for the calculation of wait time based on turnaround time.
  - If the alpha argument is specified, print the **process bursts with estimated CPU bursts** using **exponential averaging** in the **order of the process completion** time. If alpha is not specified, do not print this part.
  - See the given **sample** output for examples.
- See the sample output (**expectedoutput_alpha_0.5**.txt) from executing the following simulation:
  - ./schedule bursts.txt -a 0.5
    - See user interface below.
    - This SJN simulation uses bursts.txt input and exponential averaging using an ***alpha*** of 0.5.

Threading

Your program starts from the **main thread** (where the main method is). The main thread will:

- read and process command line arguments,
- read the process bursts from a file,
- creates and initializes a data structure to be shared between the threads, then spawns a scheduler worker thread and passes the process bursts and other scheduling related information to the scheduler thread.

The **scheduler thread** uses the bursts to simulate the execution of the processes and prints the execution sequence of the process CPU bursts and scheduling stats upon completion (see above).

The main thread must wait for the scheduler thread to complete the scheduling simulation, then exits. Do **NOT use the pthread_join** to have main thread wait for the scheduler worker thread; instead, use **busy waiting** (such as a boolean variable shared between threads) to have main thread wait for the scheduler thread. Do NOT use semaphores or monitors as they have not been taught in the class, you will practice the programming of those in a later assignment.

You will use the **POSIX pthread** (Portable Operating System Interface pthread) for creating and running the scheduler thread.

User interface

The executable uses **one mandatory** command line argument and accepts **one optional** argument. For parsing command line arguments, refer to "How can I process command line arguments?" in the FAQ link (C/C++ programming reference page).

Mandatory arguments:

- The **mandatory argument** specifies the file path to a text file that contains the **process bursts**, see the "Input" section below. Appropriate **error handling** should be present for the following cases:
  - If the file is not existent or cannot be opened, print to stdout the following error message (suppose the file name is bursts.txt) and **exit**:
    Unable to open <<bursts.txt>>
  - When reading and parsing the process bursts from the file
    - If a burst number is less than or equal to 0, print to stdout the following error message and **exit**:
      A burst number must be bigger than 0
    - If the number of bursts on a line is not of an odd number, print to stdout the following error message and **exit**:
      There must be an odd number of bursts for each process

Optional arguments:

-a N *Alpha* for exponential averaging, a float number, see above
Error handling:
If an out-of-range number (<= 0 or >= 1) is specified, program should print to the standard output:

Alpha for exponential averaging must be within (0.0, 1.0)
then exit.

Default: If **alpha is not specified**, do **NOT use the exponential averaging** for CPU burst estimation, just use the **actual bursts** from the input file for the SJN scheduling.

Important:

If you use **exit(..) function to exit** due to any argument error specified above, use an argument 0 to the exit function (#define NORMAL_EXIT 0), like:

exit(NORMAL_EXIT);

Also, **optional** arguments are processed **before mandatory** arguments, see FAQ for code (**getopt**) examples of processing command line arguments.

Examples:

./**schedule** bursts.txt -a 0.2

- Using process bursts from bursts.txt file for SJN scheduling:
  - Use exponential averaging with an ***alpha*** 0.2 for CPU burst estimations.
  - Use the **estimated** CPU bursts to determine which next CPU burst to execute.
  - Use the **CPU bursts** from the input file for simulating how long each CPU burst will be executed.

./**schedule** bursts.txt

- Using process bursts from bursts.txt file for SJN scheduling:
  - Since the alpha is not specified, **NO exponential** averaging is used for CPU burst estimations.
  - Use the CPU **bursts** from the input file to determine which next CPU burst to execute.
  - Use the CPU **bursts** from the input file for simulating how long each CPU burst will be executed.

Scheduling simulation implementation

The proposed implementation below illustrates a general idea of how to implement scheduling algorithms.

- Watch the **animation video** (posted with the assignment) to see an example of scheduling sequence of actions based on the proposed implementation.
- You may use your own approach for implementing the algorithms other than the proposed one.
- You may use the STL (standard template libraries) classes and the C standard library.

With the assumptions of a **uniprocessor** executing environment and I/O bursts accessing **separate** IOs, **scheduling** can be **simulated by maintaining two queues** of processes:

- one **queue** tracking the processes in the **ready** state, **stable sorted** based on the **next CPU** burst (estimated or actual) of the ready processes, and
- another **queue** tracking processes in the **blocked** state (due to I/O).

    - This **blocked** queue needs to be **stable sorted** based on the **next I/O** burst of the blocked processes. (see below)
- You may also want to use another collection to store the completed processes, in order of when they were completed, this is for printing at the end of the scheduling.

For **each process**, track the following information:

- A process ID: use the index of its burst line in the input file. For example, the first line of bursts is for Process 0, process ID would be 0.
- The remaining bursts of the process.
- The executed CPU bursts so far during the scheduling process.
- The executed I/O bursts so far during the scheduling process.
- The turnaround (or completion) time of the process.

Upon start, all processes are put in the **Ready** queue. At the beginning, the front of the queue has the first process admitted to the system. The scheduler should keep checking the status of the **Ready** queue and **Blocked** queue:

- With Shortest Job Next strategy, every time when **picking the next process** in the Ready queue to execute, **stable sort** the **Ready queue** based on the **next CPU** burst (estimated or actual, in ascending order) of the ready processes. If the **Ready queue is not empty**, pick the front process in the Ready queue to execute. Simulate the CPU burst execution with the following logic in the given order:
    - Increment the **executed CPU bursts** (so far) for the process by the amount of current CPU burst.
    - Increment the **total elapsed time** by the amount of currently executed CPU burst, this is needed for tracking the completion time for each process.
    - During the execution of the current CPU burst of the executing process, **all** blocked processes in the **Blocked** queue should execute their respective current I/O bursts at the same time (see above). Iterate through the blocked queue, for each blocked process:
        - Check if its current IO burst is done during the current CPU burst of the executing process, **if so**, **move the blocked process to the end of Ready queue** (you may choose to stable sort the Ready queue by next CPU burst at this moment); otherwise, update its current IO burst to (current IO burst – currently executed CPU burst by executing process)
            - The blocked queue needs to be **stable sorted** in the **ascending order** of the first remaining I/O burst of the blocked processes to ensure the proper order is maintained when multiple blocked processes are moved from **Blocked** queue to **Ready** queue during executing the current CPU burst of the executing process, where the blocked process with the shortest current I/O burst should be moved first to the **Ready** queue.
        - increment its executed I/O bursts so far (for printing, see output above)
    - The CPU burst execution will **stop** due to one of the following reasons:
        - Finishing the current CPU burst and entering the next I/O burst.
            - In this case, move the executing process to the **end** of **Blocked** queue, and stable sort the Blocked queue in the **ascending order** of the first remaining I/O burst of the blocked processes.
        - Finishing the current CPU burst and all bursts are completed for the process.

        - In this case, append the process to the completed processes.

- If the **Ready** queue is **empty** but the **Blocked** queue is **not empty**:
    - Iterate through the blocked queue, for each blocked process:
        - execute its current IO burst for the **front IO burst of the front process** in the Blocked queue,
        - update its executed I/O bursts so far, and
        - move the process to the end of Ready queue if the current IO burst finishes.
    - **Increment** the **total elapsed time** by the **front IO burst of the front process** in Blocked queue, after which the Ready queue will have at least one more process to execute.

- If the **Ready** queue is **empty** and the **Blocked** queue is **empty**:
    - Simulation is done!

Compilation and execution

- You must create and submit a Makefile for compiling your source code. Refer to the Programming page in Canvas for makefile help.
- The make file must create the **executable** with a name as **schedule.**
- Make sure to compile, run and debug your code in a **Linux environment**, because:
    - The C **getopt** for parsing command line arguments may not work in a non-Linux environment, such as Windows or MacOS.
- When using POSIX pthread, you need to use **-pthread or -lpthread to link the pthread library** to your program. Otherwise, your program may not compile or may not work although it may compile without warnings.
- You are strongly recommended to set up your local development environment under a Linux environment (e.g., Ubuntu 20.04 or 22.04), develop and test your code there first. You could also port your code to Edoras (e.g., filezilla or winscp) to compile and test to verify. The gradescope autograder will use a similar environment to Edoras to compile and autograde your code.

Programming references

Please refer to Canvas Module “Programming Resources” for coding help, particularly:

- Process command line arguments using **getopt**, refer to “How can I process command line arguments?” in FAQ.
- Create and use POSIX pthreads, and how to link the pthread library for compilation:
    a. You must include the header file **pthread**.h.
    b. Please note in most Linux systems, threads created from pthread_create are usually called light weight processes (LWP), each LWP is often mapped to a kernel-level thread, i.e., 1 to 1 mapping / multiplexing.
- C / C++ programming and practices.

Turning Into Gradescope

Make sure that all files mentioned below (Source code files and Makefile) contain your name and Red ID! Submit all files separately:

- Source code files (.h, .hpp, .cpp, .c, .C or .cc files, etc.)

- Makefile
- Do NOT submit any .o files or test files.
- **Do NOT compress / zip** files into a ZIP file and submit.
- **Do NOT put your files** into **a folder** and submit.
- Single Programmer Affidavit with your name and RED ID as signature, use **Single Programmer Affidavit.docx** from Canvas Module Assignments.
- **Number of submissions:**
    a. Please note the autograder submission count when submitting on Gradescope. For this assignment, you are **limited to a maximum of 10 submissions.** As stressed in the class, for testing, it is your responsibility as a programmer to sort out test cases based on the requirement specifications and perform those testing in your dev environment instead of relying on the autograder testing.

Grading

Passing 100% autograding may NOT give you a perfect score on the assignment. The structure, coding style, and commenting will also be part of the rubrics for the final grade (**see Syllabus Course Design - assignments**). Your code shall follow industry best practices:

- Be sure to comment on your code appropriately. Code with no or minimal comments are automatically lowered to one grade category.
- Design and implement clean interfaces between modules.
- Have proper code structure between .h and .c / .cpp files.
    - You are **required** to **separate** the logic of the **scheduler** and the **main** program, and you are encouraged to use more code separation if you see your program becoming unwieldy.
- Use meaningful variable names.
- NO global variables.
- NO hard code – Magic numbers, etc.
- Part of test data including one correct program output are given to you.

Academic honesty

Posting this assignment to any online learning platform and asking for help is considered academic dishonesty and will be reported.

An automated program structure comparison algorithm will be used to detect code plagiarism.

- Plagiarism detection generates similarity reports of your code with your peers as well as from online sources. It would be purely based on similarity check, two submissions being like each other could be due to both copied from the same source, whichever that source is, even the two students did NOT copy each other.
- We will also include solutions from the popular learning platforms (such as Chegg, github, etc.) as well as code from **ChatGPT** (see below) as part of the online sources used for plagiarism similarity detection. Note not only the plagiarism detection check for matching content, but also it checks the structure of your code.
- **If plagiarism is found in your code**, your grade for this assignment will be automatically zero, and an academic dishonesty report will be submitted.
- Note any code given to you as part of the prompt would be ignored in the plagiarism check.

The Center for Teaching & Learning and Instructional Technology Services also shared information concerning the rapidly evolving impact of artificial intelligence (AI) within academia — e.g., ChatGPT, etc.

SDSU's Center for Student Rights and Responsibilities have officially added plagiarism policies of using AI generated content for academic work, as put below:

- "Use of ChatGPT or similar entities [to represent human-authored work] is considered academic dishonesty and is a violation of the Student Code of Conduct. Students who utilize this technology will be referred to the Center for Student Rights and Responsibilities and will face student conduct consequences up to, and including, suspension."