# ­­­CPU Scheduling in Single Core and Multi-Core Systems

(Individual or Group Project)

Assigned: March 28, 2023

Due: April 17, 2023 (Hard deadline)

This project is worth 10% of the overall weight for this course. It is graded on a rubric out of 165 points.

### Learning Outcomes and Objectives

|  |
| --- |
| After successful completion of this assignment, students will be able to:   1. Understand and simulate an operating system’s (OS’s) scheduling mechanism. 2. Implement CPU scheduling algorithms for single core and multi-core systems. 3. Use and improve their critical thinking, problem solving, and software development skills to transform given project specifications into an implementation using threads. 4. Write robust code and test their solution to ensure resilience against improper inputs. 5. Communicate with the instructor and teaching assistants to seek a greater understanding and clarification on topics they need support with. |

### ­­­­Choice of Implementation

|  |
| --- |
| This project can be implemented in either standalone Java or POSIX threads (pthreads) with C/C++. Both options are supported with documentation provided on Moodle. This includes some commented source code, library tutorials, and supportive videos. This project specification is written with reference to both implementation choices, and relevant hints are provided, where applicable. |

### Student Outcomes Assessed for ABET Accreditation

| Student Outcomes (SOs) Assessed |
| --- |
| SO 1. Analyze a complex computing problem and to apply principles of computing and other relevant disciplines to identify solutions. |
| SO 5. Function effectively as a member or leader of a team engaged in activities appropriate to the program’s discipline. |
| SO 6. Be proficient in more than one programming language on more than one computing platform. |

### Detailed Instructions

This project will help you understand CPU scheduling, the effects of multi-core systems on scheduling performance, and further enforce multi-threaded synchronization through the following tasks:

1. Implementation of first-come, first-served (FCFS); round-robin (RR); non-preemptive shortest job first (NSJF); and preemptive shortest job first (PSJF) algorithms for a single-core processor
2. Implementation of FCFS, RR, and NSJF for a multi-core processor
3. Command Line
4. Report

This document will give a detailed specification of the above tasks. In addition, it will provide design and implementation hints as preparation for further work.

Note: This project will utilize only *one* ready queue for both single and multi-core systems. The scheduler is the part of an OS that decides which task from the ready queue gets to run on a core (or CPU) next. A scheduler uses an algorithm such as FCFS, RR, or SJF (among others) to make its decision.

## Task 1: Implementation of FCFS, RR, NSJF, and PSJF for a single-core CPU

For this task, you will create a simulation of a single-core CPU and scheduler. You will be implementing the following scheduler algorithms: first-come, first-served (FCFS); round-robin (RR); non-preemptive shortest job first (NSJF); and preemptive shortest job first (PSJF). You must create a dispatcher that will, depending on the selected algorithm, select a task from a ready queue and allow it to run on the CPU.

You need to create a total of **T** task threads, fork them, and add them to a ready queue. Each task should be modeled by a thread that runs for a total CPU burst time of **B** cycles before exiting. Each thread should implement a loop, where each iteration of the loop represents one CPU burst cycle. After populating the ready queue, fork a dispatcher thread which selects a task thread from the ready queue and allows it to run on the CPU for a specified time. For each run, your program should dynamically generate a value of **T** in the range [1,25] and a value of **B** in the range [1,50].

The dispatcher should select a task from the ready queue, enable it to run for a specified number of clock cycles, and then go to sleep until the thread has completed its specified burst time. At this point, the dispatcher should wake up and repeat the process until all tasks have finished all their bursts. After all of the tasks are finished, the main thread will then end the program. The task selection and burst time will depend on the scheduling algorithm given by command line input (see task 3). The CPU must be represented as a separate function from the dispatcher, which starts running the thread and then updates the current burst time and allotted burst time for the currently running thread. The CPU may be implemented as a separate thread from the dispatcher, but it is also acceptable if you implement the CPU as a method that is run on the same thread as the dispatcher (as long as the CPU implementation is clearly separate from the dispatcher itself).

The ready queue, along with any other resources shared between the dispatcher and the task threads, must be protected by semaphores or locks. Your dispatcher should be able to use the following algorithms for selecting a task from the ready queue: FCFS, RR, NSJF, and PSJF.

For PSJF, your tasks should randomly arrive at the ready queue. Specifically, you must have tasks arriving **after** threads have already started running on the CPU. Each arriving task should preempt the currently running task if its burst time is shorter than the burst time of the task that is currently running.

Regardless of the scheduling algorithm being used, all task threads should use the same implementation (and therefore run with the same thread function).

As this project is a simulation of scheduling, your program must print output statements denoting important events, including: the creation of new threads, the contents of the ready queue, the dispatcher algorithm in use, the task being selected each time the dispatcher runs, and the burst time that each task gets to execute for.

### Report

As part of your report, answer the following question about Task 1:

1. Have your program create 5 threads with burst times of 18, 7, 25, 42, and 21. Then, run your program with each of the four scheduling algorithms (for RR, use a time quantum of 5). Which scheduling algorithm was fastest? Provide a table, with columns for algorithm and runtime. (Remember: revert to random task creation and burst times before you submit the project)

## Task 2: Multi-core CPU implementation

For this task, you will need to implement a simulation of a multi-core CPU system. It will be like task 1, in that you’ll need to create and fork a random number of tasks, but this time, there will be four CPUs on which the tasks can run. You will have one dispatcher thread per core to select which task to run.

After populating the ready queue, fork one dispatcher thread for each CPU. Each dispatcher selects a task and allows it to run on that specific CPU. The number of cores will be given by command line input. See the task 3 description for more details. These CPUs should be able to run concurrently so that no one CPU prevents the others from getting a task from the ready queue and running it.

As the ready queue will be shared between dispatchers and cores, you must protect it with a semaphore or lock, along with any other shared resources. Your dispatcher should be able to use the first-come, first-served; round-robin, and non-preemptive shortest job first algorithms for selecting a task from the ready queue. You do not need to implement the preemptive shortest job first algorithm for this task.

There should be one general thread function throughout this assignment, as in task 1, and this function should be shared with task 1. Regardless of the scheduling algorithm being used, all threads should fork using a single function.

Your program must print output statements denoting important events, including: the creation of new threads, the contents of the ready queue, the dispatcher algorithm in use, the task being selected each time the dispatcher runs, and the burst time that each task gets to execute for.

### Report

As part of your report, answer the following questions about Task 2:

1. Which scheduling algorithm was most efficient for use in a multi-core system? Why?

## Task 4: Command Line

The command line requirements for this assignment are a little more involved than in the past two projects. You need to allow for multiple arguments, which may themselves take multiple parameters. The scheduling algorithm should be selected using the argument -S, which takes 1 or 2 integer parameters. The first parameter is an integer between 1-4, specifying the algorithm to use, and when the round-robin algorithm is selected, a second integer parameter between 2-10 is required, specifying the time quantum. Your program should also accept the argument -C, which requires a parameter between 1-4, specifying the number of CPU cores. If -C is not provided, your program should default to a single core.

The list of valid command line arguments is as follows:

| Argument | Meaning |
| --- | --- |
| -S <algorithm> | Scheduling algorithm (**required**) – <algorithm>: integer, 1-4 |
| -S **1** | First-come, first-served |
| -S **2** <quantum> | Round-robin – <quantum>: integer, 2-10 |
| -S **3** | Non-preemptive shortest job first |
| -S **4** | Preemptive shortest job first |
| -C <cores> | Number of cores (**optional, defaults to 1**) – <cores>: integer, 1-4 |

Make sure you validate these arguments and parameters for sanity and the correct value range, including determining whether the arguments and parameters exist. For example, -S without a parameter should not cause your program to crash; instead, your program should deliver an appropriate error message to the user and exit cleanly. Also, note that -S and -C can potentially be given in any order, so your program should not assume that -S will always come first: “-S 1 -C 1” and “-C 1 -S 1” should do the same thing.

If you are running your program directly from the command line, add your arguments and parameters to the end of the command you use to run the program. For example, to run the scheduler on a single core using the first-come, first-served algorithm, you would type something like “./main.exe -C 1 -S 1”.

For Java in IntelliJ, you can pass one or more arguments to your program using the Program Arguments field under Run > Edit Configurations. For example, for task 2, using round-robin scheduling with a time quantum of 5, running on 4 cores, you would type “-S 2 5 -C 4” here.

## Task 5: Report

Please submit a detailed report describing your design and implementation. In addition to the questions listed under each task, the report should answer the following questions:

1. Which algorithm was the most difficult to implement for a single-core system and for a multi-core system?
2. In your own words, explain how you implemented each task. Did you encounter any bugs? If so, how did you fix them? If you failed to complete any tasks, list them here and briefly explain why.
3. What sort of data structures and algorithms did you use for each task?

Your report should be in .pdf, .txt, .doc, or .odt format. Other formats are acceptable, but you run the risk of the TA being unable to open or read it. Such reports will receive 0 points with no opportunity for resubmission.

Clearly include the name and ULID of all group members in your report. The questions in the report should be arranged by their associated task, then numbered. There is no minimum length, although insufficient detail in your answers will result in a penalty.

## Submission

Throughout your code, use comments to indicate which sections of code have been edited by you. This will help us better find and grade your work. For example:

//Begin code changes by <your name here>.

...

//End code changes by <your name here>.

**For C/C++**, use the following directory structure, then compress to a .zip or .tar.gz archive.

project03\_<ULID>

│

└───project03\_<ULID>\_report.(pdf|txt|doc|odt)

│

└───code

│

└───main.c

│

└───*If needed:* <sources>.(c|cpp), <sources>.h, <subfolders>

Here, <sources>.(c|cpp), <sources>.h, and <subfolders> refer to any extra source files, headers, or subfolders your program needs. Name the source file with your main() function (where program execution will start) main.c or main.cpp.

You do not need to include any object files (of type .o), link libraries, or compiled executables in your submission. Please include the command used to compile your project in your report, as the TA may not use the same type of machine or specific development environment that you do.

**For Java**, use the following directory structure, then compress to a .zip or .tar.gz archive.

project03\_<ULID>

│

└───project03\_<ULID>\_report.(pdf|txt|doc|odt)

│

└───code

│

└───.idea

│

└───code.iml

│

└───src

│

└───com

│

└───company

│

└───Main.java

│

└───*If needed:* <sources>.java, <subfolders>

The names of the folders labeled here <com> and <company> may vary based on how you have set up your project; the folder hierarchy may be deeper or shallower than the one depicted here. The important part is that the innermost folder under src contains your Java code.

Here, <sources>.java and <subfolders> refer to any extra source files or subfolders your program needs. Name the source file with your main() method (where program execution will start) Main.java.

You do not need to include any files of type .class or .jar in your submission.

Do not submit a paper copy.

Late and improper submissions will receive a maximum of 50% credit for the first 24 hours after the deadline and zero credit afterward.

Submit your code as well as a snapshot of your code execution.

## Hints

You are expected to validate user input. The same concerns about acceptable input types and edge cases within that type apply. Under no circumstances should your code segfault, initiate a stack dump, or terminate with an assertion failure or an uncaught exception. It will be thoroughly tested with a wide range of valid and invalid inputs to check for this.

Your code should gracefully handle any errors that crop up. Graceful handling means not letting the program crash, but instead catching and managing the error so that, at the very least, the program can exit normally. Any crash, segfault, or otherwise abnormal program termination will result in a penalty, no matter the cause, so it is in your best interest to use good error-handling practices.