Project 2

A CPU Scheduling Simulator

Operating Systems

Department of Computer Science

Colorado School of Mines

March 7, 2020

Assigned Date: March 7, 2020
Deliverable 1 Due: 23:59 March 16, 2020
Deliverable 2 Due: 23:59 April 6, 2020

Introduction

The goal of this project is to develop a CPU scheduling simulation that will complete the execution of a group of multi-threaded processes. It will support several different scheduling algorithms. The user can then specify which one to use via a command-line flag. At the end of execution, your program will calculate and display several performance criteria obtained by the simulation.

Learning goals:

1. You will have better familiarity with one of the main roles of any operating system: process scheduling.
2. You will become familiar with event-driven simulation.
3. You will understand the performance implications of using different scheduling algorithms. In the future, you can reuse these concepts (scheduling, simulation, etc) in any optimization task you are given in your professional life.

This project must be implemented in C++, and it must execute correctly on the computers in the Alamode lab.

1 Simulation Constraints

The program will simulate process scheduling on a hypothetical computer system with the following attributes:

1. There is a single CPU, so only one process can be running at a time.
2. There are an infinite number of I/O devices, so any number of processes can be blocked on I/O at the same time.
3. Processes consist of one or more kernel-level threads (KLTs).
4. Threads (not processes) can exist in one of five states:
   (a) `NEW`
   (b) `READY`
   (c) `RUNNING`
   (d) `BLOCKED`
   (e) `EXIT`
5. Dispatching threads requires a non-zero amount of OS overhead:
   (a) If the previously executed thread belongs to a different process than the new thread, a full process switch occurs. This is also the case for the first thread being executed.
   (b) If the previously executed thread belongs to the same process as the new thread being dispatched, a cheaper thread switch is done.
   (c) A full process switch includes any work required by a thread switch.
6. Threads, processes, and dispatch overhead are specified via a file whose format is specified in the next section.

7. Each thread requires a sequence of CPU and I/O bursts of varying lengths as specified by an input file.
8. Processes have an associated priority, specified as part of the file. Each thread in a process has the same priority as its parent process.
   (a) `0:` `SYSTEM` (highest priority)
   (b) `1:` `INTERACTIVE`
   (c) `2:` `NORMAL`
   (d) `3:` `BATCH` (lowest priority)
9. All processes have a distinct process ID, specified as part of the file. Thread IDs are unique only within the context of their owning process (so the first thread in every process has an ID of 0).
10. Overhead is incurred only when dispatching a thread (transitioning it from `READY` to `RUNNING`); all other OS actions require zero OS overhead. For example, adding a thread to a ready queue or initiating I/O are both "free".
11. Threads for a given process can arrive at any time, even if some other process is currently running (i.e., some external entity—not the CPU—is responsible for creating threads).
12. Threads get executed, not processes.

2 Scheduling Algorithms

You scheduling simulator must support three different scheduling algorithms. These are as follows, with the corresponding flag value indicated in parenthesis:

- First Come, First Served (`--algorithm FCFS`)
- Round Robin (`--algorithm RR`)
- Priority (`--algorithm PRIORITY`)

2.1 First Come, First Served (FCFS)

First come, first served should be implemented as described in your textbook. That is to say, threads are scheduled in the order that they are added to the queue, and they run in the CPU until their burst is complete. There is not preemption in this algorithm, and all the process priorities are treated as equal.

2.2 Round Robin (RR)

Round robin should be implemented as described in your textbook. That is to say, threads are scheduled in the order that they are added to the queue. However, unlike FCFS, threads may be preempted if their CPU burst length is greater than the round robin time slice. In the event of a preemption, the thread is removed from the CPU and placed at the back of the ready queue. The CPU burst length is updated to reflect the time that it was able to spend on the CPU. All the process priorities are treated as equal.

The default time slice for the algorithm shall be 3, however, the user may input via command line flag a custom time slice.

2.3 Priority (PRIORITY)

Your priority scheduling algorithm is a non-preemptive algorithm that uses four separate first come, first served ready queues. These queues consist of the following:

- Queue 0: Dedicated to threads whose processes are of type `SYSTEM`.

- Queue 1: Dedicated to threads whose processes are of type `INTERACTIVE`.
- Queue 2: Dedicated to threads whose processes are of type `NORMAL`.
- Queue 3: Dedicated to threads whose processes are of type `BATCH`.

Your priority algorithm should select a new thread from the highest priority queue available (`SYSTEM` is a higher priority than `INTERACTIVE`, etc.)

2.4 Extra Credit Algorithms

For up to 10% extra credit each, you may implement the following scheduling algorithms.

2.4.1 Multi-level Feedback Queue (MLFQ)

Your multi-level feedback queue algorithm (`--algorithm MLFQ`) should follow these requirements:

- There are 10 queues.
- The algorithm is preemptive with a default time slice of 3, but the user is able to input a custom slice from the command line (using the `-s, --time_slice` flag).
- New threads are placed in the queue corresponding to its process priority.
- When *preempted*, threads are demoted to the next lower queue level (if possible).

2.4.2 Custom (CUSTOM)

You are to design your own custom scheduling algorithm (`--algorithm CUSTOM`), with the requirement that it must be better than the first come, first served algorithm in one metric from the following list:

- Average response time (averaged across all threads): `response-time`
- Average turnaround time (averaged across all threads): `turnaround-time`
- CPU utilization: `cpu-utilization`
- CPU efficiency: `cpu-efficiency`

The second item in the list is what you should add to the `custom` file (see Section 4).

3 Next-Event Simulation

Your simulation structure must follow the next-event pattern. At any given time, the simulation is in a single state. The simulation state can only change at event times, where an event is defined as an occurrence that may change the state of the system.

Since the simulation state only changes at an event, the ”clock” can be advanced to the next scheduled event–regardless of whether the next event is 1 or 1,000,000 time units in the future. This is why it is called a ”next-event” simulation model. In our case, time is measured in simple ”units”. Your simulation must support the following event types:

- `THREAD ARRIVED`: A thread has been created in the system.
- `THREAD DISPATCH COMPLETED`: A thread switch has completed, allowing a new thread to start executing on the CPU.
- `PROCESS DISPATCH COMPLETED`: A process switch has completed, allowing a new thread to start executing on the CPU.
- `CPU BURST COMPLETED`: A thread has finished one of its CPU bursts and has initiated an I/O request.

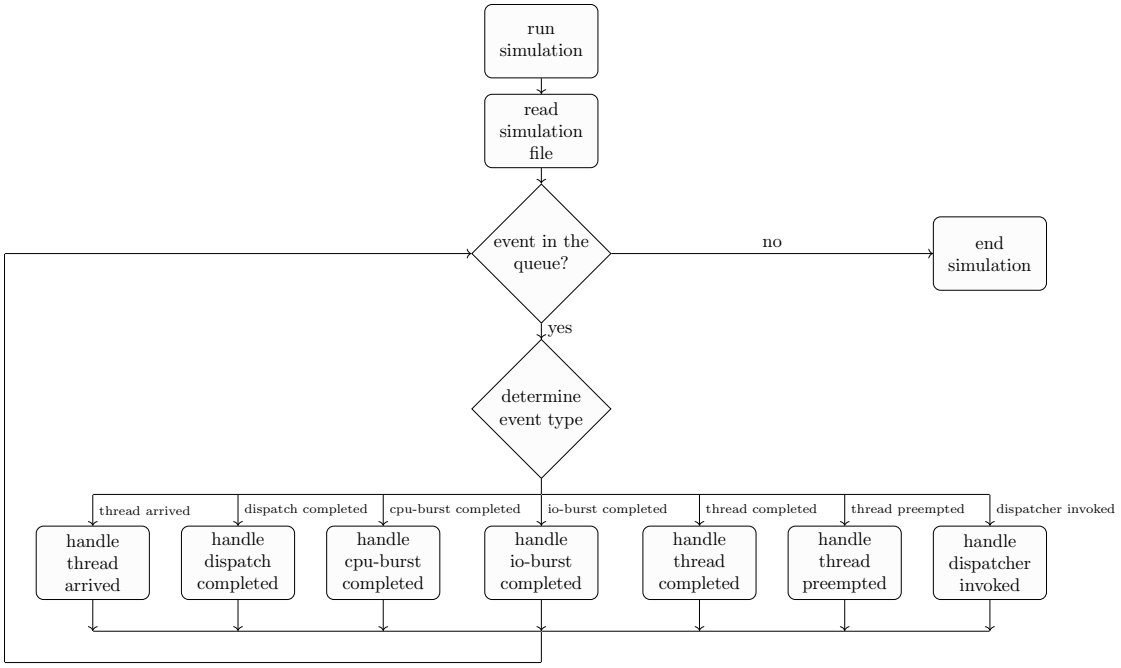

Figure 1: A high level illustration of the next-event simulation. In the starter code, all of this functionality is to be implemented within the `Simulation` class. Rounded rectangles represent functions, while diamonds are decisions that lead to different actions being taken. For example, if the event type is determined to be `THREAD ARRIVED`, then the `handle_thread_arrived(event)` function should be called.

- `IO BURST COMPLETED`: A thread has finished one of its I/O bursts and is once again ready to be executed.
- `THREAD COMPLETED`: A thread has finished the last of its CPU bursts.
- `THREAD PREEMPTED`: A thread has been preempted during execution of one of its CPU bursts.
- `DISPATCHER INVOKED`: The OS dispatcher routine has been invoked to determine the next thread to be run on the CPU.

The main loop of the simulation should consist of processing the next event, perhaps adding more future events in the queue as a result, advancing the clock (by taking the next scheduled event from the front of the event queue), and so on until all threads have terminated. See Figure 1 for an illustration of the event simulation. Rounded rectangles indicate functions that you will need to implement to handle the associated event types.

3.1 Event Queue

Events are scheduled via an event queue. The event queue is a priority queue that contains future events; the priority of each item in the queue corresponds to its scheduled time, where the event with the highest "priority" (at the front of the queue) is the one that will happen next.

To determine the next event to handle, a priority queue is used to sort the events. For this project, the event queue should sort based on these criteria:

- The time the event occurs. The earliest time comes first (time 3 comes before time 12).
- If two events have the time, then the tie breaker should be the events' number: as each new event is created, it should be assigned a number representing how many events have been created. For example, the first event in the simulation should be given the number `0`, the second the number `1`, and so on. The earliest number should come first (event number 6 comes before event number 7).

4 Deliverables

You are required to submit each deliverable by 23:59 on the due date, however you may take advantage of your slip days to turn the deliverable in late.

The first deliverable should be submitted through Canvas, while the second deliverable must be submitted using your GitHub repository, created from the GitHub classroom link that will be provided on Canvas.

4.1 Deliverable 1 Due: 23:59 March 16, 2020

This deliverable is designed to help you understand the simulation framework and does not involve any coding.

[**D.1**] For every `handle_*` function, draw a flow chart illustrating what needs to occur in these functions to handle the given event type.

- Figures 2 and 3, for `handle_thread_arrived` and `handle_dispatcher_invoked`, respectively, are provided to help provide an understanding of the type of diagram that you need you create. These diagrams reference functions that may need to be implemented, but whose declarations are in the starter code. Figure 1 is a diagram illustrating the entire next-event simulation. Most of the functionality within Figure 1 should be implemented in the `Simulation` class.

[**D.1**] Using the simulation provided in Appendix A and both the FCFS and RR algorithms:

- Create a trace of events and transitions, by hand (i.e., not programmatically).
- In addition, calculate all the required statistics and metrics for the simulation, by hand (i.e., not programmatically)—see Section 9 for the simulation output requirements, and Appendix B for an example of what you would need to turn in.

Submit these items through Canvas.

4.2 Deliverable 2: Due: 23:59 April 6, 2020

[**D.2**] Implement the entire process simulation. Using starter code is optional as long as your code passes the items in the checklist and tests given in Section 5.

4.2.1 D2 CHECKLIST

Please MAKE SURE you do all the following, prior to submission:

1. Your code compiles on ALAMODE machines: To compile your code, the grader should be to `cd` into the root directory of your repository and simply type `make` using the provided `makefile`.
2. Your simulation should be able to be executed by typing `./cpu-sim` in the root directory of your repository.
3. You keep the `makefile`, the `test-my-work.sh`, and `submit-my-work` files, as well as the `src/`, `submission-details/`, and `tests/` folders from the starter code, in the root directory of your repository.
4. Your program parses input flags correctly, and outputs the correct information in response. See Sections 8 and 9.
5. Your program determines the file to parse from the command line.
6. You have the full simulation logic implemented.
7. The FCFS, RR and PRIORITY algorithms are implemented.
8. All required metrics are displayed on program completion and match the user input flag choices.

9. Any improper command line input should cause your program to print the help message and then immediately exit.
10. Your code passes all the tests given in Section 5 on ALAMODE machines.
11. Make sure the `submission-details/` folder contains:
    - An `author` file that contains your name: from the root of your repository, type `echo YOUR NAME > submission-details/author`
    - A `time-spent` file that contains the time you have spent on this project, in *minutes*: Please keep entering `echo MINUTES >> submission-details/time-spent` as you progress through the project.
12. Extra credit algorithms (MLFQ and CUSTOM) should be included in your submission, if you decide to do so.
    - If you have implemented the multi-level feedback queue algorithm, create a file inside `submission-details/` called `mlfq`: within the directory, run this command: `touch mflq`
    - If you have implemented a custom algorithm, create a file inside `submission-details/` called `custom` that contains the metric that your algorithm improves upon: within the directory, run this command: `echo metric > custom` (replace `metric` with your chosen metric from Section 2!)
13. You *commit*ed and *push*ed your code.
14. The submission script, `submit-my-work`, successfully runs.
    - This script has been provided with the starter code so that your code compiles and it is properly committed at the time of submission.
    - To use it, make sure that it has execution permissions (`chmod +x submit-my-work`) and type `./submit-my-work` from the root of your repository.

5 Testing your simulation and grading

Grading for this project is dependent on your program's ability to produce the correct output given a simulation input file, so it is vital that you follow all output formatting requirements.

- The `tests/` folder in the starter code contains a number of input and output pairs that your simulation will be tested against. 80% of your D2 grade will be based on the successful execution of the script below. The scripts runs your simulation for every input file in the `tests/input/` folder, and runs `diff` between the output of your simulation against the reference outputs under `tests/output/` folder. If there is no difference (i.e., no output), your simulation ran as expected.
- The remaining 20% of your D2 grade will be based on the input files we will generate during grading. This is to make sure that you haven't hard-coded the outputs in your simulation.
- You should expect your code to be evaluated based on how similar it is to the expected output by using a function such as `diff`. Make sure that all debugging and other non-required print statements have been commented out before submitting your code. Both `stdout` and `stderr` will be captured, so ensure that nothing unexpected is going to be printed to either of these output streams. Logger functionality is provided with the starter code to help ensure that your program will output as expected by the grading scripts.

In order for you to easily test your simulation against the inputs and outputs under the `tests/` folder, we have provided a bash script named `test-my-work.sh` in the root directory of your repository. You can run it by typing `./test-my-work.sh` (ensure it has execution permissions). For a specific, input/output/parameter combination, if the output of your simulation does not match the expected output, the testing will stop and

give you more details. Otherwise, it will print a `Test passed!` message. We will use a similar script in our grading.

6 Getting Started

Starter code has been provided for you to help you get started. The starter code contains complete code that implements logger functionality, a class called `Logger`, so that you can easily print output in the correct format. The `Simulation` class has its functionality for reading and parsing the simulation file implemented for you, but you will need to implement the rest of the functionality for the next-event simulation. A number of other classes have also been provided, however you will need to implement many of them. The starter code contains documentation to help you understand how these classes and their functionality should be implemented, so it is recommended that you read through the starter code carefully before starting to program.

Included with the starter code is a string formatting library, fmtlib[1]. To use the string formatting library, you will need to `#include "utilities/fmt/format.h"` in your file. You can see an example of how to use the library within `src/utilities/logger.cpp`.

You are free to use the starter code and the libraries if you find them beneficial for implementing your project. You are not required to use any of the provided starter code, and as long as your program is implemented in C++, runs on the Alamode computers, does not crash, meets all specified requirements, and produces the correct output, you are free to design your program as you see fit.

The starter code includes a `makefile` that builds everything under the `src/` directory, placing temporary files in a `bin/` directory and the program itself, named `cpu-sim`, in the root of the repository. Do not make changes to the `makefile` without prior approval by the instructors.

Chapter 9 in your textbook describes uniprocessor scheduling, and provides good background information on what you are trying to implement. It also provides a number of diagrams that you may find helpful for understanding how threads should be between states (for example, Figure 9.1).

7 Simulation File Format

The simulation file specifies a complete specification of scheduling scenario. It's format is as follows:

```
num_processes thread_switch_overhead process_switch_overhead

process_id process_type num_threads     // Process IDs are unique
thread_0_arrival_time num_cpu_bursts
cpu_time io_time
cpu_time io_time
...                                     // Repeat for num_cpu_bursts
cpu_time

thread_1_arrival_time num_cpu_bursts
cpu_time io_time
cpu_time io_time
...                                     // Repeat for num_cpu_bursts
cpu_time

...                                     // Repeat for the number of threads

process_id process_type num_threads     // We are now reading in the next process
thread_0_arrival_time num_cpu_bursts
cpu_time io_time
```

[1] https://github.com/fmtlib/fmt

```
cpu_time io_time
...                                     // Repeat for num_cpu_bursts
cpu_time

thread_1_arrival_time num_cpu_bursts
cpu_time io_time
cpu_time io_time
...                                     // Repeat for num_cpu_bursts
cpu_time

...                                     // Repeat for the number of threads

...                                     // Keep reading until EOF is reached
```

Here is a commented example. The comments will not be in an actual simulation file.

```
2 3 7       // 2 processes, thread overhead is 3, process overhead is 7

0 1 2       // Process 0, Priority is INTERACTIVE, it contains 2 threads
0 3         // The first thread arrives at time 0 and has 3 bursts
4 5         // The first pair of bursts: CPU is 4, IO is 5
3 6         // The second pair of bursts: CPU is 3, IO is 6
1           // The last CPU burst has a length of 1

1 2         // The second thread in Process 0 arrives at time 1 and has 2 bursts
2 2         // The first pair of bursts: CPU is 2, IO is 2
7           // The last CPU burst has a length of 7

1 0 3       // Process 1, priority is SYSTEM, it contains 3 threads
5 3         // The first thread arrives at time 5 and has 3 bursts
4 1         // The first pair of bursts: CPU is 4, IO is 1
2 2         // The second pair of bursts: CPU is 2, IO is 2
2           // The last CPU burst has a length of 2

6 2         // The second thread arrives at time 6 and has 2 bursts
2 2         // The first pair of bursts: CPU is 2, IO is 2
3           // The last CPU burst has a length of 3

7 5         // The third thread arrives at time 7 and has 5 bursts
5 7         // CPU burst of 5 and IO of 7
2 1         // CPU burst of 2 and IO of 1
8 1         // CPU burst of 8 and IO of 1
5 7         // CPU burst of 5 and IO of 7
3           // The last CPU burst has a length of 3
```

8 Command Line Parsing

Your simulation must support invocation in the format specified below, including the following command line flags:

```
./cpu-sim [flags] [simulation_file]

-h, --help
    Print a help message on how to use the program.

-m, --metrics
    If set, output general metrics for the simulation.
```

```
-s, --time_slice [positive integer]
    The time slice for preemptive algorithms.

-t, --per_thread
    If set, outputs per-thread metrics at the end of the simulation.

-v, --verbose
    If set, outputs all state transitions and scheduling choices.

-a, --algorithm <algorithm>
    The scheduling algorithm to use. Valid values are:
        FCFS: first come, first served (default)
        RR: round robin scheduling
        PRIORITY: priority-based scheduling
        MLFQ: multi-level feedback queue
        CUSTOM: a custom algorithm
```

Users should be able to pass any flags together, in any order, provided that:

- If the `--help` flag is set, a help message is printed to `stdout` and the program *immediately* exits.
- If `--time_slice` is set, it must be followed immediately by a positive integer.
- If `--algorithm` is set, it must be followed immediately by an algorithm choice.
- If `--algorithm` is not set, your program shall default to using first come, first served as its scheduling algorithm.
- If a filename is not provided, the program shall read in from `stdin`.

Any improper command line input should cause your program to print the help message and then immediately exit. Information on proper output formatting can be found in Section 9.

You are *strongly encouraged* to use the `getopt` family of functions to perform the command line parsing. Information on `getopt` can be found here: `http://man7.org/linux/man-pages/man3/getopt.3.html`

9 Output Formatting

For efficient and fair grading, it is vital that your simulation outputs information in a well-defined way. The starter code provides functionality for printing information, and it is *strongly encouraged* that you use it. The information that your simulation prints is dependent on the flags that the user has input, and in the following sections we describe what should be printed for each flag.

9.1 No flags input

If the user has not input any flags to your program, you should only output the following:

```
SIMULATION COMPLETED!
```

9.2 `--metrics`

When the `metrics` flag has been passed to your simulation, it should output the following information:

```
1 SIMULATION COMPLETED!
2
3 SYSTEM THREADS:
4     Total Count:                  3
5     Avg. response time:       23.33
```

```
    Avg. turnaround time:     94.67

INTERACTIVE THREADS:
    Total Count:                 2
    Avg. response time:       10.00
    Avg. turnaround time:     73.50

NORMAL THREADS:
    Total Count:                 0
    Avg. response time:        0.00
    Avg. turnaround time:      0.00

BATCH THREADS:
    Total Count:                 0
    Avg. response time:        0.00
    Avg. turnaround time:      0.00

Total elapsed time:            130
Total service time:             53
Total I/O time:                 34
Total dispatch time:            69
Total idle time:                 8

CPU utilization:            93.85%
CPU efficiency:             40.77%
```

9.3 --per_thread

When the `per_thread` flag has been passed to your simulation, it should output information about each of the threads.

```
SIMULATION COMPLETED!

Process 0 [INTERACTIVE]:
    Thread  0:    ARR: 0      CPU: 8      I/O: 11     TRT: 88     END: 88
    Thread  1:    ARR: 1      CPU: 9      I/O: 2      TRT: 59     END: 60

Process 1 [SYSTEM]:
    Thread  0:    ARR: 5      CPU: 8      I/O: 3      TRT: 92     END: 97
    Thread  1:    ARR: 6      CPU: 5      I/O: 2      TRT: 69     END: 75
    Thread  2:    ARR: 7      CPU: 23     I/O: 16     TRT: 123    END: 130
```

9.4 --verbose

When the `verbose` flag has been passed to your simulation, it should output, at each state transition, information about the state transition that is occurring. It should be outputting this information "on the fly".

```
At time 0:
    THREAD_ARRIVED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from NEW to READY

At time 0:
    DISPATCHER_INVOKED
    Thread 0 in process 0 [INTERACTIVE]
    Selected from 1 threads. Will run to completion of burst.
```

This continues until the end of the simulation:

```
1
2  At time 127:
3      THREAD_DISPATCH_COMPLETED
4      Thread 2 in process 1 [SYSTEM]
5      Transitioned from READY to RUNNING
6
7  At time 130:
8      THREAD_COMPLETED
9      Thread 2 in process 1 [SYSTEM]
10     Transitioned from RUNNING to EXIT
11
12 SIMULATION COMPLETED!
```

9.5 Multiple Flags

If multiple flags are input, all should be printed, in this order:

1. The verbose information.
2. `SIMULATION COMPLETED!`
3. Per thread metrics.
4. General simulation metrics.

9.6 Recommendations

Again, it is highly recommended that you take advantage of the existing logger functionality!

Appendices

A Deliverable 1 Simulation

```
1 4 9

4 1 1
3 4
2 9
5 3
8 2
9
```

B Example Simulation Output

For the following simulation:

```
1 3 7

0 1 1
0 3
4 5
3 6
1
```

this was output:

```
At time 0:
    THREAD_ARRIVED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from NEW to READY

At time 0:
    DISPATCHER_INVOKED
    Thread 0 in process 0 [INTERACTIVE]
    Selected from 1 threads. Will run to completion of burst.

At time 7:
    PROCESS_DISPATCH_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from READY to RUNNING

At time 11:
    CPU_BURST_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from RUNNING to BLOCKED

At time 16:
    IO_BURST_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from BLOCKED to READY

At time 16:
    DISPATCHER_INVOKED
    Thread 0 in process 0 [INTERACTIVE]
    Selected from 1 threads. Will run to completion of burst.

```

```
At time 19:
    THREAD_DISPATCH_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from READY to RUNNING

At time 22:
    CPU_BURST_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from RUNNING to BLOCKED

At time 28:
    IO_BURST_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from BLOCKED to READY

At time 28:
    DISPATCHER_INVOKED
    Thread 0 in process 0 [INTERACTIVE]
    Selected from 1 threads. Will run to completion of burst.

At time 31:
    THREAD_DISPATCH_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from READY to RUNNING

At time 32:
    THREAD_COMPLETED
    Thread 0 in process 0 [INTERACTIVE]
    Transitioned from RUNNING to EXIT

SIMULATION COMPLETED!

Process 0 [INTERACTIVE]:
    Thread  0:    ARR: 0      CPU: 8      I/O: 11     TRT: 32     END: 32

SYSTEM THREADS:
    Total Count:                  0
    Avg. response time:        0.00
    Avg. turnaround time:      0.00

INTERACTIVE THREADS:
    Total Count:                  1
    Avg. response time:        7.00
    Avg. turnaround time:     32.00

NORMAL THREADS:
    Total Count:                  0
    Avg. response time:        0.00
    Avg. turnaround time:      0.00

BATCH THREADS:
    Total Count:                  0
    Avg. response time:        0.00
    Avg. turnaround time:      0.00

Total elapsed time:             32
Total service time:              8
Total I/O time:                 11
Total dispatch time:            13
```

```
Total idle time:                  11

CPU utilization:              65.62%
CPU efficiency:               25.00%
```

C Function Diagrams

This section contains a couple function diagrams, similar to the ones that you will need to create for Deliverable 1. These diagram reference functions that are present, but may need to be implemented, in the starter code. For example, `handle_thread_arrived(event)` is a function within the `Simulation` class.

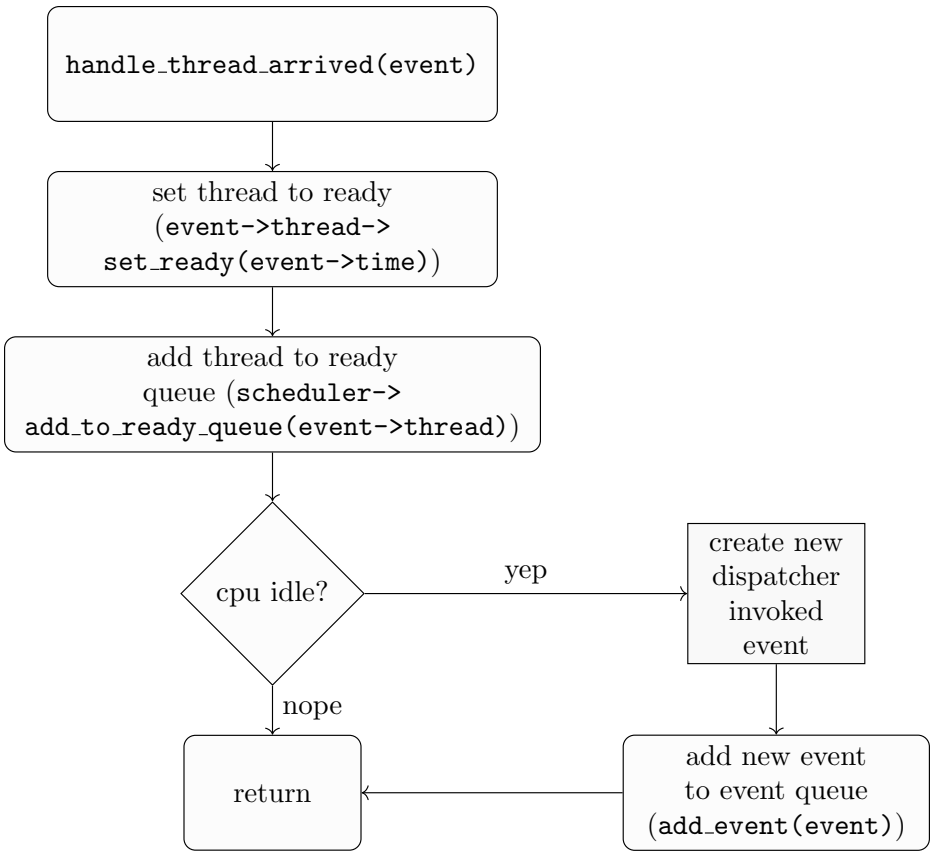

Figure 2: An example implementation of the `handle_thread_arrived(event)` function.

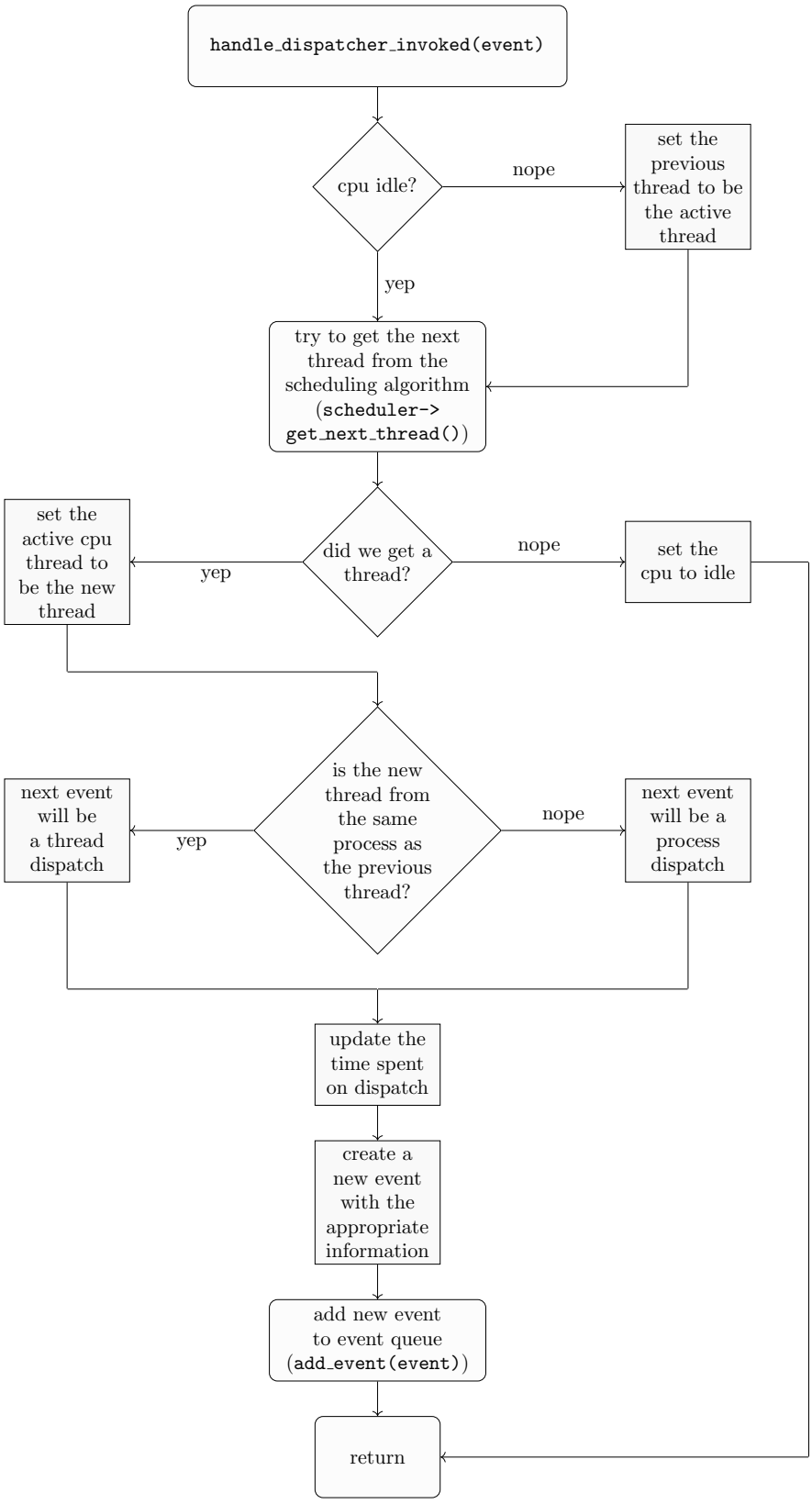

Figure 3: An example implementation of the `handle_dispatcher_invoked(event)` function.