CSE130 Spring 2021 : Assignment 4

In this assignment you will implement a selection of CPU scheduling algorithms and test them against a simulator that generates groups of threads with random arrival times and random bursts of CPU and I/O activity.

This lab is worth 5% of your final grade.

Submissions are due NO LATER than 23:59, Monday May 10 2021 (1 week)

Setup

SSH in to one of the two CSE130 teaching servers using your CruzID Blue password:

```
      $ ssh <cruzid>@noggin.soe.ucsc.edu
or    $ ssh <cruzid>@nogbad.soe.ucsc.edu
or    $ ssh <cruzid>@olaf.soe.ucsc.edu
or    $ ssh <cruzid>@thor.soe.ucsc.edu
```

(use Putty http://www.putty.org/ if on Windows)

Authenticate with Kerberos: ***(do this every time you log in)***

```
$ kinit
```

(you will be prompted for your Blue CruzID password)

Authenticate with AFS: ***(do this every time you log in)***

```
$ aklog
```

Create a suitable place to work: ***(only do this the <u>first</u> time you log in)***

```
$ mkdir -p CSE130/Assignment4
$ cd CSE130/Assignment4
```

Install the lab environment: ***(only do this once)***

```
$ tar xvf /var/classes/CSE130/Assignment4.tar.gz
```

Build the starter code:

```
$ cd ~/CSE130/Assignment4
$ make
```

(always work in this directory)

Then try:

```
$ make grade
```

(runs the required functional tests - see below)
(also tells you what grade you will get - see below)

Run the scheduler executable:

```
$ ./scheduler <options> <algorithm>

options are:
   -h, --help      show this message
   -l, --license   show license information
   -v, --verbose   verbose output
   -t <int>        number of threads (default : 4)
   -q <int>        round-robin quantum (default : 4, min 1)
```



```
algorithm is one of:
   --fcfs          First Come First Served (default)
   --rr            Round Robin
   --np-priority   Non-Preemptive Priority
   --p-priority    Preemptive Priority
   --np-sjf        Non-Preemptive Shortest Job First
   --p-sjf         Preemptive Shortest Job First
   --np-srtf       Non-Preemptive Shortest Remaining Time First
   --p-srtf        Preemptive Shortest Remaining Time First
```

Run 6 threads First Come First Served:

```
$ ./scheduler -t 6 -v --fcfs
```

Run 2 threads Round Robin with a quantum of 3:

```
$ ./scheduler -t 2 -q 3 -v --rr
```

Run 10 threads Preemptive Shortest Job First:

```
$ ./scheduler -t 10 --p-sjf
```

Background Information

Consult the lecture handouts, the textbook, and on-line resources to refresh your memory of what a CPU scheduler is and how the First Come First Served (FCFS) and other scheduling algorithms works.

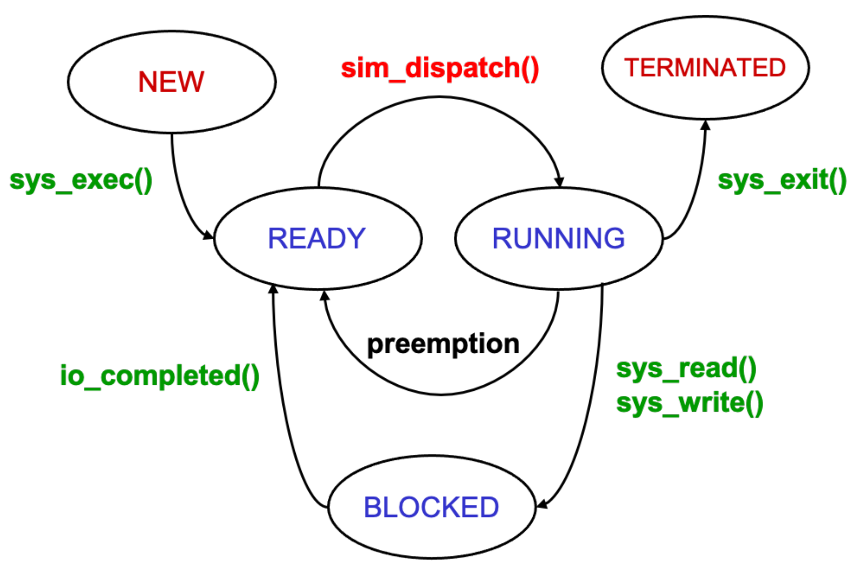
Two types of thread execution profiles are generated by the simulator:

- Single CPU burst
- Two CPU bursts separated by an IO burst

For FCFS, in the first case a dispatched thread runs to completion. In the second case a dispatched thread runs for a while them makes an IO request; once the IO request is complete the tread is returned to the ready queue and once dispatched for the second time runs to completion.

Example profiles and execution outputs are given in **Appendix 1.**

Lifecycle of a thread can be illustrated like so:



Functions you nust implement are in **`bold_green()`** functions that you will call are in **`bold_red()`** - consult `scheduler.c` and `simulator.h` for details.

For example: When the simulator creates a new thread, it will call **`sys_exec()`** and your scheduler should move the thread to the READY state by placing it on your ready queue. When your scheduler wants a thread to start executing, it calls **`sim_dispatch()`**.

Notes:

- "**preemption**" is not a function you have to implement per-se, it's the act of kicking threads off the CPU because they no longer have the right to be there. For example:
  - Round Robin:
    - The quantum has expired and there are threads in the ready queue
  - Preemptive Priority
    - A thread with a higher priority (see `simulator.h`) arrived
  - Preemptive Shortest Job First
    - A thread with a shorter length (see `simulator.h`) arrived

- For this assignment, waiting time is defined as the sum of time spent waiting for the CPU and time waiting for a requested IO operation to start.
  - An example calculation of turnaround and waiting times can be found in **Appendix 2**.

Provided Scripts

To aide debugging, a couple of scripts are provided:

`compare.sh`

Compares the schedule you generated to the expected schedule. For example:

```
$ ./compare.sh -t 4 -v --fcfs
```

Will display the message "Schedules Match" when they match, even if your turnaround and waiting time calculations are incorrect. "ERROR Schedules Mismatched" is displayed when the schedules do not match.

The number of threads (the `-t` option) must be less than or equal to 8.

`mismatch.sh`

Repeatedly runs `compare.sh` until a mismatched schedule is found, or 500 matching schedules have been generated. For example:

```
$ ./mismatch.sh -t 7 -v --rr
```

Will stop when your generated schedule and the expected schedule differ.

The number of threads (the `-t` option) must be less than or equal to 8.

Press `Ctrl-C` ("Control C") to interrupt (stop) the script.

Requirements

FCFS Scheduler:

- Accurately calculate mean turnaround and mean waiting times for 150 randomly generated thread execution profiles, each for between 2 and 31 threads. The more you get right, the more points your score.

RR Scheduler:

- As for FCFS.

Non-Preemptive Scheduler:

- Implement one or more of the supported preemptive schedulers:
    - Non-Preemptive Priority
    - Non-Preemptive Shortest Job First
    - Non-Preemptive Shortest Remaining Time First

  If you implement more than one, whichever scores the most passes will be included in your grade.

Preemptive Scheduler:

- Implement one or more of the supported preemptive schedulers:
    - Preemptive Priority
    - Preemptive Shortest Job First
    - Preemptive Shortest Remaining Time First

  If you implement more than one, whichever scores the most passes will be included in your grade.

Extra credit is awarded if you complete all schedulers, and they work perfectly.

What steps should I take to tackle this?

FCFS Scheduler:

- There are any number of different ways to implement the scheduler, but a first step towards understanding how the simulator works is to simply print messages inside each simulator callback - the skeleton functions found in `scheduler.c`.

- Once you can see threads being created, define a linked list for your ready queue and put the threads in it in the order they arrive. Feel free to use the queue implementation defined in `queue.h`, it has everything you need for this assignment.

- Then at any scheduling opportunity, remove a thread from the ready queue and dispatch it.

- Whenever an IO operation completes, place the thread back on the ready queue.

- When the simulation ends (`get_stats()` is called) work out the turnaround time and waiting for each thread and the means of both metrics.

- Recommended approach is to get the scheduling correct then work on taking timings. Attempting both at the same time is rarely productive.

RR Scheduler:

- Base it on your FCFS Scheduler. Add preemption by, at the start of every clock tick, working out if a context switch is required. Care must be taken if a new thread arrives in a tick where the quantum / time-slice expires for the currently executing thread.

Other Schedulers:

- Sort the ready queue according to the scheduling algorithm and take care of preemption as necessary. The non-preemptive priority scheduler is the next simplest to implement after FCFS so you may want to attempt this before RR.

How much code will I need to write?

A model solution that satisfies all requirements (not extra credit) has approximately 100 lines of executable code.

Grading scheme

The following aspects will be assessed:

1. (100%) **Does it work?**

a.	FCFS Scheduler	(30%)
b.	RR Scheduler	(20%)
c.	One Non-Preemptive Scheduler	(20%)
d.	One Preemptive Scheduler	(20%)
e.	Your implementations are free of compiler warnings and memory errors	(10%)

   20% Extra credit is awarded if you complete all schedulers, and they work perfectly.

2. (-100%) **Did you give credit where credit is due?**

   a. Your submission is found to contain code segments copied from on-line resources and you failed to give clear and unambiguous credit to the original author(s) in your source code (-100%). You will also be subject to the university academic misconduct procedure as stated in the class academic integrity policy.

   b. Your submission is determined to be a copy of a past or present student's submission (-100%)

   c. Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes too significant a percentage of your submission:

      - < 25% copied code — No deduction
      - 25% to 50% copied code — (-50%)
      - > 50% copied code — (-100%)

What to submit

In a command prompt:

```
$ cd ~/CSE130/Assignment4
$ make submit
```

This creates a gzipped tar archive named `CSE130-Assignment4.tar.gz` in your home directory.

****** UPLOAD THIS FILE TO THE APPROPRIATE CANVAS ASSIGNMENT ******

Appendix 1 - Example Executions

FCFS: No IO but thread 2 has to wait for thread 1 to finish before it gets the CPU.

```
+-----+----------+--------+-----------+----------+----------+
| tid | priority | arrive | cpu burst | io start | io burst |
+-----+----------+--------+-----------+----------+----------+
|   1 |        0 |      3 |         9 |        0 |        0 |
+-----+----------+--------+-----------+----------+----------+
|   2 |        4 |      3 |         6 |        0 |        0 |
+-----+----------+--------+-----------+----------+----------+

First Come First Served

Simulation:
   3   1 arrive
   3   2 arrive
   3   1 cpu start
  11   1 exit
  12   2 cpu start
  17   2 exit

Scheduled Usage:
      00----+----+----+----+----5----+----+----+----+----+
  cpu |    |    |    |  1 |  1 |  1 |  1 |  1 |  1 |  1 |
   io |    |    |    |    |    |    |    |    |    |    |
      10----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  2 |  2 |  2 |  2 |  2 |  2 |
   io |    |    |    |    |    |    |    |    |
      +----+----+----+----+----+----+----+----+

Expected Usage:
      00----+----+----+----+----5----+----+----+----+----+
  cpu |    |    |    |  1 |  1 |  1 |  1 |  1 |  1 |  1 |
   io |    |    |    |    |    |    |    |    |    |    |
      10----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  2 |  2 |  2 |  2 |  2 |  2 |
   io |    |    |    |    |    |    |    |    |
      +----+----+----+----+----+----+----+----+

Mean Turnaround Time: 12
   Mean Waiting Time:  4
+-----+-----------------+--------------+
| tid | turnaround time | waiting time |
+-----+-----------------+--------------+
|   1 |               9 |            0 |
+-----+-----------------+--------------+
|   2 |              15 |            9 |
+-----+-----------------+--------------+
```

Scheduled Usage is how your code scheduled the threads.

Expected Usage is how they should have been scheduled.

If expected and scheduled usage match (as they do in this example), your scheduler is working correctly.

FCFS: 2 (first thread to arrive) initiates an IO operation before 1 arrives so there is some idle time on the CPU starting at time 6. 1 is still running when the IO operation for 2 completes so 2 has to wait in the ready queue for a while until the CPU becomes available - which it does when 1 requests an IO operation. 2 is still running when the IO operation for 1 completes so 1 has to wait for the CPU to become available again before it can continue

```
+-----+----------+-------+----------+----------+----------+
| tid | priority | arrive | cpu burst | io start | io burst |
+-----+----------+-------+----------+----------+----------+
|   2 |        3 |      2 |        10 |        4 |        5 |
+-----+----------+-------+----------+----------+----------+
|   1 |        1 |      8 |        19 |       14 |        3 |
+-----+----------+-------+----------+----------+----------+

First Come First Served

Simulation:
  2   2 arrive
  2   2 cpu start
  5   2 cpu end
  6   2 io start
  8   1 arrive
  8   1 cpu start
 10   2 io end
 21   1 cpu end
 22   2 cpu start
 22   1 io start
 24   1 io end
 27   2 exit
 28   1 cpu start
 32   1 exit

Scheduled Usage:
    00----+----+----+----+----5----+----+----+----+----+
  cpu |    |    |  2 |  2 |  2 |  2 |    |    |  1 |  1 |
   io |    |    |    |    |    |    |  2 |  2 |  2 |  2 |
    10----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |
   io |  2 |    |    |    |    |    |    |    |    |    |
    20----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  2 |  2 |  2 |  2 |  2 |  2 |  1 |  1 |
   io |    |    |  1 |  1 |  1 |    |    |    |    |    |
    30----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  1 |
   io |    |    |    |
      +----+----+----+

Expected Usage:
    00----+----+----+----+----5----+----+----+----+----+
  cpu |    |    |  2 |  2 |  2 |  2 |    |    |  1 |  1 |
   io |    |    |    |    |    |    |  2 |  2 |  2 |  2 |
    10----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |
   io |  2 |    |    |    |    |    |    |    |    |    |
    20----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  2 |  2 |  2 |  2 |  2 |  2 |  1 |  1 |
   io |    |    |  1 |  1 |  1 |    |    |    |    |    |
    30----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  1 |
   io |    |    |    |
      +----+----+----+

Mean Turnaround Time: 25
   Mean Waiting Time:  7
+-----+-----------------+--------------+
| tid | turnaround time | waiting time |
+-----+-----------------+--------------+
|   1 |              25 |            3 |
+-----+-----------------+--------------+
|   2 |              26 |           11 |
+-----+-----------------+--------------+
```

Round Robin: 1 arrives first in Tick 0 and starts to run. 2 also arrives in Tick 0 but has to wait for 1. At Tick 4, 1 is still on the CPU but the quantum has expired so 2 preempts 1. After three ticks 2 requests IO so 1 gets the CPU back and stays there for 9 ticks as 2 is waiting for IO to complete.

When the IO for 2 completes after 9 ticks, 1 is a single tick into a 4-tick time-slice so is allowed to continue. At the end of Tick 18, 1 has used all the quantum is preempted in favour of 2. After 4 more ticks 2's time-slice expires but 2 is finished anyway so 1 is given the CPU to complete its last two CPU ticks.

```
+-----+----------+--------+-----------+----------+----------+
| tid | priority | arrive | cpu burst | io start | io burst |
+-----+----------+--------+-----------+----------+----------+
|   1 |        4 |      0 |        18 |        0 |        0 |
+-----+----------+--------+-----------+----------+----------+
|   2 |        2 |      0 |         7 |        3 |        9 |
+-----+----------+--------+-----------+----------+----------+

Round-Robin (quantum = 4)

Simulation:
   0   1 arrive
   0   2 arrive
   0   1 cpu start
   4   2 cpu start
   6   2 cpu end
   7   1 cpu start
   7   2 io start
  15   2 io end
  19   2 cpu start
  22   2 exit
  23   1 cpu start
  24   1 exit

Expected Usage:
    00----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  1 |  1 |  2 |  2 |  2 |  1 |  1 |  1 |
   io |    |    |    |    |    |    |    |  2 |  2 |  2 |
    10----+----+----+----+----5----+----+----+----+----+
  cpu |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 |  2 |
   io |  2 |  2 |  2 |  2 |  2 |  2 |    |    |    |    |
    20----+----+----+----+----5----+----+----+----+----+
  cpu |  2 |  2 |  2 |  1 |  1 |
   io |    |    |    |    |    |
      +----+----+----+----+----+

Mean Turnaround Time: 24
   Mean Waiting Time:  7
+-----+-----------------+--------------+
| tid | turnaround time | waiting time |
+-----+-----------------+--------------+
|   1 |              25 |            7 |
+-----+-----------------+--------------+
|   2 |              23 |            7 |
+-----+-----------------+--------------+
```

"Scheduled Usage" omitted - identical.

Appendix 2 - Calculating Turnaround and Waiting Times

Some things to remember:

- Time is measured in Ticks.
- A Tick starts at time Tick and ends at time Tick + 1
- If a Thread gets on the CPU in Tick 23 and immediately gets off again, it spent 1 Tick on the CPU
- General formula for turnaround time is: Last Tick +1 - Arrive Tick
- From class handouts (Scheduling 1, slide 11) for any thread where start, arrive, finish are times:

$$\textbf{Waiting Time} = (\text{start}_1 - \text{arrival}) + \sum_{n=2}^{N} (\text{start}_n - \text{finish}_{n-1})$$

- Time = End Tick - Start Tick + 1 i.e.
  - Start Time = Start Tick
  - Finish Time = End Tick +1

For the Round Robin example in Appendix 1 (previous page):

Thread 1:

- Arrive Tick: 0 (arrival)
- Last Tick: 24 (finish_3 - 1)

⇨ **Turnaround: 24 + 1 - 0 = <u>25</u>**

- start_1 : 0
- finish_1: 3 + 1 = 4
- start_2 : 7
- finish_2 : 18 + 1 = 19
- start_3 : 23

⇨ **Waiting:**

			Waiting For
start_1 - arrival	0 - 0	0	CPU
+ start_2 - finish_1	7 - 4	+ 3	CPU
+ start_3 - finish_2	23 - 19	+ 4	CPU
		= <u>7</u>	

Thread 2:

- Arrive Tick: 0 (arrival)
- Last Tick: 22 (finish_3 - 1)

⇨ **Turnaround: 22 + 1 - 0 = <u>23</u>**

- start_1 : 4
- finish_1: 6 + 1 = 7
- start_2 : 7
- finish_1: 15 + 1 = 16
- start_3 : 19

⇨ **Waiting:**

			Waiting For
start_1 - arrival	4 - 0	4	CPU
+ start_2 - finish_1	7 - 7	+ 0	IO
+ start_3 - finish_2	19 - 16	+ 3	CPU
		= <u>7</u>	