Computer Science 360 – Introduction to Operating Systems
Summer 2021

Assignment #2
Due: Tuesday, June 30, 11:55 pm via push to your Gitlab.csc repository

Urgent request

There is a lot of detail in this assignment description and some of this detail will only begin to become clear when the whole description is read once through. Therefore I make the following request: *Please read through the whole description once through before sending any questions.* Questions about the interpretation of this assignment's requirements *must* be posted to the RocketChat channel for the course. I will post my answers, clarifications, and possible corrections in the same forum.

Programming Platform

For this assignment **your code must work on the Virtualbox/Vagrant configuration you provisioned for yourself in assignment #0.** You may already have access to your own Unix system (e.g., Ubuntu, Debian, macOS with MacPorts, etc.) yet I recommend you work as much as possible while with your CSC360 virtual machine. Bugs in systems programming tend to be platform-specific and something that works perfectly at home may end up crashing on a different computer-language library configuration. (We cannot give marks for submissions of which it is said "It worked on Visual Studio!")

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. However, sharing of code is strictly forbidden. If you are still unsure about what is permitted or have other questions regarding academic integrity, please direct them as soon as possible to the instructor. (Code-similarity tools will be run on submitted work.) Any fragments of code found on the web and used in your solution **must be properly cited** where it is used (i.e., citation in the form of a comment giving the source of code).

Use of gitlab.csc.uvic.ca

Each student enrolled in the course has been assigned a Git repository at gitlab.csc.uvic.ca. For example, the student having Netlink ID `grogu` would have their CSC 360 repository at this location:

```
https://gitlab.csc.uvic.ca/courses/2021051/CSC360/assignments/grogu/csc360-coursework
```

Please form the address of your repository appropriately and perform a git clone in environment you have provisioned as part of assignment #0. You are also able to access this repository by going to `https://gitlab.csc.uvic.ca` (and use your Netlink username and password to log in at that page).

Goals of this assignment

A. Write a C program implementing a solution to the *vaccination problem* (i.e. a kind of *producer-consumer problem*).

B. Write a C program implementing a solution to the given problem involving *reusable barriers*.

C. Describe your completed work in a `a2/README.md` file submitted along with your solution for this assignment. That is, your description for both work in both parts A and B must appear in this single markdown file.

What is provided to you

There is a significant amount of code provided to you for completing the assignment, and in order to simplify distribution we have chosen to add this code directly into your `gitlab.csc` repository for the course. **As of noon on Tuesday, June 15th**, you will find provided to you the code for Parts A and B within a directory named `a2/` in your own repo.

Part A: The Vaccination Problem

A vaccination centre consists of a registration desk (`reg_desk`) and from one to ten vaccination stations (`vac_station`). Members of the public arriving at the centre present themselves to the registration desk; the registration desk then ***enqueues*** such person (an instance of `PersonInfo`) at onto a single queue used for the whole centre (`queue`). That is, this single queue is an sequence of persons – in essence ordered by arrival at the registration desk – who are now queued up for a vaccination station to become available. (Multiple people arriving at the same time must be enqueued in the order in which they arrive at the registration desk.)

Meanwhile each of the vaccination stations ***dequeues*** a person from this single shared queue, and although each station is capable of vaccinating the person at the front, only one station will successfully remove the that person from the queue and perform their vaccination.

You are given a queue implementation in the files `queue.c` and `queue.h`. Please read this code carefully. However, you are not to modify this code unless given express written permission by the instructor.

Here are the concurrent elements of this problem:

- The registration desk **is a single POSIX thread**, running the code in `reg_desk`.
- Each vaccination station **is its own POSIX thread**, running their own instance of the code in `vac_station`.
- There is a single queue (a global variable) named `queue`.

This is therefore very similar to a producer/consumer problem.

We have provided to you a file named `vaccine.c` which, when completed, will implement the simulation. The code provided accepts two command-line arguments when run in the shell (i.e. the number of vaccination stations for the simulation, and a simulation-events file).

As an example, here are the contents of one of the simulation-events files (i.e., `cases/01.txt`):

```
1:3,60
2:5,70
3:6,50
4:8,30
5:8,40
6:9,50
```

Each line above has the format:

```
<person_id>:<arrival_time>,<service_time>
```

`person_id` uniquely identifies someone arriving for vaccination, `arrival_time` and `service_time` are in tenths of seconds. All values are integers. Please note that the starter code for `reg_desk()` in `vaccine.c` already opens up the given file and reads the file line by line within an event loop.

To continue this example, the simulation events shown above will be used for a single simulation run using three vaccination station via the following command:

```
./vaccine 3 cases/01.txt
```

and what follows is one possible output from such a run (i.e., the specific vaccination stations used by each person is possibly random and may change from run to run):

```
Person 1: Arrived at time 3.
Person 1: Added to the queue.
Vaccine Station 1: START Person 1 Vaccination.
Person 2: Arrived at time 5.
Person 2: Added to the queue.
Vaccine Station 2: START Person 2 Vaccination.
Person 3: Arrived at time 6.
Person 3: Added to the queue.
Vaccine Station 3: START Person 3 Vaccination.
Person 4: Arrived at time 8.
Person 4: Added to the queue.
Person 5: Arrived at time 8.
Person 5: Added to the queue.
Person 6: Arrived at time 9.
Person 6: Added to the queue.
Vaccine Station 3: FINISH Person 3 Vaccination.
Vaccine Station 3: START Person 4 Vaccination.
Vaccine Station 1: FINISH Person 1 Vaccination.
Vaccine Station 1: START Person 5 Vaccination.
Vaccine Station 2: FINISH Person 2 Vaccination.
Vaccine Station 2: START Person 6 Vaccination.
Vaccine Station 3: FINISH Person 4 Vaccination.
Vaccine Station 1: FINISH Person 5 Vaccination.
Vaccine Station 2: FINISH Person 6 Vaccination.
```

Your task is to modify `vaccine.c` as follows:

- provide code in `main()` which creates the necessary `reg_desk` and `vac_station` threads;
- complete the code for `reg_desk` (i.e. code for the producer);
- complete the code for `vac_station` (i.e. code for the consumers) including simulating the vaccination service time required for each specific person arriving at a station;
- initialize and use mutex and condition variables correctly in order to control access to the `queue` by all of the threads;
- produce output to the console as required by the comments in `vaccine.c` in order to report progress of the simulation.

You are only permitted to use POSIX `pthread_mutex_t` and `pthread_cond_t` synchronization constructs in Part A (i.e., you may not use `sem_t` semaphores or any other POSIX Pthreads synchronization constructs).

Part B: Meetup using Pthreads

Preamble: `resource.c`

For this part you are provided some code for listening to network ports and launching server threads. Most of this code must be kept unmodified – in fact, the only files you are to change in order to complete this part of the assignment are:

- `meetup.c`, `meetup.h`

The *meetup* problem described below involves the reading and writing of "resources". Normally we would allow you to declare character arrays for this, but for this part of the assignment we need to introduce a bit of latency into our read to memory and writes from memory. Therefore for this part you will be required to use a new type provided to you named `resource_t` for all shared data (besides Pthreads synchronization constructs):

- `init_resource(resource_t *, char *)` accepts an address to a `resource_t` instance plus a character array (i.e., string) to be used as a label for that resource.
- `read_resource(resource_t *, char *, int)` accepts an address to a `resource_t` instance and a character array, where the contents of the resource are to be copied into the character array. There is a delay before the read is completed. The last parameter is the size of the character array.
- `write_resource(resource_t *, char *, int)` accepts an address to a `resource_t` instance and a character array, where the contents of the character array are to be copied into the resource. There is a delay before the write is completed.
- `print_stats()` accepts an address to a `resource_t` instance and outputs statistics such as the number of reads and writes performed on the resource instance. The last parameter is the size of the character array.

Note that since `resource_t` is a plain-old C struct, you can have as many `resource_t` variables as are needed in your solution to the meetup problem described below. The program `example.c` contains some sample code showing the declaration and use of a `resource_t` variable. To compile and run `example.c`, use `make`:

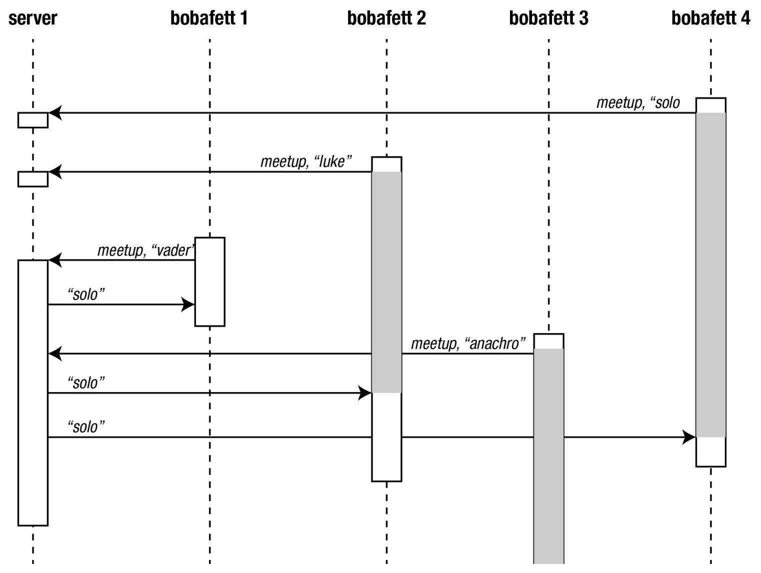
```
$ make example
gcc -c example.c
gcc -o example example.o resource.o
```

```
$ ./example
```

Meetup problem: Description

Consider the following fanciful scenario: There are a large number of people at some cosplay convention in Victoria who have come as Boba Fett (`https://bit.ly/1t4Hn0r`). Groups of them agree that on the morning after the convention they will meet downtown at the entrance to Bastion Square. The groups agree that they will wait for *n* Boba Fetts to arrive, and only once *n* are present, the codeword brought by the first Boba Fett to arrive will be shared with other *n-1* Boba Fetts. After that point, the *n* cosplayers may depart. Because these cosplayers all look like Boba Fett, however, they cannot easily tell each other apart. All that matters, however, is that Boba Fetts group off in sizes of *n*. Therefore the first *n* Boba Fetts to arrive will share the codeword brought by the first to arrive; the next *n* Boba Fetts to arrive will the share the code of the *(n+1)*th Boba Fett who arrives, etc. Boba Fetts are blocked until they know their codeword and all *n* Boba Fetts for the next group have arrived.[1]

Below is a sequence diagram showing one possible scenario where *n*=3. The grey shading used for swimlanes indicates when a Boba Fett thread is blocked. Note that Boba Fett 3 appears to be blocked for the remainder of the scenario (as he/she is perhaps showing off his/her Star Wars knowledge just a little too much.)



[1] This is, of course, a very fanciful scenario – hanging around downtown Victoria wearing cosplay masks might be awkward if not creepy, although it might be considered pandemic-friendly in this day and age...

A meetup is essentially a *barrier* (or more precisely, a *reusable barrier*). The first *n* threads to arrive are synchronized by the barrier – only when there are *n* threads will the threads proceed past the barrier.

There are several twists, however, that you must implement:

- The codeword must be stored in `meetup.c` as some instance of `resource_t`. Therefore there will be some latency when reading and writing a codeword.
- The value of *n* is given as a command-line argument when starting up `myserver` (see below).
- Although the scenario described above has the codeword provided by the first Boba Fett/thread, it is possible (again via a command-line argument when starting up the server) to specify for some run of `myserver` that the *last* arriving Boba Fett/thread is the one providing the codeword shared amongst the N threads.

Here is an example where the value of *n* is given as 2 and the first thread arriving in a group of two Boba Fetts/threads provides the codeword:

```
$ ./myserver --meetup 2 --meetfirst
```

Here is an example where the value of *n* is specified to be 4 and the *last* thread arriving in a group of four Boba Fetts/threads provides the codeword:

```
$ ./myserver --meetup 4 --meetlast
```

The meetup provided by the server runs until it is terminated by Ctrl-C (i.e., the meetup having the given configuration can be re-used many times while `myserver` is running). A diagram in this assignment description shows how Boba Fetts interact with the meetup (i.e., via calls to `curl`).

Within `meetup.c` you are to complete the following two functions.

1. `join_meetup(char *value, int len)`: If appropriate for this call of `join_meetup`, use `write_resource()` to copy the contents of the char array referred to parameter *value* into the codeword resource variable. If too few threads have arrived to allow all to proceed, then block the caller of `join_meetup`. If enough have already arrived, then use `read_resource()` to copy the contents of the codeword resource into char array referred to by parameter `value`. (The `len` field is the size of the char array passed in as the first argument.)

2. `initialize_meetup(int n, int mf)`: Any code for initializing synchronization constructs or other shared data needed for the correct operation of your meetup solution must appear in this function. The first

parameter is the value of N to be used for all calls the `join_meetup`; the second parameter has one of two values: `MEET_FIRST` or `MEET_LAST`.

And here is how we communicate with the server. We will use *curl* as our client (i.e., we're depending upon the "GET" message provided via the HTTP standard implemented in *curl*). For example, assuming our server is running in one window and listening to port 11415, and where each client in the diagrammed example is in its own window, the client commands from the example (from top to bottom) would be:

- *bobafett 4*: `curl "localhost:11415/?op=meetup&val=solo"`
- *bobafett2:* `curl "localhost:11415/? op=meetup&val=luke"`
- *bobafett1*: `curl "localhost:11415/? op=meetup&val=vader"`
- *bobafett3*: `curl "localhost:11415/? op=meetup&val=anachro"`

Note the use of quotation marks around the argument to `curl`. These are needed as the ampersand symbol (&) has a special interpretation by the shell and we want to suppress that interpretation in order to permit an HTTP-protocol `GET` message that has two parameters.

One last detail: The server starts up a heartbeat thread that periodically prints a message ("<3"). This simply indicates that the server is still scheduling threads and isn't otherwise blocked.

You are only permitted to use POSIX `semaphore_t` in this part of the assignment (i.e., you may not use POSIX mutexes, condition variables, or any other POSIX Pthreads synchronization constructs other than semaphores). Your solution must also be *free of starvation*.

What you must submit

- All submission is via a git push of your `gitlab.csc` respository. This therefore must include:
  - Your changes to `vaccine.c` corresponding to your work for Part A;
  - Your changes to `meetup.c` and `meetup.h` corresponding to your work for Part B;
  - a file `a2/README.md` with details of your submitted solution.
  - **If you simply add, commit, and push all your work in a2/ then you will – in effect – have performed submission as requested.**

- Note that **you are not** permitted to change any other files without the express written permission of the instructor. When we evaluate your solutions, we will extract the four files mentioned in the previous bullet and will combine them with original version of the remaining files.

Evaluation

Our grading scheme is relatively simple.

- "A" grade: An exceptional submission demonstrating creativity and initiative. Solutions to Parts A and B run without any problems; description in Part C is also provided.

- "B" grade: A submission completing the requirements of the assignment. Solutions to Parts A and B run without any problems; description in Part C is also provided.

- "C" grade: A submission completing most of the requirements of the assignment. Solutions to Parts A or B run with some problems; Part C is either not provided or not sufficient.

- "D" grade: A serious attempt at completing requirements for the assignment. Parts A and B run with quite a few problems; Part C is either not provided or not sufficient.

- Less than 50% (i.e., and "F" grade): Either no submission given, or submission represents very little work. **Note that a non-working submission can receive a grade higher than 0%.**