Computer Science 360 – Introduction to Operating Systems
Summer 2018

Assignment #2

Due: Wednesday, June 27th, 11:30 pm by submission to conneX
(no late submissions accepted)

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 "Assignment #2" discussion forum on conneX. I will post my answers, clarifications, and possible corrections in the same forum.

Programming platform

For this assignment you must do your work using *linux.csc.uvic.ca*. You can remotely login to this machine anywhere on campus or from home by using *ssh*.

You may already have access to your own Unix system (e.g., Linux, macOS) yet we recommend you work *linux.csc.uvic.ca* rather than try to complete the assignment on your machine for later submission to conneX. Bugs in systems programming tend to be platform-specific, and something that works perfectly at home may end up crashing on a different hardware configuration. *Your code will be evaluated on linux.csc.uvic.ca, therefore you must ensure any work done on personal laptops or desktops also works correctly on that server.*

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.)

Goals of this assignment

1. Write a C program implementing a solution to the given *readers/writers problem*.

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

You will demonstrate your work for credit and explain your chosen data structures and algorithms that appear in the source code of your solution.

Preamble: *resource.c*

For this assignment you are provided a lot of code (in */home/zastre/csc360/a2*), such as that 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 your assignment are:

- *rw.c, rw.h*
- *meetup.c, meetup.h*

Both of the problems described later (i.e., *readers/writer* and *meetup*) will involve the reading and writing of "resources". Normally I would allow you to declare character arrays for this, but for this assignment we need to introduce a bit of latency into our read to memory and writes from memory. Therefore for Tasks 1 and 2 below 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 problems 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
```

Task 1: Readers/Writers using Pthreads

The readers/writers problem – and its various solutions as described in class and in the textbook – commonly occurs in systems programming and especially in client-server application architectures. We will experiment with a very simple server application in this assignment that accepts and processes *read* and *write* requests. That is, the server will receive read and write requests, will launch threads for servicing those requests, and these threads will call functions in *rw.c*. (Later I will explain how we make these requests of a server through the use of *curl*.)

Within *rw.c* you are to complete the following three functions.

1. *rw_read(char *value, int len)*: As long as there is no thread writing the resource, this function will read the resource variable *data* and copy it into *value* via a call to *read_resource()*. (The *len* parameter is the size of the character array passed in as the argument *value*.) If there are threads writing, then the thread calling *rw_read* must be blocked until the writer is finished.

2. *rw_write(char *value, int len)*: As long as there are no threads reading the resource, this function will write the value into the resource variable *data*. via a call to *write_resource()*. (The *len* field is the size of the character array passed in as the first argument.) If there are threads reading, then the thread that has called *rw_write* must block until the readers are finished.

3. *initialize_readers_writer()*: Any code for initializing synchronization constructs or other variables needed for the correct operation of your readers/writer solution must appear in this function. It is called from within *myserver.c.*

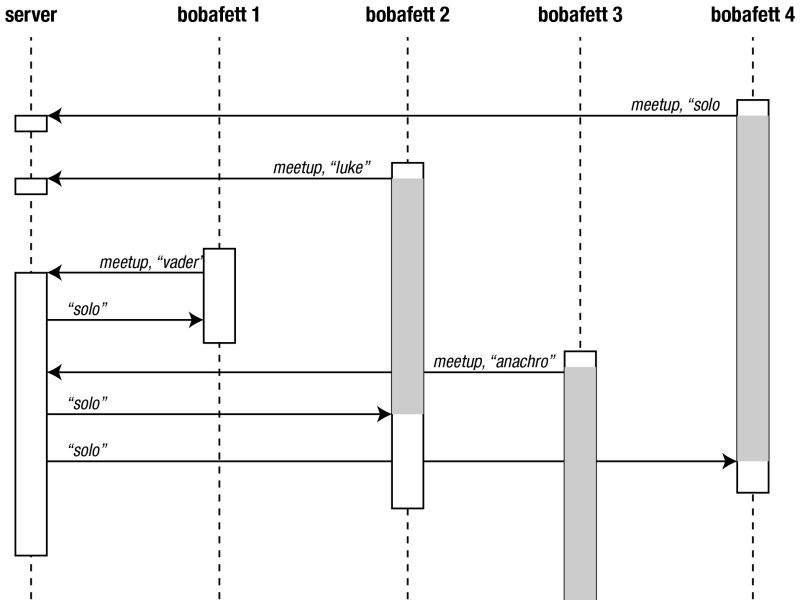
You can assume that there is a single variable in *rw.*c that will be read and written, and that it is named *data* (i.e., an instance of *resource_t* named *data* in *rw.c* has file scope).

You are only permitted to use POSIX *pthread_mutex_t* and POSIX *pthread_cond_t* in *rw.c* (i.e., you may not use POSIX semaphores or any other POSIX Pthreads synchronization constructs other than mutexes and condition variables).

Task 2: Meetup using Pthreads

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.)



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.

[1] This is, of course, a very fanciful scenario – hanging around downtown Victoria wearing cosplay masks might be awkward if not creepy.

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). On page 7 are details of 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.

You are only permitted to use POSIX *sem_*t semaphores in *meetup.c* (i.e., you may not use *pthread_mutex_t* or *pthread_cond_t* or any other POSIX Pthreads synchronization constructs). Your solution must also be *free of starvation*.

What is provided to you

Several C source-code files have been provided that implement the tricky bits of the server. In fact, the server behaves like a very simple HTTP server by listening to a specific port on the computer. Client requests will come in as HTTP messages, and replies from the server will be in the form of HTML text. Note that the code for receiving/parsing messages and sending back replies is already provided to you.

The files you are given are:

- ***meetup.c* & *meetup.h*: As described above.**
- *myserver.c* & *server.h*: This code already contains functionality for listening to a specific port for client requests, and calls the needed routines in *rw.c* and *meetup.c*.
- *network.c* & *network.h*: Used by *myserver.c* to do the TCP/IP dirty work; don't worry about the code in here (although in CSC 361 you'll learn better what it is doing). **You must change the value of COMM_PORT in *network.h* (see below), but otherwise no changes are permitted to these two files.**
- *requests.h*: Contains the declaration of an enumerated type used by the server to categorize requests. **You are *not* to change this file.**
- *resource.c & resource.h*: As described above.
- ***rw.c* & *rw.h*: As described above.**
- *makefile*: For building *myserver*. All of the needed compiler flags and libraries are included in the *makefile*. To compile and link *myserver*, type "make myserver". **You are not to change this file.**

These files are found on *linux.csc.uvic.ca* in */home/zastre/csc360/a2* and you can copy them into your own directory.

Mentioned above in the description of *network.c* & *network.h* was the requirement that you change *COMM_PORT* in *network.h*. Each student's server must listen to a different port. The port you must use is computed by taking the last four digits of your student number and then adding 10000. For example, if your student number was V00831415, then your value for COMM_PORT is 11415, i.e.,

```
#define COMM_PORT 11415
```

You must set *COMM_PORT* as indicated here; otherwise you and another student might accidentally use the same port (with less-than-hilarious results ensuing).

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 a GET message that has two parameters.

A similar URL format is used for the readers/writer part of the assignment.

- some reader: curl "localhost:11415/?op=read"
- some writer: curl "localhost:11415/?op=write&val=justin"

And just a reminder: The provided code does all of the work of retrieving and parsing the URLs received by *myserver*. The code in *myserver* will call your code in *rw.c* and *meetup.*c. Running *myserver* with no arguments will provide the readers/writer functionality in that server.

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.

What you must submit

- C source-code files named *rw.c*, *rw.h*, *meetup.c* and *meetup.h* which together contain your solution to Assignment #2.

- Any additional comments for the instructor in a file named *NOTES.md*.

- Submit these four (or five) files via the conneX link for the assignment. *Please do not submit ZIP files.*

Evaluation

Given that there are a variety of possible solutions to this assignment, the teaching team will not evaluate submissions by just using a marking script. Students will instead demonstrate their work to our course marker. Sign-up sheets for demos will be provided a few days before the due-date; each demo will require 15 minutes.

Our grading scheme is relatively simple.

- "A" grade: An exceptional submission demonstrating creativity and initiative. The *myserver* program runs without any problems.

- "B" grade: A submission completing the requirements of the assignment. The *myserver* program runs without any problems.

- "C" grade: A submission completing most of the requirements of the assignment. The *myserver* program runs with some problems.

- "D" grade: A serious attempt at completing requirements for the assignment. The *myserver* program runs with quite a few problems.

- "F" grade: Either no submission given, or submission represents very little work.