Programming Project 5

Synchronization

CS 441/541 – Fall 2020

Project Available		Oct. 26
Component	**Points**	**Due Date (at 11:59 pm)**
Proper use of Pthreads	5	
Proper input/output	5	
Problem solution	20	
Style & Organization	5	
Documentation	5	
Write-up	5	
Demo	5	
Total	50	Nov. 30
Submission	Group	Bitbucket PR

Objectives

In this project, you'll implement a novel synchronization problem as a way to learn how to construct a solution to a problem using Pthreads and semaphores.

Packaging & handing in your project

A template project is available on BitBucket. You ***MUST*** use the provided semaphore library as it works more reliably on various operating systems.

In order to form a group, use the appropriate tools in Canvas to create your group. If you have any trouble, let the instructor know.

You will submit your code as a pull request (PR) and set me (ssfoley) as a reviewer. In order to do this, be sure to follow our usual workflow for the class:

- **Fork** the template code.
- Give me **write permissions** on the repo. If you are working with a partner, give your teammate write access now, too.
- Create a **branch** called `dev` where you will do your work.
- Clone your dev branch and do the work.
- When you are ready to turn it in, create a **pull request from dev to master** on your fork. Please **make me a reviewer on the PR** so I can easily find your work.

Deliverables

◯ Code written in a **consistent style** according to the **Style Requirements** handout.
◯ Properly parses the command line arguments.
◯ Properly displays the state of the code.
◯ Properly synchronizes access to shared variables in the system.
◯ Your program **must never busy wait** for an event to occur. If you need to cause a thread to wait then it should do so using semaphores. **Do not assume** that the semaphore wait provides a proper queue for the waiting threads (if you need such a concept).
◯ Make sure to seed the random number generator before using it. Seed the random number generator before starting threads with the following command: `srandom(time(NULL))`

- ◯ You will need to put bounds on the random number generator by using the modulus operator on the output: `i = random()%LIMIT;`
- ◯ To sleep for less than 1 full second you must use the usleep() command (instead of sleep()) to sleep for some number of microseconds.
- ◯ When printing output to stdout you may notice that the output from multiple threads interleave themselves. To *fix* this problem you might consider creating a binary semaphore to protect calls to the `printf` function so only one thread is printing to the console at a time.
- ◯ You must print out the parsed command line parameters before starting execution. If an optional parameter is not supplied, then you must display the default value for that parameter.
- ◯ If the user provides an incorrect set of command line arguments to your program you must immediately display an error message and a message describing the correct use of your program. After displaying those messages your program will immediately exit.

Testing

Testing is an important part of the software development life cycle. You must describe in your documentation how you tested your project to ensure it met the requirements of the problem.

Note, that for synchronization problem just running the program one, twice, ..., 100, ..., 10,000 times **will not suffice** to highlight all of the possible ways that the threads can be interwoven during execution. Testing and debugging tend to be one of the greatest challenges to writing concurrent code because of this reality.

Your documentation must discuss the testing methodology that you used used to validate that your solution was valid in design and in the implementation of that design.

Pirates and Ninjas Problem

After an epic feud spanning years, the leaders from the pirate and the ninja forces met to negotiate a peace agreement. They agreed that the battles weakened their forces and reputations, creating a power vacuum. A new quarrel, between some melancholy vampires and a pack of werewolves, threatened to dominate the media and relegate the pirate v. ninja feud to obscurity. They could not let this happen.

While both sides needed peace to rebuild their forces, they could not let this become public knowledge for fear that it would make them look weak. If pirates or ninjas were even seen in public together, they must fight to the death. To complicate matters, both the pirates and the ninjas both used the same costume department. Before pirating or ninjaing (ninjasting?), a pirate or ninja must go through the costume department to acquire the proper garb, makeup, and even to rehearse lines. A pirate or ninja may need costume assistance multiple times a day.

The costume department has a fixed amount of space and helpers to help the costume changer. The default number of costume booths is two.

You must synchronize the pirates and the ninjas so that no ninja or pirate should ever have to fight to the death. You can assume that if any pirate is in the costume department, it will be forced to fight any ninja that enters (and vice versa). Of course, two pirates may safely use the costume department at the same time, as may two ninjas. You can assume the ninjas and pirates will only see each other if they are in the costume department. The pirate/ninja peace agreement requires that neither side may deprive the other of the costume department by always occupying the department.

You must implement a program that creates the requisite number of ninja and pirate threads, and use semaphores to synchronize the costume department. Each ninja and pirate thread should identify itself and which costume team it is using when it enters or leaves the costume department. Simulate a pirate or ninja costume preparation using the `usleep()` function call.

The following are the base requirements for your program:

- Pirates and ninjas cannot be in the costume department at the same time.
- If a pirate or a ninja is waiting it cannot be forced to wait for an indefinite amount of time.
- If a series of pirates arrive at the costume department one after another, then the first pirate to reach the costume department should be the first to enter followed by the second as so on. The same rule applies to a series of ninjas arriving to the costume department.
- Your program will take up to three arguments in the following order.
  1. Time to run in seconds. **Required argument**
     This is the length of time that your application will run before terminating itself.
  2. Number of Pirates. **Optional, default 5**
     The user should be able to specify any positive integer greater than 0.
  3. Number of Ninjas. **Optional, default 5**
     The user should be able to specify any positive integer greater than 0.
  4. Number of Costume Booths. **Optional, default 2**
     The user should be able to specify any positive integer greater than 0.
- While in the costume department, simulate costume preparation by sleeping the thread for a random amount of time between 0 and 50,000 microseconds.
- After leaving the costume department, simulate the time to wreak havoc by sleeping the thread for a random amount of *microseconds* between 0 and 1 second.
- Each pirate and ninja should print a message as it starts waiting for the costume department, enters, and leaves the costume department. Each pirate and ninja should display their allegiance, thread ID (integer number between 0 and the number of that type of thread), and action whenever their state changes.
- Before the application exits you must print, for each pirate and ninja, the number of times they entered and left the costume department.
- The number of pirate and ninja threads must be **dynamically allocated** based upon the command line input, and *not* have a fixed upper bound.

After You Finish Coding

Once you have completed your solution, in a ***special section*** within your documentation you must answer the following questions, **separately**:

1. Describe your solution to the problem (in words, not in code).
2. Describe how your solution meets the goals of this part of the project, namely (each in a separate section):
   - How does your solution avoid deadlock?
   - How does your solution prevent pirates and ninjas from being in the costume department at the same time?
   - How does your solution avoid depriving one side or the other access to the costume department? How is your solution "fair"?
   - How does your solution prevent "starvation"?
   - How does your solution preserve the order of pirates/ninjas waiting on the costume department?
   - How does your solution keep the costume booths relatively full? How does your solution have good resource utilization?
3. Describe how you verified that your implementation matched your design.
4. Describe how you tested your implementation.

To receive the full points you must provide a convincing argument as to why your solution is correct both in design, and in implementation of that design.

An example (the ... indicates output removed for brevity - you should display all state changes):

```
shell$ ./costume 10 3 2
Time To Live (seconds) : 10
Number of Pirates      : 3
Number of Ninjas       : 2
------------------------------
 Thread ID |   Action
-----------+-----------------------------------
 Pirate  0 | Waiting
 Pirate  1 | Waiting
 Ninja   0 | Waiting
 Pirate  0 | Costume preparation
 Pirate  2 | Waiting
 Pirate  1 | Costume preparation
 Pirate  1 | Leaving
 Ninja   1 | Waiting
 Pirate  0 | Leaving
 Ninja   0 | Costume preparation
 Ninja   1 | Costume preparation
 Pirate  0 | Waiting
 Ninja   0 | Leaving
 Ninja   1 | Leaving
 Pirate  2 | Costume preparation
 Pirate  0 | Costume preparation
...
 Pirate  1 | Waiting
 Pirate  1 | Costume preparation
-----------+-----------------------------------
 Pirate  0 : Entered   10 / Left   10
 Pirate  1 : Entered   15 / Left   14
 Pirate  2 : Entered    7 / Left    7
 Ninja   0 : Entered   11 / Left   11
 Ninja   1 : Entered   20 / Left   20
-----------+-----------------------------------
```