CS 3013 - Operating Systems

Project 3 (100 points) Assigned: Tuesday, February 5, 2019
Checkpoint: Tuesday, February 12, 2019
Due: Friday, February 15, 2019

# Project 3: Threads and Synchronization

The following two problems will give you a chance to practice synchronization. You must create threads for each of your actors and debug in a synchronous way. Each thread must use a degree of randomness for your scheduling; however, you may want to use the same seed value for your randomness initially to create reproducible code for debugging purposes.

You should use the `pthread` library. You must solve one of these synchronization problems with mutexes/condition variables (e.g., pthread_mutex_lock and pthread_mutex_unlock) and the other with semaphores (e.g., sem_wait, sem_post, and sem_init). You can decide which you use for each problem, but **you cannot use the same primitive for both**. Both can be solved with either, but one way may be more straightforward than the other.

## Problem 1: Pirates and Ninjas

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

While both sides needed a peace to rebuild their forces, they could not let this become public knowledge for fear that it would make them look weak. If pirates and ninjas are ever seen in public together, they must fight to the death. To complicate matters, both the pirates and the ninjas 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 (admittedly, the last of which does not take long for a ninja).

The costume department staffs 2–4 teams and each team can outfit one individual at a time. On any given day, the department will costume 10–50 pirates and 10–50 ninjas.

As the head of the costume department, it is your job to 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 pirate will only see each other if they are in the costume department (they have separate entrances/exits). The pirate/ninja peace agreement requires that neither side may deprive the other of the costume department by always occupying the department, so you must preserve fairness. It would be a shame if the peace agreement failed due to a wardrobe malfunction.

Each execution of your program represents the operation of the costume shop over a single day. Each pirate and each ninja is represented as a unique thread. You must implement code that creates the requisite number of ninja and pirate threads and use either semaphores or locks 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. For this problem, no time-of-day scheduling is allowed (e.g., ninjas in one half of each hour and pirates in the other half of each hour). All arrivals will be at random, and the amount of time that any individual takes to be outfit is also random. Though, you can expect that pirates will tend to take longer than ninjas, i.e., you should simulate this behavior by choosing appropriate random values. Simulate a pirate or ninja costume preparation using the `sleep()` call with a value drawn from random

distribution. A pirate or ninja may need costume assistance multiple times a day (ripped costumes are a natural consequence of rough-and-tumble work). At the end of their costume session, the ninja or pirate must inform the costume department if they will be back that day. This decision is made in the spur of the moment with a probability of answering "yes" being 25%.

During operation, the costume department should keep track of the usage statistics for proper billing; specifically, each pirate and ninja must spend 1 gold piece for every minute inside of the costume department (expensive, but you run a high-end shop after all). However, if any pirate or ninja has to wait for more than 30 minutes prior to entering the shop then you have to costume them for free. At the end of the day, you should calculate and print the itemized bills for both the pirates and the ninjas. For each ninja/pirate you should list the number of visits, the amount of time of each visit, the wait times, and the total gold owed to the costume department.

You also need to keep track of your own expenses profits. Each costuming team costs 5 gold pieces per day to staff. You should print also the amount of time that each team was busy, the amount of time each team was free, the average queue length, the gross revenue (amount of gold), the gold-per-visit (amount of gold divided by number of visits), and your total profits.

Once you have completed your solution, explain how your solution maximizes profits while not depriving one side or the other of the costume department. Your description should include the output of an example run and an analysis of your synchronization scheme in the context of that run. Save this in a text file, `problem1_explanation.txt`, accompanying your code.

## Command Line Arguments

For the purposes of simulation, assume that 1 second of execution represents 1 minute of time in the ninja-pirate world. Your program should take the following arguments.

- The number of costuming teams (min. 2, max. 4).
- The number of pirates (10–50).
- The number of ninjas (10–50).
- The average costuming time of a pirate. This is the average amount of time (in execution seconds) they spend in the costume shop.
- The average costuming time of a ninja. This is the average amount of time (in execution seconds) they spend in the costume shop.
- The average arrival time of a pirate. This is the average amount of time (in execution seconds) they spend adventuring before visiting the costuming department. Note, some individuals will visit multiple times.
- The average arrival time of a ninja. This is the average amount of time (in execution seconds) they spend adventuring before visiting the costuming department. Note, some individuals will visit multiple times.

## Random Number Distributions

Linux has several random number generators. `rand()` is the simple random number generator specified in *Kernighan and Ritchie*, and generates integers in the range `0..32767`. A better random number generator is *drand48()* , which generates uniformly distributed floating point numbers in the range `[0.0..1.0)`.[1]

[1]The thread-safety of random number generators has been debated extensively. See the following URL for a discussion: `http://www.evanjones.ca/random-thread-safe.html`. More important is that the multi-threaded nature of this assignment means that the sequence is not repeatable. To keep this assignment simple, you may ignore these issues.

Students from previous terms have used following method (called the Box-Muller transform) for generating random numbers with an approximate normal distribution. After generating two unit random numbers `a` and `b` (in the range `[0, 1)`), use this formula: `z = sqrt(-2 * log(a)) * cos(2 * pi * b)`, where `log(..)` is the natural logarithm function defined in `<math.h>`.

# Problem 2: Taming Massachusetts Drivers

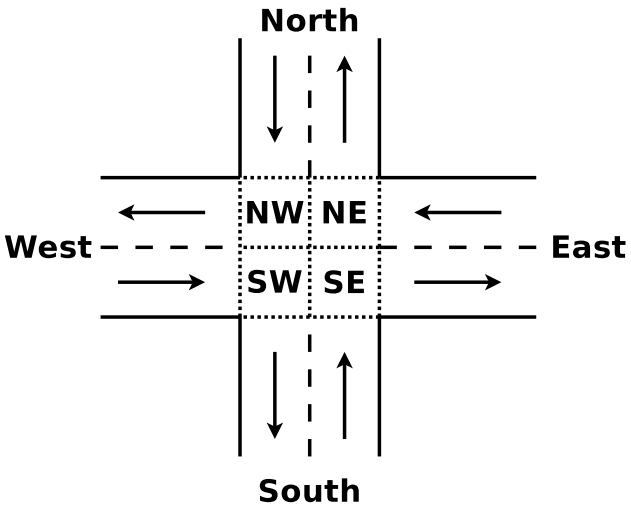
After years of accidents and high car insurance premiums, the state government has decided that they should actually enforce traffic laws instead of letting the Massachusetts drivers decide whose turn it is to go at an intersection. Apparently, these drivers were under the false impression that it was always their turn to go. The governor has charged you with designing a more efficient (and safe) intersection. To help, in your first phase, the governor is willing to ship off all the Massachusetts drivers and bring in some Minnesotan drivers, who are known to be more cooperative on the roadways.

## Modeling the Intersection

For the purposes of this problem we will model the intersection as shown above, dividing it into quarters and identifying each quarter by its nearest map directions. We assume that the drivers, even the Massachusetts ones, drive on the right side of the road and do not perform U-turns in the intersection. When using the intersection, a driver will use 1-3 quadrants, based on the type of use. For example, if a car approaches from the North, depending on where it is going, it proceeds through the intersection as follows:

- Right: NW quadrant only
- Straight: NW, then SW quadrant
- Left: NW, then SW, then SE quadrants

Before you begin coding, answer the follow questions in your `problem2_explanation.txt` text file:




1. **Phase 1:** Assume that the Minnesotan drivers follow the convention that whoever arrives at the intersection first proceeds first. Using the language of synchronization primitives, describe the way this intersection is controlled. In what ways is this method suboptimal?
2. **Phase 2:** The governor allows the Massachusetts drivers to return and sends the Minnesotans back home. Assume these drivers do not follow the convention described above. In particular, Massachusetts drivers are happy to enter an intersection, even if only the front bumper of their car will fit. In what instances can this mindset produce a deadlock? (It will be helpful to think of this in terms of the model we are using instead of trying to visualize an actual intersection). How can such a deadlock be prevented?

## Implementing your solution

With the model intersection, your solution must meet the following requirements:

- No two cars can be in the same portion of the intersection at the same time. (Even in MA, they call this an accident).
- Drivers will not pass each other going the same way. Each direction has a single lane of traffic and vehicles cannot pass each other in the intersection or in the line approaching the intersection (you can model this as a queue). If two cars both approach from the same direction, the first car to reach the intersection should be the first to go.
- Your solution must improve traffic flow without allowing traffic from any direction to starve traffic from any other direction.
- Your solution must maximize parallelism. If cars from multiple directions may safely use the intersection at the same time, they should do so. You cannot simply apply the rule "only one car may use the intersection at a time" since that would cause massive backups.
- Each car should print a message as it approaches, enters, and leaves the intersection indicating the car number, approach direction and destination direction.
- You may not simply turn the intersection into a roundabout (or otherwise change the model). The instructor has ample empirical evidence that motorists are simply incapable of using such systems.

You must create a simulator with 20 car threads. Each should be randomly assigned an approach direction and a random turn direction (left, right, or straight). You may recycle cars (reassigning a thread to represent a new car), thus allowing you to loop infinitely.

When you have completed your solution, explain how your solution meets the requirements described above in a text file, `problem2_explanation.txt`, accompanying your code.

## Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. However, as an example, any assignment in which one of the two synchronization problems is addressed will receive full credit towards the checkpoint. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

## Deliverables and Grading

When submitting your project, please include the following:

- All of the files containing the code for all parts of the assignment.
- Your `problem1_explanation.txt` explanation for the Pirates and Ninjas problem.
- Your `problem2_explanation.txt` explanation for the intersection problem.
- The test files or input that you use to convince yourself (and others) that your programs actually work.
- Output from your tests.
- A document called README.txt explaining your project, any defects, and anything that you feel the instructor should know when grading the project. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please only standard zip files for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://cerebro.cs.wpi.edu/cs3013/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://cerebro.cs.wpi.edu/cs3013/files.php`),
3. Complete your Partner Evaluation (URL: `https://cerebro.cs.wpi.edu/cs3013/evals.php`), and
4. Schedule your Project Demonstration (URL: `https://cerebro.cs.wpi.edu/cs3013/demos.php`), which may be posted slightly after the submission deadline.

A grading rubric has been provided at the end of this specification to give you a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback.

Groups **must** schedule an appointment to demonstrate their project to the teaching assistants. Groups that fail to demonstrate their project will not receive credit for the project. If a group member fails to attend his or her scheduled demonstration time slot, he or she will receive a 10 point reduction on his or her project grade.

During the demonstrations, the TAs will be evaluating the contributions of group members. We will use this evaluation, along with partner evaluations, to determine contributions. If contributions are not equal, under-contributing students may be penalized.

# Project 3 – Synchronization – Grading Sheet/Rubric

Evaluation?

Grader: _____ Student Name: _____ _____

Date/Time: _____ Student Name: _____ _____

Team ID: _____ Student Name: _____ _____

Late?: _____

Checkpoint?: _____ Project Score: _____

| Earned | Weight | Task ID | Description |
|---|---|---|---|
| _____ | 0% | 0 | Primitives – Students must use a different primitive for Part 1 and Part 2. If the same primitive is used for both, students may choose which part will be graded. The other part is to be assigned a score of 0. |
| _____ | 15% | 1 | Part 1 – Threads/processes implemented correctly. Prerequisite: Task 0. |
| _____ | 15% | 2 | Part 1 – Correct mutual exclusion. Prerequisite: Task 1. |
| _____ | 10% | 3 | Part 1 – Ensures fairness and proper explanation in problem1_explanation.txt. Prerequisite: Task 1. |
| _____ | 10% | 4 | Part 1 – Correct user-side program for testing. Prerequisite: Tasks 1, 2, 3. |
| _____ | 15% | 5 | Part 2 – Threads/processes implemented correctly. Prerequisite: Task 0. |
| _____ | 15% | 6 | Part 2 – Correct mutual exclusion and prevention of deadlocks. Prerequisite: Task 5. |
| _____ | 10% | 7 | Part 2 – Ensures fairness and proper explanations in problem2_explanation.txt. Prerequisite: Tasks 5. |
| _____ | 10% | 8 | Part 2 – Correct user-side program for testing. Prerequisite: Tasks 5, 6, 7. |

Grader Notes: