COMP 3500 Introduction to Operating Systems
Project 4 – Concurrent Computing: Cats and Mice

Long Version: 5.2
10/17/2019

Points Possible: 100
Submission via Canvas.

There should be no collaboration among students. A student shouldn't share any project code or solution with any other student. Collaborations among students in any form will be treated as a serious violation of the University's academic integrity code.

Requirements

- This project assignment has to be done individually by each student.
- You should accomplish the written exercises and submit `exercises.txt`.
- You must write a concurrent computing program to solve the cats-and-mice problem.
- You are allowed to discuss with other students to solve the coding problems at the design level (e.g., algorithms) rather than the programming level.

Objectives:

- To solve a synchronization problem using the mechanisms developed in project 3
- To build your programming skills in concurrent computing
- To improve your source code reading skills
- To strengthen your debugging skill

1. Project Goals

This project caters to you basic programming skills in modern concurrent computing. You will apply your synchronization mechanisms implemented in *Project 3* to solve a synchronization problem named *Cats and Mice*. On completion of this programming project, you are expected to demonstrate an ability to employ the *lock* and *CV* mechanisms to implement future concurrent computing programs (e.g., multithreading and multicore programming).

2. Getting Started

2.1 Setup `$PATH`

Like Project 3, this project requires you correctly configure `$PATH` using the command below. If you are using bash as your shell you should add the following line near the end of the `~/.bashrc` file.

```
export PATH=~/cs161/bin:$PATH
```

2.2 Git Repository and Project Configuration

This project is an extension of Project 3. You may continue maintaining the Git repository of project 3 as a repository of this project.

Important! Prior to working on this project, please tag your Git repository (See the following command below). The purpose of tagging your repository is to ensure that you have something against which to compare your final tree.

```
%git tag asst1b-start
```

Assuming that you are familiar with Git and GitLab, I intentionally omit detailed instructions on managing your source-code repositories. Please refer to Section 2.2 in project 3's specification for details on managing Git repository.

There is no need to configure project 4 because this project is an extension of project 3. In case you must reconfigure this project, you may refer to Sections 2.3 and 2.4 for instructions on configuring project and rebuilding os161.

2.3 Command Line Arguments to OS/161

As those in project 3, your implementation in this project will be tested by running OS/161 with command line arguments that correspond to the menu options in the OS/161 boot menu. Please don't change the OS/161 root menu option strings.

3. Concurrent Programming with OS/161 (This section is identical to that in Project 3)

If your code is properly synchronized, it is guaranteed that the timing of context switches and the order in which threads run will not change the behavior of your solutions. Although your threads may print messages in different orders, you can easily verify that they follow all of constraints applied to them; you can also verify that there is no deadlock.

3.1 Built-in Thread Tests

Important! When you booted OS/161 in project 2 (a.k.a., ASST0), you may have seen the options to run the thread tests. The thread test code makes use of the semaphore synchronization primitive. You should be able to trace the execution of one of these thread tests in GDB to see how the scheduler acts, how threads are created, and what exactly happens in a context switch. You should step through a call to `mi_switch()` and see exactly where the current thread changes.

Thread test 1 (" tt1 " at the prompt or `tt1` on the kernel command line) prints the numbers 0 through 7 each time each thread loops. Thread test 2 (" tt2 ") prints only when each thread starts and exits. The latter is intended to show that the scheduler doesn't cause any starvation (e.g., the threads should all start together, run for a while, and then end together).

3.2 Debugging Concurrent Programs

`thread_yield()` is automatically called for you at intervals that vary randomly. This randomness is fairly close to reality, but it complicates the process of debugging your concurrent programs.

The random number generator used to vary the time between these `thread_yield()` calls uses the same seed as the random device in System/161. The implication is that you can reproduce a specific execution sequence by using a fixed seed for the random number generator. You can pass an explicit seed into random device by editing the "random" line in your `sys161.conf` file. For example, to set the seed to 1, you would edit the line to look like:

```
28 random seed=1
```

Important! It is strongly recommended that while you are writing and debugging your solutions you pick a seed and use it consistently. Once you are confident that your threads do what they are supposed to do, set the random device to autoseed. This allows you to test your solutions under varying conditions and may expose scenarios that you had not anticipated.

4. Programming Exercises

4.1 Solving a Synchronization Problem using Semaphores or Condition Variables

This project offers you an opportunity to write a straightforward concurrent program and to get a more detailed understanding of how to use threads to solve synchronization problems. We have provided you with basic driver code (`catmousesem()` and `catmouselock()`) that starts a predefined number of threads. You are responsible for what those threads do.

Again, remember to specify a seed to use in the random number generator by editing your `sys161.conf` file. It is a lot easier to debug initial problems when the sequence of execution and context switches is reproducible.

When you configure your kernel for `ASST1`, the driver code and extra menu options for executing your solutions are automatically compiled in.

You must implement a solution for the casts-and-mice problem by choosing one of the following options:

- **Option 1** (Difficulty Level 3.0/5.0): a semaphore-based solution in `catsem.c` or
- **Option 2** (Difficulty Level 4.0/5.0): a condition-variable-based solution with locks in `catlock.c`

You should start your implementation by modifying the existing source code file named `catsem.c` (Option 1) or `catlock.c` (Option 2), which are located in the `src/kern/asst1` directory.

4.2 The Synchronization Problem: Cats and Mice

A professor has several cats and some mice that like to hang out in her house. The cats and mice operate need to work out a deal where the mice are allowed to steal bits of cat food, provided that the cats never see the mice actually doing so. If a cat sees a mouse eating from a cat dish, then the cat is obligated to eat the mouse.

You must implement the eating habits of cats and mice, which follow the synchronization policy below.

- no mouse ever gets eaten, and
- neither the cats or the mice starve.

Important! You must implement a solution (i.e., `catlock.c`) using the lock mechanism coupled with condition variables. Each cat and mouse thread should identify itself and which

dish it is eating from at the point when it begins and when it finishes eating. Simulate a cat or mouse eating by calling `clocksleep()`.

We make the following assumptions while solving this synchronization problem:

- there are two cat food dishes, 6 cats, and two house mice to be coordinated,
- only one mouse or cat may eat at a given dish at any one time,
- if a cat is eating at either dish, a mouse attempting to eat from the other dish will be seen and therefore eaten,
- when cats aren't eating, they will not see mice eating.

The driver code for the Cats-and-Mice problem is found in the following existing source-code file, the original version of which only forks the required number of cat and mouse threads.

`kern/asst1/catlock.c`

Important! There is no necessity of modifying `kern/asst1/catsem.c`, which is reserved for another solution using the semaphore mechanism.

4.3 Test Drivers

The two test drivers are the `catmousesem()` (for option 1) and `catmouselock()` (for option 2) functions, which are invoked in `menu()` (see Lines 501 and 502 in `menu.c`). You must modify these two driver functions in terms of for loops. For example, a *while* loop might be better than a *for* loop in the driver functions. Without finalizing these two drive functions, your cats and mice won't work. For option 1, you must modify `catmousesem()`. If you choose option 2, you should implement `catmouselock()`.

4.4 Written Exercises

Once you have completed your solution, answer the following questions in `exercises.txt` file.

Option 1:

- Explain how to avoid starvation in your implementation.
- Comment on your experience of implementing the Cats-and-Mice program. Can you derive any principles about the use of the *semaphore* synchronization primitives?

Option 2:

- Explain how to avoid starvation in your implementation.
- Comment on your experience of implementing the Cats-and-Mice program. Can you derive any principles about the use of the *lock and condition variable* synchronization primitives?

5. Deliverables

Make sure the final versions of all your changes are added and committed to the `Git` repository before proceeding. We assume that you haven't used `asst1b-end` to tag your repository. In case you have used `asst1b-end` as a tag, then you will need to use a unique tag in this part.

```
%cd ~/cs161/src
$git add .
%git commit -m "ASST1b final commit"
```

```
$git tag asst1b-end
%git diff asst1b-start..asst1b-end > ../project4/asst1b.diff
```

`asst1b.diff` should be in the `~/cs161/project4` directory. It is prudent to carefully inspect your `asst1b.diff` file to make sure that all your changes are present before compressing and submitting this file through `Canvas`. It is your responsibility to know how to correctly use `Git` as a version control system.

Important! Before creating a tarball for your project 4, please ensure that you have the following two files in the `~/cs161/project4` directory.

`exercises.txt` and
`asst1b.diff`

You can create a tarball using the commands below:

```
%cd ~/cs161
%tar vfzc project4.tgz project4
```

Now, submit your tarred and compressed file named `project4.tgz` through Canvas. You must submit your single compressed file through Canvas (no e-mail submission is accepted).

6. Grading Criteria

1) A lock-CV based solution in `catlock.c`: 70%
2) Adhering to coding and comment styles as well as documentation guidelines: 10%.
3) Written exercises (`exercises.txt`): 20%.

7. Make Your Code Readable

It is very important for you to write well-documented and readable code in this programming assignment. The reason for making your code clear and readable is two-fold. First, there is a likelihood that you will read and understand code written by yourselves in the future. Second, it will be a whole lot easier for the TA to grade your programming projects if you provide well-commented code.

Since there are a variety of ways to organize and document your code, you are allowed to make use of any particular coding style for this programming assignment. It is believed that reading other people's code is a way of learning how to write readable code. In particular, reading the OS/161 code and the source code of some freely available operating system provides a capability for you to learn good coding styles. Importantly, when you write code, please pay attention to comments which are used to explain what is going on in your system. Some general tips for writing good code are summarized as below:

- A little time spent thinking up better names for variables can make debugging a lot easier. Use descriptive names for variables and procedures.
- Group related items together, whether they are variable declarations, lines of code, or functions.
- Watch out for uninitialized variables.
- Split large functions that span multiple pages. Break large functions down! Keep functions simple.

- Always prefer legibility over elegance or conciseness. Note that brevity is often the enemy of legibility.
- Code that is sparsely commented is hard to maintain. Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."
- Backing up your code as you work is difficult to remember to do sometimes. As soon as your code works, back it up. You always should be able to revert to working code if you accidentally paint yourself into a corner during a "bad day."

8. Late Submission Penalty

- Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.
- Assignment submitted more than 3 days (72 hours) after the deadline will not be graded.

9. Rebuttal Period

- You will be given a period of one week (i.e., 7 days) to read and respond to the comments and grades of your homework or project assignment. The TA may use this opportunity to address any concern and question you have. The TA also may ask for additional information from you regarding your homework or project.