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

Short Version: 1.2
6/7/2022

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.

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`

```
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
```

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.

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

3.1 Built-in Thread Tests

Important! 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. 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
```

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.

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. 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`.
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
```

Submit your tarred and compressed file named `project4.tgz` through Canvas.

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%.