CS350 Operating Systems Spring 2014

Assignment One

This assignment has two parts. The first part, described in Section 1, requires you to implement several kernel synchronization primitives. The second part (Section 2) requires you to solve a thread synchronization problem.

Important: before you start working on this assignment, you should reconfigure and rebuild your OS/161 kernel:

```
cd kern/conf
./config ASST1
cd ../compile/ASST1
bmake depend
bmake
bmake install
```

All kernel builds for this assignment should occur in the `kern/compile/ASST1` directory.

1 Implement Kernel Synchronization Primitives

The OS/161 kernel includes four types of synchronization primitives: spinlocks, semaphores, locks, and condition variables. Spinlocks and semaphores are already implemented. Locks and condition variables are not – it is your task to implement them.

1.1 Implement Locks

Your first task is to implement locks. The interface for the lock structure is defined in `kern/include/synch.h`. Stub code is provided in `kern/threads/synch.c`. You can use the implementation of semaphores as a model, but **do not build your lock implementation on top of semaphores** or you will be penalized. In other words, your lock implementation should not use `sem_create()`, `P()`, `V()` or any of the other functions from the semaphore interface.

Locks are used throughout the OS/161 kernel. You will need properly functioning locks for this and future assignments to ensure that the kernel's threads are properly synchronized. Because of this, implementing locks correctly - though not difficult - is the most important part of this assignment. **Make sure that you get locks working before moving on to the other parts of the assignment.**

1.2 Implement Condition Variables

The second task is to implement condition variables for OS/161. The interface for the `cv` structure is defined in `kern/include/synch.h` and stub code is provided in `kern/thread/synch.c`. Each condition variable is intended to work with a lock: condition variables are only used *from within the critical section that is protected by the lock*

1.3 Testing Locks and Condition Variables

The file `kern/test/synchtest.c` implements a simple test case for locks, and another for condition variables. You can run the lock test from the kernel menu by issuing the `sy2` command, e.g,:

```
% sys161 kernel "sy2;q"
```

Similarly, the `sy3` command will run the condition variable test. If the lock test reports "Lock test done" without reporting any failure messages, it has succeeded. The output from the condition variable test should be self-explanatory.

Testing synchronization primitives like locks and condition variables is difficult. Both `sy2` and `sy3` are subject to false positives. In other words, an incorrect lock or condition variable implementation may pass these tests. However, if your implementation *fails* a test, there is definitely a problem. Since the synchronization tests are not perfect, we will use code inspection - in addition to testing - to evaluate your lock and condition variable implementations.

Although you are free to implement locks however you want, **you should not modify any of the kernel's test programs**, i.e., do not modify any of the files in `kern/test`. Furthermore, you should not make any changes to the way that the tests are invoked, e.g., do not change "`sy2`" to "`sy2a`".

2 Solve a Synchronization Problem

For this part of the assignment, you are expected to implement a solution to a synchronization problem called the "cats and mice" problem. The synchronization primitives that you may use in your solution are **semaphores, locks, and condition variables**. You are free to use whichever of these synchronization primitives you choose, however you like. However, **you must not directly use any "lower-level" methods of synchronization**, such as wait channels or spinlocks.

The Cats and Mice Problem

A number of cats and mice inhabit a house. The cats and mice have worked out a deal where the mice can steal pieces of the cats' food, so long as the cats never see the mice actually doing so. If the cats see the mice, then the cats must eat the mice (or else lose face with all of their cat friends). There are `NumBowls` catfood dishes, `NumCats` cats, and `NumMice` mice.

In OS/161, the cats and mice are simulated by some code in the file `kern/synchprobs/catmouse.c`. You are free to inspect the code in this file to understand how the simulation works. However, **you may not change this file in any way.**

The cat and mouse simulation works by creating one thread for each simulated cat, and one thread for each simulated mouse. Each cat thread repeatedly sleeps (blocks), and then tries to eat from one of the bowls. Each mouse thread behaves similarly. Each cat and mouse iterates a fixed number of times before terminating.

Your job is to synchronize the cats and mice so that the following requirements are satisfied:

R1: Cats and mice should never be eating at the same time.
In other words, if a cat is eating from any bowl, then no mouse should be eating from any bowl. Similarly, if a mouse is eating from any bowl, then no cat should be eating from any bowl. This will ensure that the mice don't get eaten by the cats.

R2: Only one mouse or one cat may eat from a given bowl at any one time.

R3: Neither cats nor mice should starve.
A cat or mouse that wants to eat should eventually be able to eat. For example, a synchronization solution that permanently prevents all mice from eating would be unacceptable. So would a solution in which cats were always given priority over mice. When we actually test your solution, each simulated cat and mouse will only eat a finite number of times; however, even if the simulation were allowed to run forever, neither cats nor mice should starve.

In addition to the above requirements, which are required for *correctness*, your solution should also be *efficient*, which we define as follows:

R4: Your synchronization mechanism should not impose an upper bound on the number of bowls that can be in use simultaneously.

In other words, if there are enough creatures, it should be possible for them to use all of the available bowls simultaneously.

2.1 Implementing Your Solution

In the directory `kern/synchprobs`, there is a file called `catmouse_synch.c`. Your solution to the "cats and mice" problem should be implemented entirely in this file.

The `catmouse_synch.c` file contains six functions, which are invoked by the cat and mouse simulation programs:

- `cat_before_eating`: called each time a cat eats, before it eats
- `mouse_before_eating`: called each time a mouse eats, before it eats
- `cat_after_eating`: called each time a cat eats, after it eats
- `mouse_after_eating`: called each time a mouse eats, after it eats
- `catmouse_sync_init`: called only once, before the cats and mice are created
- `catmouse_sync_cleanup`: called only once, after all cats and mice have finished

Implement your solution to the "cats and mice" problem by re-implementing these functions. In particular, you should use the `cat_before_eating` and `mouse_before_eating` functions to make creatures wait before eating, when waiting is necessary to satisfy the synchronization requirements. For example, if a mouse is eating and a cat attempts to eat, the cat must be prevented from eating (at least) until the mouse has finished, otherwise requirement R1 will be violated. To enforce this, your implementation of `cat_before_eating` should cause that cat (thread) to *block* until it is OK for it to eat. In other words, `cat_before_eating` should not return until it is OK for the cat to eat, since once it returns the cat will eat.

It is up to you to decide when it is OK for a cat (or mouse) to eat - this is the question that you must answer when designing a synchronization mechanism that satisfies the requirements.

You may use `catmouse_sync_init` to create or initialize any synchronization primitives or variables that your solution requires.

As provided to you, `catmouse_synch.c` implements a very simple default synchronization mechanism. This default mechanism satisfies all of the synchronization correctness requirements using a single semaphore, which is used as a lock. This default mechanism is *correct*, i.e, it satisfies R1, R2 and R3. However, it does *not* satisfy the efficiency requirement (R4) because only one creature can be eating at any time, regardless of how many bowls there are. You should replace this default mechanism with an implementation of a new mechanism that is both correct and efficient.

2.2 Testing

You can launch the cat and mouse simulation from the OS/161 kernel menu using the `sp2` command. In its short form, it takes 4 parameters, like this:

```
sys161 kernel "sp2 2 3 4 6;q"
```

These parameters specify the number of bowls, the number of cats, the number of mice, and the number of eat/sleep iterations each creature makes before finishing. Thus, the command above will simulate 3 cats and 4 mice eating from 2 bowls, with each cat and mouse iterating 6 times. A longer version of the command allows you to specify the amount of time each creature spends eating and sleeping. Issue the `sp2` command without parameters for see full usage instructions.

When the cat and mouse simulation program terminates, it prints several simulation statistics: the utilization of the bowls, and the average waiting times for cats and for mice. We will use these statistics to check whether your synchronization mechanism satisfies the synchronization requirements. In particular, when there are equal numbers of cats and mice, a synchronization mechanism that does not starve cats or mice should show similar waiting times for cats and mice. Also, for a given number of bowls, an efficient solution should show increasing bowl utilization if we run a series of simulations with larger and larger numbers of creatures.

3 What to Submit

You should submit your kernel source code using the `cs350_submit` command, as you did for Assignment 0. It is important that you use the `cs350_submit` command - do not use the regular `submit` command directly.

Assuming that you followed the standard OS/161 setup instructions, your OS/161 source code will be located in `$HOME/cs350-os161/os161-1.99`. To submit your work, you should run `cs350_submit 1` in the directory `$HOME/cs350-os161/`. The parameter "1" to `cs350_submit` indicates that you are submitting Assignment 1. This will package up your OS/161 kernel code and submit it to the course account.