Instructor: Ding Yuan
Course Number: ECE344

Operating Systems

ECE344, WINTER 2014
UNIVERSITY OF TORONTO

ASSIGNMENT 1: SYNCHRONIZATION

Release date: Jan 31
Due date: Feb 21, 11:59 pm.
TA responsible for this Lab: Yongle Zhang (zhangyongle dot kevin at gmail dot com).

Objectives

After this assignment, you should:

- Understand how OS161 implements semaphores.
- Be comfortable developing/implementing synchronization primitives.
- Be able to select an appropriate synchronization primitive for a given problem.
- Be able to properly synchronize different types of problems.

Introduction

In this assignment, you will implement synchronization primitives for OS161 and learn how to use them to solve several synchronization problems. Once you have completed the written and programming exercises you should have a fairly solid grasp of the pitfalls of concurrent programming and, more importantly, how to avoid those pitfalls in the code you will write later this term.

To complete this assignment you will need to be familiar with the OS161 threading code. The thread system provides interrupts, control functions, and semaphores. You will implement locks and condition variables.

Write readable code!

In your programming assignments, you are expected to write well-documented, readable code. There are a variety of reasons to strive for clear and readable code. Since you will be working in pairs, it will be important for you to be able to read your partner's code. Also, since you will be working on OS161 for the entire term, you may need to read and understand code that you wrote several months earlier.

There is no single right way to organize and document your code. It is not our intent to dictate a particular coding style for this class. The best way to learn about writing readable code is to read other people's code. Read the OS161 code, read your partner's code, read the source code of some freely available operating system. When you read someone else's code, note what you like and what you don't like. Pay close attention to the lines of comments which most clearly and efficiently explain what is going on. When you write code yourself, keep these observations in mind.

Here are some general tips for writing better code:

- Group related items together, whether they are variable declarations, lines of code, or functions.
- Use descriptive names for variables and procedures. Be consistent with this throughout the program.
- 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."

You and your partner will probably find it useful to agree on a coding style -- for instance, you might want to agree on how variables and functions will be named (my_function, myFunction, MyFunction, etc.), since your code will have to interoperate.

Begin your assignment

The very first thing you need to make sure is that you do not have any outstanding updates in

your Subversion tree. Use `svn update` and `svn commit` to get your tree committed. Make sure that you do not commit generated files.

Now, "tag" your Subversion repository. The purpose of tagging your repository is to make sure that you have something against which to compare your final tree. Again, make sure that you do not have any outstanding updates in your tree (such as the new files).

Now, tag your repository exactly as shown below.

```
% svn copy -m "starting assignment 1" $ECE344_SVN/trunk $ECE344_SVN/tags/asst1-start
```

Configure OS161 for ASST1

We have provided you with a framework to run your solutions for ASST1. This framework consists of driver code (found in `kern/asst1`) and menu items you can use to execute your solutions from the OS161 kernel boot menu.

You have to reconfigure your kernel before you can use this framework. The procedure for configuring a kernel is the same as in ASST0, except you will use the ASST1 configuration file:

```
% cd ~/ece344/os161
% cd kern/conf
% ./config ASST1
```

You should now see an ASST1 directory in the `compile` directory.

Building for ASST1

When you built OS161 for ASST0, you ran `make` from `compile/ASST0`. In ASST1, you run `make` from (you guessed it) `compile/ASST1`.

```
% cd ~/ece344/os161/kern
% cd compile/ASST1
% make depend
% make
% make install
```

If you are told that the `compile/ASST1` directory does not exist, make sure you ran config for ASST1.

Command line arguments to OS161

In order to execute the tests in this assignment, you will need more than the 512 KB of memory configured into System/161 by default. We suggest that you allocate at least 2MB of RAM to System/161. This configuration option is passed to the `busctl` device with the `ramsize` parameter in your `sys161.conf` file. Make sure the `busctl` device line looks like the following:

```
31 busctl ramsize=2097152
```

Concurrent programming with OS161

If your code is properly synchronized, the timing of context switches and the order in which threads run should not change the behavior of your solution. Of course, your threads may print messages in different orders, but you should be able to easily verify that they follow all of the constraints applied to them and that they do not deadlock.

Built-in thread tests

When you booted OS161 in ASST0, you may have seen the options to run the thread tests. The thread test code uses the semaphore synchronization primitive. You should 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 be able to 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 starvation -- the threads should all start together, spin for awhile, and then end together.

Debugging concurrent programs

`thread_yield()` is automatically called for you at intervals that vary randomly. While this randomness is fairly close to reality, 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. This means 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
```

We recommend 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 should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated.

Code reading

To implement synchronization primitives, you will have to understand the operation of the threading system in OS161. It may also help you to look at the provided implementation of

semaphores. When you are writing solution code for the synchronization problems it will help if you also understand exactly what the OS161 scheduler does when it dispatches among threads.

Thread questions

1. What happens to a thread when it exits (i.e., calls `thread_exit()`)? What about when it sleeps?
2. What function(s) handle(s) a context switch?
3. What does it mean for a thread to be in each of the possible thread states?
4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?
5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

Scheduler questions

1. What function(s) choose(s) the next thread to run?
2. How does it (do they) pick the next thread?
3. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

Synchronization questions

1. Describe how `thread_sleep()` and `thread_wakeup()` are used to implement semaphores. What is the purpose of the argument passed to `thread_sleep()`?
2. Why does the lock API in OS161 provide `lock_do_i_hold()`, but not `lock_get_holder()`?

Identifying synchronization problems

The following problems are designed to familiarize you with some of the problems that arise in concurrent programming and help you learn to identify and solve them.

Identify deadlocks

1. Here are code samples for two threads that use binary semaphores. Give a sequence of execution and context switches in which these two threads can deadlock.
2. Propose a change to one or both of them that makes deadlock impossible. What general principle do the original threads violate that causes them to deadlock?

```
semaphore *mutex, *data;
void me() {
  P(mutex);
  /* do something */
  P(data);
  /* do something else */

  V(mutex);
  /* clean up */
```

```
  V(data);

}

void you() {
  P(data)
  P(mutex);

  /* do something */

  V(data);
  V(mutex);
}
```

1. Here are two more threads. Can they deadlock? If so, give a concurrent execution in which they do and propose a change to one or both that makes them deadlock free.

```
lock *file1, *file2, *mutex;

void laurel() {
  lock_acquire(mutex);

  /* do something */

  lock_acquire(file1);
  /* write to file 1 */
  lock_acquire(file2);
  /* write to file 2 */
  lock_release(file1);
  lock_release(mutex);

  /* do something */

  lock_acquire(file1);
  /* read from file 1 */
  /* write to file 2 */

  lock_release(file2);
  lock_release(file1);
}


void hardy() {
  /* do stuff */

  lock_acquire(file1);
  /* read from file 1 */
  lock_acquire(file2);
  /* write to file 2 */

  lock_release(file1);
  lock_release(file2);
  lock_acquire(mutex);

  /* do something */

  lock_acquire(file1);

  /* write to file 1 */

  lock_release(file1);
  lock_release(mutex);
}
```

Coding synchronization primitives

We know: you've been itching to get to the coding. Well, you've finally arrived!

1. Implement locks

Implement locks for OS161. 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.

2. Implement condition variables

Implement condition variables for OS161. The interface for the `cv` structure is also defined in `synch.h` and stub code is provided in `synch.c`. Again, do not build your implementation using semaphores.

Solving synchronization problems

The following problems will give you the opportunity to write some fairly straightforward concurrent programs and get a more detailed understanding of how to use threads to solve problems. We have provided you with basic driver code that starts a predefined number of threads. You are responsible for what those threads do.

Remember to specify a seed to use in the random number generator by editing your `sys161.conf` file. It is much 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.

There are two synchronization problems posed for you. You must solve one problem using locks/CV's, and the other using semaphores. It is up to you to decide which problem to solve using which primitive, but you cannot use the same primitive for both problems. Both problems can be solved with either primitive, but one way may be more straightforward than another.

Synchronization problem 1: Of Mice and Cats

One of the esteemed professors in our department has a number of cats and (unfortunately) a number of mice that inhabit her 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 to all their cat friends).

There are two catfood dishes, 6 cats (not really, but it makes the problem more interesting), and two house mice.

Your job is to synchronize the cats and mice. No mouse should ever get eaten. You can assume that if a cat is eating at either food dish, any mouse attempting to eat from the other dish will be seen and eaten. When cats aren't eating, they will not see mice eating. Also, you may not starve either the cats or the mice. Only one mouse or cat may eat at a given dish at any one time.

Cats and mice must be able to eat from both dishes when possible. For example, the synchronization should not force the cats and mice to eat from only one dish at a time.

Also, make no assumptions about the time that a cat or mouse may spend outside the house (i.e., not eating). For example, the cats and mice should not eat in a round-robin order, or else a single cat who has gone for a walk will delay all others.

The driver code for the Pet Synchronization problem is found in the driver files `catsem.c` and `catlock.c`. **Please download the following two files:** `catlock.c` and `catsem.c` from the class web site. Then, replace the existing files in `kern/asst1` with these files. The new files update the definition of the `sem_eat()` and `lock_eat()` functions in the two files.

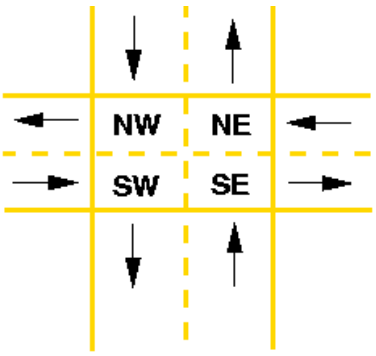
Right now the driver code only forks the required number of cat and mouse threads. Your job is to implement a solution using either semaphores or locks. When a cat or a mouse is eating, you need to call the `sem_eat` or `lock_eat()` functions that are provided. These functions help each cat or mouse thread identify itself when it begins and ends eating. While eating, the cat or the mouse thread sleep for a second by calling `clocksleep()`. The parameters of these functions are "cat" or "mouse", the cat number (0-5) or mouse number (0-1), the bowl number (1 or 2), and the iteration number (0-3, assume that each cat or mouse will eat four times).

Here is a sample output:

```
cat: 0 starts eating: bowl 1, iteration 0
cat: 0 ends eating: bowl 1, iteration 0
mouse: 0 starts eating: bowl 1, iteration 0
mouse: 1 starts eating: bowl 2, iteration 0
mouse: 0 ends eating: bowl 1, iteration 0
mouse: 1 ends eating: bowl 2, iteration 0
```

Synchronization problem 2: Podunk Traffic Problem

Traffic through the main intersection in the town of Podunk, KS (feel free to insert the name of your favorite small town) has increased over the past few years. Until now the intersection has been a four-way stop but now the impending gridlock has forced the residents of Podunk to admit that they need a more efficient way for traffic to pass through the intersection. Your job is to design and implement a solution using any synchronization primitives you have implemented.



Modelling the intersection

For the purposes of this problem we will model the intersection as shown above, dividing it into quarters and identifying each quarter with which lane enters the intersection through that portion. (Just to clarify: Podunk is in the US, so we're driving on the right side of the road.) Turns are represented by a progression through one, two, or three portions of the intersection (for simplicity assume that U-turns do not occur in the intersection). So if a car approaches from the North, depending on where it is going, it proceeds through the intersection as follows:

- Right: NW
- Straight: NW-SW
- Left: NW-SW-SE

Before you begin coding, answer the follow questions:

1. Assume that the residents of Podunk are exceptional and follow the old (and widely ignored) convention that whoever arrives at the intersection first proceeds first. Using the langauge of synchronization primitives describe the way this intersection is controlled. In what ways is this method suboptimal?
2. Now, assume that the residents of Podunk are like most people and do not follow the convention described above. In what one instance can this four-way-stop intersection 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).

Implementing your solution

We have given you the model for the intersection. The following are the requirements for your solution:

- No two cars can be in the same portion of the intersection at the same time. (In Podunk they call this an accident).
- Residents of Podunk do not pass each other going the same way. If two cars both approach from the same direction and head in the same direction, the first car to reach the intersection should be the first to reach the destination. Similarly, cars shouldn't “jump” over each other in the intersection. For example, say that a car enters the intersection and is going straight. Then another car enters the intersection from the same direction and is going left. The second car should exit the intersection after the first one. However, suppose that a car enters the intersection and is going left. If another car enters the intersection from the same direction and is going right, then it may leave the intersection earlier because the first car, even though it is ahead of the second car, may not yet be able to make the left turn.
- Each car should print a message as it approaches the intersection (`approaching`), enters one ore more regions of the intersection ((`region1`), `region2` and `region3`), and leaves the intersection (`leaving`), indicating the car number, approach direction and destination direction. You should call the `message` function that is provided for printing.
- Cars approaching an intersection from a given direction should enter the intersection in the same order. Note that there should be no synchronization before a car is `approaching` the intersection to slow it down. In other words, do not simply print `approaching` just before entering `region1` of the intersection, using the `message` function, e.g., there should be a synchronization primitive between the two events.
- There are no other ordering requirements. For example, there are no ordering requirements for cars approaching from different directions.
- Your solution should allow two or more cars to be in the intersection at a time, without allowing traffic from any direction to starve traffic from any other direction.

allowing traffic from any direction to starve traffic from any other direction.

The driver for the Podunk Traffic problem is in `stoplight.c` (a not so subtle hint about one possible solution). **Please download the following file:** `stoplight.c` from the class web site. Then, replace the existing file in `kern/asst1` with this file. The new file modifies the definition of the `message()` function.

The driver file consists of `createcars()` which creates 20 cars and passes them to `approachintersection()` which assigns each a random direction. We forgot to assign them a random turn direction; please do this in `approachintersection()` as well.

The file `stoplight.c` also includes routines `gostraight()`, `turnright()` and `turnleft()` that may or may not be helpful in implementing your solution. Use them or discard them as you like.

Here is a sample output for one car (car 8) that arrives from the East and is heading West. Note that the regions (e.g., `region1`) are defined relative to each car. A car going right will output `region1`. A car going straight will output `region1` and `region2`. A car going left will output `region1`, `region2` and `region3`.

```
approaching: car =  8, direction = E, destination = W
region1:     car =  8, direction = E, destination = W
region2:     car =  8, direction = E, destination = W
leaving:     car =  8, direction = E, destination = W
```

Preparing your assignment for submission

Finally, you need to prepare your code for submitting your assignment.

1. Once you are confident that you have completely done your assignment, run `make clean` from the `os161` directory. This will clean all generated files. Then use `svn status` in the `os161` directory to find out the status of all files. Any files that have a `?` before them are not in the repository. If you have created these files by hand, then add them to the repository using svn add. If these are generated files, use the instructions here to ignore these files.
2. Then run `svn commit` from the `os161` directory so that all modified files are checked in your repository. Use `svn status` in the `os161` directory again to make sure that all your modified source files are properly committed. Make sure that your partner's changes are also committed.
3. Tag your repository for the end of asst1. We will be using the start and the end tags to checkout your code for marking:

```
% svn copy -m "ending assignment 1" $ECE344_SVN/trunk $ECE344_SVN/tags/asst1-end
```

Testing your assignment with os161-tester

Please read the instructions for testing your code.