[CPEN 331. Fall 2019](https://sites.google.com/view/cpen331-fall2019/home)

**Assignment 3**

In this assignment you will solve a synchronization problem using the synchronization primitives in OS161, two of which you have implemented in the previous assignment. You will also learn about having multiple remotes configured for your git clone, and about branching and merging with git.

**Step 1. Prepare**

**Make sure you have a working implementations of locks and condition variables**

Before you begin working on this assignment, you must have a working implementation of the synchronization primitives that you completed in Assignment 2. If your implementation is not working, you are welcome to use the solution code that we distributed to you separately. (If you are not sure how to obtain it, ask your TA.) It is entirely up to you how you want to integrate the solution code into your tree. One option is to simply copy over the affected files. There is probably only two of them at this point: synch.h and sync.c. Another option is to use the [git reset](http://www.google.com/url?q=http%3A%2F%2Fgit-scm.com%2Fdocs%2Fgit-reset&sa=D&sntz=1&usg=AFQjCNHfaw1BOD6CT5cNqQDc5oGKHGS7yw) command to discard your commits and apply the patch to your tree, then commit and push the changes to git. Regardless of what you decide to do, make sure that your working tree compiles, runs and passes all the synchronization tests in the kernel menu (sy2, sy3 and sy4). If you need help, talk to your TAs during the lab session.

**Pull some new code required for this assignment**

We have created stubs and driver code for the synchronization problem you will be solving in this assignment. Follow these steps to integrate the code into your repository.

First, you need to add another remote to your git clone. The new remote is for the git repository set up by your instructor to share with you certain bits of the OS161 codebase. It is the same repository that you used to create your original master repository a couple of weeks ago. Go into the directory where you have the clone of your master repo (probably ~/os161/src) and type the following:

git remote add instructor http://dev.ece.ubc.ca/git/OS161

Next, examine the remotes your repository knows about by typing:

git remote -v

You will see the output that looks something like this:

instructor http://dev.ece.ubc.ca/git/OS161 (fetch)

instructor http://dev.ece.ubc.ca/git/OS161 (push)

origin https://username@bitbucket.org/username/os161.git (fetch)

origin https://username@bitbucket.org/username/os161.git (push)

As you can see, the new remote was added under the name "instructor". There is nothing magic about the name "instructor". It is simply the name we chose to use; you could give it any other name you wish. Now, if you want to fetch from that repository, you will refer to it by its name: 'instructor', in this case. (As a side note, if you use git commands without providing the name of the remote, git automatically defaults to the name "origin", which is the name assigned to your own master repository).

Now, fetch data from the new remote:

git fetch instructor

You will see that this command resulted in fetching of the two branches: master and synchprobs. Next, you need to integrate the code from synchprobs into your current tree. You will do this by merging the branch instructor/synchprobs into your master branch. (More information on git merging can be found [here](https://www.google.com/url?q=https%3A%2F%2Fgit-scm.com%2Fbook%2Fen%2Fv2%2FGit-Branching-Basic-Branching-and-Merging&sa=D&sntz=1&usg=AFQjCNGDpg813hLbmafxgqKEPpH0OigliQ).)

git merge instructor/synchprobs

At this point, git will probably complain about merge conflicts. Follow [this tutorial](https://www.google.com/url?q=https%3A%2F%2Fsites.google.com%2Fsite%2Fos161ubc%2Fworking-with-bitbucket%2Fhandling-merge-conflicts&sa=D&sntz=1&usg=AFQjCNEgFaqmyfztmaYkGYJSMCydASbuCg) to resolve them before proceeding.

If you now examine your source tree, you will see that there is a couple of new files that have been added, such as kern/conf/SYNCHPROBS and kern/synchprobs/airballoon.c. The first one is the configuration file that you will use for your assignment. The second one is the stub code for the synchronization problem you will solve. Make sure these files were successfully added to your source tree.

Now, push that new code into your master repository and tag your tree to indicate that you are beginning to work on Assignment 3:

git push

git tag asst3-start

git push --tags

**Step 2. Understand the problem**

Here is the synchronization problem you will have to solve. This is a well-known and a very serious known as AirBalloon.

After a war erupts in their kingdom, Princess Marigold must help Prince Dandelion (her younger brother) escape from danger. Marigold places Dandelion in a hot air balloon, which is connected to the ground by NROPES ropes -- each rope is connected to a hook on the balloon as well as a stake in the ground. Marigold and Dandelion must work together to sever all of these ropes so that Dandelion can escape. Marigold unties the ropes from the ground stakes while Dandelion unhooks the ropes from the balloon. To free a rope it must be either severed from the ground or unhooked from the balloon: not both.

Unfortunately, one of Princess Marigold and Prince Dandelion's enemy, Lord FlowerKiller, is also at work. FlowerKiller is rearranging the ropes to thwart Princess Marigold and Prince Dandelion. He will randomly switch ropes attached to two diffrent stakes. This leads to chaos!

Without Lord FlowerKiller's dastardly behavior, there would be a simple 1:1 correspondence between balloon hooks and ground stakes (each hook corresponds to exactly one stake, and each stake corresponds to exactly one hoop). However, while Lord FlowerKiller may break this symmetry. Say, Lord FlowerKiller decides to switch ropes connected to Stake 1 and Stake 5. Before the switch, the rope connected to Stake 1 on the ground is also connected to Hook 1 on the balloon, and the rope connected to Stake 5 on the ground is also connected to Hook 5 on the balloon. After the switch, however, one rope is connected to Stake 1 on the ground and to Hook 5 on the balloon, and vice versa.

As Marigold and Dandelion cut ropes, they must delete mappings, so that they remove all the ropes as efficiently as possible. Each character is represented by a thread. Dandelion selects ropes to sever by generating a random balloon hook index, and Marigold selects ropes by generating a random ground stake index.

Marigold and Dandelion must work as efficiently as possible, so one should not try to disconnect a rope that has already been disconnected. For example, if Dandelion already unhooked a rope, Marigold should see that the rope is hanging loose and not try to unook it from the stake. Similarly, Dandelion should not try to unook a rope that Marigold disconnected on the ground.

Lord FlowerKiller is on the ground, so like Marigold, he selects ropes by their ground stake index.

Due to recent advances in genetic technology, there is another unfortunate circumstance: Lord FlowerKiller figured out how to clone himself, so now there are N\_LORD\_FLOWERKILLER copies of of him at work (each is a separate thread)!

**To begin thinking about problems to avoid in your implementation, consider these examples:**

Marigold randomly selects Stake 7, sees that the rope is still attached. She marks the rope as severed. Suppose that the hook correpsonding to the severed rope is Hook 11. If Dandelion now randomly selects Hook 11, he must see that the rope is severed and not try to sever it again. ***What data structures might you use to avoid a race condition in this situation? What would you need to lock to make sure that Marigold and Dandelion don't sever the same rope twice?***

Worse yet, Lord FlowerKiller might be wreaking havoc with the same ropes. For example, imagine that he decides to swap the rope attached to Stake 7 with the rope attached to Stake 4, and vice versa. ***How do you make sure to avoid race conditions that this situation creates?***

Now consider how Lord FlowerKillers might step on each other toes. Suppose Lord FlowerKiller1 grabs Stake1 and Stake4 to swap the corresponding ropes, and Lord FlowerKiller2 graps Stake4 and Stake1. ***How might this cause deadlock? What would you do to prevent it?***

In addition to our character threads, there is also a Balloon thread that does not do much. It gets started in the beginning of the program, at the same time as the other threads, and just sits there and waits until all the ropes have been severed. Then it announces that Prince Dandelion escaped and exits.

***Side note:*** *You may be wondering why you have to solve a synchronization problem that seems to be completely unrelated to the code in the OS kernel. Here is why. As you will realize in the future programming assignments, which all directly involve programming kernel services, proper synchronization among threads is essential. You will not get far if you are unable to properly synchronize the code in the scheduler, virtual memory system and the file system. This exercise lets you practice concurrent programming and the use of synchronization primitives in a less challenging environment.*

**Step 3. Understand the requirements**

Your solution must satisfy these conditions:

* Avoid deadlocks and race conditions
* Stick to the API provided in the driver code. You may write any other code you want, but do not change or go around the code already provided to you. If unsure, as your TA!
* You may use semaphores, locks and condition variables available in OS161, but do not use spinlocks, wchans, or atomic primitives directly, and don't turn on/off the interrupts.
* Permit Marigold, Dandelion and Lord FlowerKiller threads to operate concurrently (no "big lock" solutions)
* The Balloon thread exits after all ropes have been severed
* The Main thread (the one that starts all other threads) exits after all threads are done.
* You must deallocate all the memory allocated by your code before all threads exit. If you don't, there will be a memory leak! We will run your program multiple times to make sure your kernel does not leak.
* Marigold must access the ropes via stakes. She does not have access to hooks. Programmatically, this means that you need to have some kind of data structure representing stakes, which Marigold indexes. Marigold can update stakes and ropes (that are connected to stakes), but she may not update hooks. Similarly, LordFlowerkillers may access ropes via stakes only. Dandelion, on the other hand, may access ropes only via hooks. He cannot see or access stakes.

To grade your code, we will read it to check that it meets the specification and use an automated script to check the correctness of the output. Please read this section very carefully to understand what your code should print. Be sure to follow these rules. Otherwise our script may report your solution as incorrect. You don't want to lose points for small typos!

* Every time Marigold severs the rope, the Marigold thread prints: "**Marigold severed rope N from stake K**", where N