CS350 Operating Systems Winter 2022

A2 - Assignment Specification

1 Introduction

In this assignment, you will implement kernel synchronization primitives for OS/161 as your A2 kernel space programming component. The A2 `userspace` programming assignment objective is to use POSIX Pthread API and its synchronization primitives to create multi-threaded programs.

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

In the `linux.student` environment, this is done by:

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

All OS/161 kernel builds for this assignment should occur in the `kern/compile/ASST1` directory in the `linux.student` environment.

In the `cs350-container`, you can configure and build the kernel by:

```
build_kernel ASST1 //configure and build
cd /os-compile/
sys161 kernel-ASST1 //run the kernel
```

It is important to note that for OS/161 Kernel space programming assignment, we will use the `ASST1` assignment directory, even though this is assignment 2. We are using the configuration from previous terms, at which time, this was assignment 1, hence the tag `ASST1`. The userspace programming assignment will be `ASSTUSER2`.

Some general advice for this assignment:

- **Start early.** The instructions are detailed, but even debugging simple mistakes are time consuming. This holds doubly true if you are not especially familiar with `C`.
- **Compile often.** By checking whether the code compiles after every programming prompt, explicitly specified in the following sections, you will be able to pinpoint problems very quickly. The code **should** compile after every properly implemented programming prompt.

In this assignment, for the A2 - OS/161 kernel programming componenet, we will use two prompts.

Explore prompts: guide you towards a better understanding of the OS/161 kernel code.

Programming prompts: give you step by step implementation instructions for implementing synchronization primitives in OS/161 kernel. **Read these carefully!**

2 Pre-lab A2 - Code Review OS/161

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.

The relevant files for this assignment are:

- `os161-1.99/kern/include/synch.h`
- `os161-1.99/kern/thread/synch.c`

2.1 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.**

2.2 Condition Variables

The interface for the condition variables for OS/161 are in 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.*

3 A2: OS/161 Kernel Side Programming

In this section, we will guide you through the OS/161 synchronization primitive programmig assignment for A2.

3.1 Implementing `locks`

The `lock` structure in OS/161 currently only has a single member, the name of the lock.

```
struct lock {
        char *lk_name;
        // add what you need here
        // (don't forget to mark things volatile as needed)
};
```

Listing 1: `lock` structure in `os161-1.99/kern/include/synch.h`

We will need to complete the following operations on `locks`:

`lock_create` - When the lock is created, no thread should be holding it.

`lock_destroy` - When the lock is destroyed, no thread should be holding it.

`lock_acquire` - Get the lock. Only one thread can hold the lock at the same time.The call to `lock_acquire`, should be a blocking call, so that if the lock cannot be acquired, because it is unavailable, lock acquire should cause the thread to block, until the lock is available.

`lock_release` - Free the lock. Only the thread holding the lock may do this.

`lock_do_i_hold` - Return true if the current thread holds the lock; false otherwise.

It is important that these operations be atomic. A spinlock should be used to ensure atomicity of `lock_acquire` and `lock_release`. To enable blocking, if the lock cannot be acquired, lock acquire must release the spinlock and cause the thread to block in the `wchan`.

Programming: To implement these operations on the `locks`. We will need to introduce additional members to the `lock` structure

- **a `lk_owner` which will be the `thread` that owns the lock**
- **a boolean variable, `lk_held`, to indicate whether the lock is available or not**
- **a `wchan`, `lk_wchan`**
- **a `spinlock`, `lk_spnlk`**

A simple way to uniquely identify a thread is to use the address of the thread structure. OS/161's `curthread` global variable points to the thread structure of the currently running thread.

Programming: Implement `lock_create` and `lock_destroy` functions.

- **create the `wchan` and initialize the `spinlock`: use the `sem_create` and `sem_destroy` functions to help you create the `lk_wchan` and initialize the `lk_spinlock` in `lock_create` and `lock_dstroy`**
- **initialize the `lk_owner` to NULL, and**
- **initialize `lk_held` to false**

Remember you should use kmalloc and kfree, respectively. Everything allocated in `lock_create` should be freed in `lock_destroy`, including the lock structure itself.

To implement `lock_acquire`, the pseudo code is presented below:

```
lock_acquire(lock *lk) {
   KASSERTIONS( ... lock not null, don't already own lock ... )
   spinlock_acquire( lk->spin)
   while ( lk->held ) {
          wchan_lock(lk->wchan)
          spinlock_release( lk->spin )
          wchan_sleep(lk->wchan)
          spinlock_acquire( lk->spin)
   }
   lk->held = true
   lk->owner = curthread                        //curthread is current thread
  spinlock_release( lk->spin )
}
```

Listing 2: Lock Pseudocode.

Explore: You are encouraged to think about the following concepts:

- What is the purpose of the spinlock used by a lock?
  - The lock structure has a field which indicates if the lock is available or not.
  - To take the lock, a thread must test-and-set the field which holds the lock's availability
  - We would have a race condition because the test-and-set operation requires mutual exclusion.
  - We can use a spinlock, instead of assembly, to protect the critical section.
  - The spinlock is only held for a brief amount of time — only for the test-and-set.
- Why we DO NOT block a thread (put it to sleep) while it owns the spinlock!

- Why a loop? When a thread, say T_B, is unblocked and continues execution there is a possibility that the lock was taken by another thread, T_A, that was scheduled to run before this thread, T_B, could take the lock. Therefore after being unblocked it needs to ensure that lock is still available. Hence, the test is in a loop.
- Why must you release the spinlock prior to calling `wchan_sleep`?
  - A thread that calls `wchan_sleep` will block.
  - If that thread owns the spinlock when it blocks, then no other threads may acquire that spinlock. Those threads will end up spinning — instead of blocking
  - Must release spinlock prior to blocking to ensure that threads attempting to acquire the lock do not spin on the spinlock
- Why should you lock the wait channel prior to releasing the spinlock? So that you can ensure that there are no threads left on the wait channel. A thread that is going to sleep must acquire the wait channel lock (using wchan lock) before releasing the spinlock and sleeping. Otherwise, the thread may fail to receive a wakeup signal when the lock is released.

The correct implementation of `locks` in OS/161 should enforce some constraints on how locks are used. In `lock_release`, a lock can not be released by a thread that is not holding the lock. This means that a thread can't release the lock before it acquires the lock and it can not release a lock that is held by another thread.
A common question is how to handle these types of problems. Some possible options are:

- Do not release the lock but do not report an error. This is the least desirable option since a programmer calling lock release in an incorrect way will not be notified of the error.
- Crash the kernel using `panic()`
- Use `KASSERT()` to test for and catch the error

Between `panic()` and `KASSERT()`, `KASSERT` is preferred, since `lock_acquire()` and `lock_release()` will be called within the kernel, so there is a high degree of confidence that they will be called correctly. If they are not there is a bug in the kernel and we want to be sure to fix the bug.

Programming: Implement the `lock_release` function to:

- **assert that the lock is not null,**
- **assert that the current thread does indeed hold the lock**
- **acquire the spinlock in the lock `lk_spnlk` to atomically set:**

  - **the owner to NULL**
  - **the `lk_held` to false;**

- **wake up one or all waiting threads and then release the spinlock**

Programming: Implement `lock_do_i_hold`:

- **assert that the lock is not null**
- **test the identifier of the calling thread against the identifier of the lock holder. If the stored lock holder is a simple value that will fit into a register, e.g. a pointer to a thread structure, then it is not necessary to acquire the spinlock to read that value.**

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"
```

If the lock test reports "Lock test done" without reporting any failure messages, it has succeeded. Testing synchronization primitives like locks and condition variables is difficult. `sy2` is 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.
You can test with sy2 and uw1 tests. We run each of these two tests using several system configurations. A successful output from the `sy2` test should look like:

```
Starting lock test...
cleanitems: Destroying sems, locks, and cvs
Lock test done.
```

If the test produces other output, likely gibberish, it has failed.
A successful output from the `uw1` test should look like:

```
Starting uwlocktest1...
value of test_value = 0 should be 0
TEST SUCCEEDED
cleanitems: Destroying sems and locks
uwlocktest1 done.
```

If the test doesn't work it will report something like TEST FAILED.

3.2 Implementing Condition Varaibles in 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.
We will need to implement the following operations on `cvs`:

`cv_create` - initialize the `cv` struct members

`cv_destroy` - destroy the `cv` struct members

`cv_wait` - Release the supplied lock, go to sleep, and, after waking up again, re-acquire the lock.

`cv_signal` -Wake up one thread that's sleeping on this CV.

`cv_broadcast` -Wake up all threads sleeping on this CV.

Important: You should read the `wchan` operations in `kern/include/wait.h` to see if you have to call `kfree` for `wchan`.
For all three operations `cv_wait`, `cv_signal`, and `cv_broadcast`, the current thread must hold the lock passed in. Note that under normal circumstances the same lock should be used on all operations with any particular CV. These operations must be atomic.

Programming: To implement these operations on `cvs`. We will need to introduce an additional member to the `cv` structure

- **a `wchan`, `cv_wchan`**

Programming: Implement `cv_create` and `cv_destroy` functions.

- **create and destroy the `cv_wchan`: use the `sem_create` and `sem_destroy` functions to help you**

The OS/161 API for `cv_signal` and `cv_broadcast` have both a `lock` and a `cv` as parameters. The documentation of the two functions indicates that "you should own the lock passed into the function". The lock, which protects access to the shared variable, is used to both check the condition, and, modify it. This is to prevent a race condition where multiple threads might read and write that shared variable at the same time.
It makes no claim that the owned lock has to be the one used to protect that shared variable/condition. You could pass any lock that you own into those functions. Doing so would have no impact on how we use the cv, i.e., we would still want to check (in most cases) that the condition is met upon wake.

Explore: Why might `cv_signal` want a pointer to the lock?
Briefly, because you should be the owner of the lock that protects the global condition before you signal that the condition has been met.

1. When you call `cv_wait`, some global condition is not being met. This happens inside of a lock (which ultimately, is used to protect that global condition).

2. Any thread that changes that condition will require that same lock to make that change. So, the thread that makes the condition true, will do so inside of the critical section protected by that one lock.

3. Once a thread makes the condition true, it should immediately signal — because at that EXACT moment in time, the condition is true. This means the signal should happen inside of the critical section protected by the lock. If you were to signal outside of this critical section there is a chance another thread will have changed the condition value prior to your signalling (which is undesirable).

Hence, the signalling thread should be the owner of the lock passed in.
Side Note: various implementations of Mesa-style CVs do NOT do this, but they are not broken. The waiting thread, once returned to the critical section, simply needs to re-check the condition before proceeding — which you may note, we usually do anyway.

Programming: Implement `cv_signal` so that:

assert that `cv`, `lock` are not NULL

assert that the thread calling signal does own the `lock`.
Hint: you implemented a function called `lock_do_i_hold`

it must unblock exactly one thread waiting on the condition variable, assuming that there is at least one such thread. This can be implemented easily using `wchan_wakeone`.

Programming: Implement `cv_broadcast` so that:

assert that `cv`, `lock` are not NULL

assert that the thread calling signal does own the `lock`.
Hint: you implemented a function called `lock_do_i_hold`

it must unblock all threads waiting on the condition variable. A call to `wchan_wakeall` is the simplest way to do this.

Since the `cv` functions are called from within a critical section protected by a lock, it is not necessary for CVs to have their own spinlocks for enforcing atomicity. However, since `cv_wait` releases and re- acquires the lock, it must be handled carefully. It must acquire the wait channel lock via `wchan_lock` before releasing the lock. Otherwise, there is a danger that the waiting thread may miss a subsequent `cv_signal` or `cv_broadcast`.

Programming: Implement `cv_wait`, so that you:

- **assert that `cv`, `lock` are not NULL**
- **assert that the thread calling signal does own the `lock`.**

- **Iacquire the wait channel lock, using `wchan_lock`**
- **release the lock**
- **sleep on the wait channel**
- **reacquire the lock after waking up**

When you are using condition variables and locks for synchronization, it is important to realize that when a shared variable is modified, it could change the condition that other threads are waiting on, so we may want to signal or broadcast those threads. This should be done right after modifying that variable.
Why? Because at that exact moment in time we know for certain that the condition has been met. By calling signal or broadcast, we guarantee that at the time the waiting threads are woken, the condition is true. However, that does not mean that the woken thread(s) will be able to proceed in their critical section.
When a thread is woken, recall that it does not immediately run. Woke threads are placed onto the ready queue , and the OS will choose when those threads will run. It could be right away, but it could also be the distant future. Hence, there is a possibility, that after the signaling thread (which modified the condition) releases the lock that another thread is then able to modify the condition before the woke thread is able to proceed! This is why we often check the condition in a while loop (the same is true for the semaphore `P` and `lock_acquire` functions).
The file `kern/test/synchtest.c` implements a simple test case for locks, and another for condition variables. You can run the cv test from the kernel menu by issuing the `sy3` command, e.g,:

```
% sys161 kernel "sy3;q"
```

The output from the condition variable test should be self-explanatory. Successful output should look like this (repeated several times):

```
Starting CV test...
Threads should print out in reverse order 5 times.
Thread 31
Thread 30
...
Thread 0
```

Testing synchronization primitives like locks and condition variables is difficult. `sy3` is 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.

4 A2-userspace Programming Requirements

In A2-usersapce you are required to implement a multithreaded program and use synchronization primitives. This means working with threads, mutexes, and condition variables.

4.1 Starter Code for A2-userspace Programs

In A2-userspace, there are two different questions that need to be answered. In question 1, which we will refer to as, a2q1,you are required to implement a multithreaded function that search for an article in a library, similar to the way, you might search for a word in a file. Your multithreaded implementation should meet a two-fold speedup against the single threaded version that is already implemented.

In question 2, a2q2, you are required to use synchronization primitives to implement a solution to a producer-consumer problem. In this problem, there is a resource that is computed by producers and consumers. There is a capacity on the resource that must be maintained by ensuring the ratio of the number of producers and consumers that can use the resource at the same time. You are encouraged to read the code to understand the context clearly.
For the A2-userspace we have provided you with starter code. The folder src contains two subfolders a2q1 and a2q2. Download the folder from the course website under the A2 starter code. The files in this folders are as follows:

a2q1 contains the following files:

- data.h and map.h - header files containing data definitions for articles and functions, respectively.
- main.c - contains the SingleThreadedWordCount function that searches for the total number of occurrences of an article in the library
- map.c - contains the MultithreadedWordCount function, which you have to **implement** to conduct a multithreaded search in the library for the number of occurences of the article.

The solution needs to find the same number of occurrences as the single threaded version in main.c, and provide the two-fold speedup.

a2q2 contains the following files:

- assignment.h and structs.h: include the header files for the header for the functions and resource
- orderme.c: the main function that creates the producer and consumer threads and computes the resource

- `assignment.c`: contains the functions that you must complete implementing so that the consumers and producers can compute the `resource` while maintaining the resource capacity.

`Makefile`: each folder contains a `Makefile` to build and run the program files

For A2-userspace assignments, you are only allowed to change `map.c` and `assignment.c` for a2q1 and a2q2, respectively.Edits to other files will not be recognized as we will use a fresh copy of the source code and place your implementation in it.

4.2 Multithreaded Program

In this assignment, we introduce threads and the pthread API. Each process can have multiple threads. Threads have their own set of CPU registers and stack segments, but share the code and heap segments with each other. Since each thread can potentially run on a separate CPU, multithreading lets us use more than one core at a time to speed up computation.
The API used by C to create threads is the POSIX threads (pthreads) API. The main three functions this API provides are:

`pthread_create`: This call creates a new thread. The thread starts exe- cuting from the function given as an argument.

`pthread_exit`: This call destroys the thread, but not the process. The exiting thread may pass a pointer to a variable in the heap as an exit value, to be read by another thread using `pthread_join`

`pthread_join`: Wait for another thread in the same process to be done.

A common pattern with multithreading is to spawn multiple workers using `pthread_create` from an initial thread, then wait for them to be all done by repeatedly calling `pthread_join`. The initial thread communicates to each worker what data it needs to process by passing to each thread different arguments; this makes it very easy to parallelize tasks where each thread needs to only work on part of the data at a time.

For this question we parallelize exactly this kind of workload. We are given a "library" of news articles, each of which is composed of a sequence of words. The task we need to parallelize is counting the number of times a certain word occurs in all articles. To count the number of occurences we traverse the library one article at a time, and one word at a time. We compare each word against the one we are looking for, and increment a counter if they are identical.

The task is trivially parallelizable because we can process each article separately. That means that we can create an arbitrary number of threads, split the work between them, and gather all the individual

counters using `pthread_join`. There is no need for synchronization since each thread has its own input.

For this question, fill in the function in the file `map.c`. The solution needs to find the same number of occurrences as the single threaded version in `main.c`, and provide considerable speedup.

To run `a2q1` you will require the number of articles to create, a seed and the number of threads to use for the multi-threaded search.

```
% ./a.out
Usage: a2q1 [NUMARTICLES] [SEED] [NUMTHREADS]
```

This is a sample of the output, without a correct solution to the multi-threaded version, with a small number of articles.

```
% ./a.out 30 3 4
Parallelizing with 4 threads...
ERROR: Single threaded version found 1584 occurences.
ERROR: Multi threaded version found 0 occurences.
ERROR: Please check for race conditions or other bugs.
```

Asample of the output, with a multi-threaded solution that is not correct:

```
% ./a.out 300 3 100
Parallelizing with 100 threads...
ERROR: Speedup is 1.260673, less than 2.000000
ERROR: Single Threaded is 10.919990s, mulithreaded is 8.662032s
ERROR: Please fix any bottlenecks in the code.
```

A sample of the output, with a correct solution to the multi-threaded version.

```
% ./a.out 30 3 100
Parallelizing with 100 threads...
Found 1584 occurences of abc.
```

4.3 Mutex and CVs

The two main synchronization primitives of the pthread library are mutexes and condition variables.Mutexes function like spinlocks in that they provide mutual exclusion, but are more efficient: A thread waiting on a mutex will be removed from off the processor, leaving it free for another thread that can actually do work. A thread waiting on a spinlock on the other hand will continuously try to take the mutex (lock) by continuously executing the lock function until it receives it. This leads to wasted CPU cycles if the thread that is already in the critical section holdsthe spinlock for a long time.

Condition variables are used to deal with race conditions that arise when multiple threads attempt to grab the same mutex. For example, assume that we have a counter that threads either increment or decrement, and which must stay above 0. If a thread wants to decrement the counter but it is at 0, it has to wait until the counter is incremented by another thread before decrementing it.

We protect a counter by a mutex both for reads and writes. If a thread grabs the mutex to decrement the counter and finds it is at 0, it has to wait for another thread to increment it. The derementing thread, however, is holding the mutex and thus preventing any modification to the counter. The decrementing thread must thus leave the mutex and wait until an increment happens.

The issue that arises then is how long to wait. A naive solution would be to use sleep calls like in the code below:

```
while ( true ) {
lock ( ) ;
if ( condition == true ) {
/* Leave with the lock taken */
break ; }

/* Else try again later */
unlock ( ) ;
sleep (TIMEOUT) ; }
```

This solution, however, has two main weaknesses. If the timeout is too large, the thread sleeps even when it could execute, potentially leading to massive performance penalties for the waiting thread. If the timeout is too small, the thread wakes up too often and wastes CPU time by needlessly trying to execute. The solution is to use condition variables, an API for notifying waiting threads to attempt to take the mutex and check if the condition they were testing holds. The waiting thread calls `pthread_cond_wait`, while the thread that notifies, uses `pthread_cond_signal` or `pthread_cond_broadcast` to wake up one or all waiting threads, respectively.
We use the POSIX Pthread API to synchronize between threads. Each thread in the code is a producer or a consumer of a shared resource. The threads enter and exit from the resource at regular intervals. At any point in time, the ratio of producers to consumers must be higher than a given value (e.g., for a ratio of two there can only be 4 consumers present in the resource if 2 or more producers are also present).

Use condition variables to ensure that the number of producers and consumers in the resource leads to a valid capacity ratio. Since the

arrival and departures of producers and consumers can invalid the capacity assertion, it is required that producers that want to exit must check whether they can safely exit without invalidating the capacity assertion. In the case that a producer's exit will invalidate the capacity assertation, the prodcuer thread needs to wait until more producers enters or consumers leave. The same holds for consumers wanting to enter the resource. Currently, the program does not work with a useful resource,nor does it perform any useful computation. it is only an exercise to work with mutexes and condition variables.
We have provided the `assignment.c` file, with stub code, you are required to implement the functions:

`consume_enter`: to ensure that consumer threads can only enter compute, if the capacity assertion does not fail

`consume_exit`: to ensure that we account for the consumer threads in the resource

`produce_enter`: to ensure that new producer threads are accounted for in the resource

`produce_exit`: to ensure that producer threads can only exit if the capacity assertion does not fail

To run the `a2q2` program:

```
% ./a.out
Usage: ./a.out <# consumers> <# producers> <ratio>
```

You must specify the number of consumers, number of producers and ratio to maintain for the resource capacity.
In a correct implementation, the program **always terminates**, a sample output from a correct implementation of the functions in `assignment.c` will have no output. For instance, consider the following executions of the a2q2 program:

```
 % ./a.out 8 3 2
Assertion failed: (num_producers * ratio >= num_consumers),
function resource_setup, file orderme.c, line 168.
zsh: abort
 %   ./a.out 8 3 2
 % ./a.out 16 3 2
Assertion failed: (num_producers * ratio >= num_consumers),
function resource_setup, file orderme.c, line 168.
zsh: abort
 % ./a.out 16 3 2
 % ./a.out 9 3 2
Assertion failed: (num_producers * ratio >= num_consumers),
function resource_setup, file orderme.c, line 168.
zsh: abort
```

```
 % ./a.out 9 3 2
 % ./a.out 8 4 2
 % ./a.out 10 5 2
 % ./a.out 20 10 2
 % ./a.out 21 10 2
Assertion failed: (num_producers * ratio >= num_consumers),
function resource_setup, file orderme.c, line 168.
zsh: abort
% ./a.out 21 10 2
 % ./a.out 20 11 2
 % ./a.out 22 11 2
```

You should not see any error messages as illustrated below:

```
% ./a2 8 4 2
Assertion failed: (resource->numAssertion failed:
(resource->numAssertion failed: (resource->num_consumers <=
resource->num_prod_consumers <=
resource->num_producers * resource->ratio),
functiucers * resource->ratio),
functiAssertion failed: (resource->numon assert_capacity,
```

Also note that you are required to run the program with a valid number of producers and consumers, given the ratio. **Important: to not modify any other files in the source code except `assignment.c`.**

5 Submitting Your Work

To submit your work, you must use the `cs350_submit` program in the `linux.student.cs` computing environment.
Important! You must use `cs350_submit`, not `submit`, to submit your work for CS350.

Note the usage for `cs350_submit` command is as follows

```
% usage: cs350_submit <assign_dir> <assign_num_type>
```

The `assign_dir` is the path to the root folder of the programming assignment. For the A2-kernel side programming assignment, the `assign_dir` is the path to your `os161-1.99` folder.
The `assign_num_type` for the kernel side is `ASST1`.

Note: There is no mistake here, we will use the ASST1 kernel to build and test this A2-OS/161 kernel side programming assignment. By using the `ASST1` kernel configuration, we will ensure the synchronization primitives are tested correctly. Also, note that if you used

the `#if OPT_A1` statements, they will be automatically included in the ASST1 kernel. Similarly, if you are using the cs350-container, you will use `ASST1` to build, and test the kernel for this progrmaming assignment, e.g. `build_kernel ASST1`.

For the `userspace` programming assignment, the `assign_dir` is the root directory for the `userspace` programming assignment. The `userspace` programming assignment root directory should maintain the directories that were in the starter code that was distributed for A2 userspace. The `assign_num_type` for the `userspace` is `ASSTUSER2`.
Therefore, to run the `cs350_submit` command for submitting the A2-`userspace` programming assignment the command will look like this:

```
% cs350_submit cs350-student/a2 ASSTUSER2
```

The argument `assign_dir` in the `cs350_submit` command, packages up your OS/161 kernel code or `userspace` program, respectively, and submits it to the course account using the regular `submit` command.
This assignment only briefly summarizes what `cs350_submit` does.
Look carefully at the output from `cs350_submit`. It is a good idea to run the `cs350_submit` command like this:

```
cs350_submit cs350-student/a2 ASSTUSER2 | tee submitlog.txt
```

This will run the `cs350_submit` command and also save a copy of all of the output into a file called `submitlog.txt`, which you can inspect if there are problems. This is handy when there is more than a screen full of output.
You may submit multiple times. Each submission completely replaces any previous submissions that you may have made for this assignment.