Introduction

In this assignment we will add synchronization tools\methods to the xv6 kernel. We start with a simple binary semaphore and then use it to implement more advanced synchronization means such as counting semaphores. Synchronization by itself would be meaningless if we didn't have several threads 'racing' on shared resources. Thus we must add support for threads to the xv6 implementation. Finally, we will implement a toy example of user level code that has several threads that use our synchronization means to achieve a common shared goal.

- Tip: xv6 was (and still is) developed as part of MIT's 6.828 Operating Systems Engineering course. You can find a lot of useful information and getting started tips there: http://pdos.csail.mit.edu/6.828/2010/overview.html

Task 0: Downloading xv6

Begin by downloading our revision of xv6, from the os112 svn repository:

- Open a shell, and traverse to the desired working directory.
- Execute the following command (in a single line):

  ```bash
  ```

This will create a new folder called assignment2 which will contain all project files.

Task 1: Threads

In this part of the assignment you will add system calls that allow users to create threads. One key characteristic of threads is that all threads of the same process share the same virtual memory space. As such, a simple way to add threads to xv6's implantation is to use code much similar to that of the "fork" system call, which creates another process. To achieve a thread-like behavior we will have the newly created "process" share the same virtual memory space of the "parent process". By doing so, both "processes" share the same virtual memory space, and can be considered as if they were two threads within the same process. You will need to mark the new "process" internally so the kernel can tell whether it is a process or one of our newly created "threads". Each thread should keep the PID of the process within which it is running.
Implement the following new system calls:

**int thread_create( void*(*start_func)(), void* stack, unit stack_size );**

Calling `thread_create` will create a new thread within the context of the calling process. The newly created thread state will be runnable. The caller to `thread_create` must allocate a stack for the new thread to use. `start_func` is a pointer to the function the thread will start executing.
When successful, the identifier of the newly created thread is returned. In case of error, a non-positive value is returned.

**int thread_getid();**

Upon success, this function returns the caller thread's id. In case of error, a non-positive error identifier is returned. For now we will settle on the simple solution where a thread's ID is identical to its (underline) process ID as assigned by the fork() call.

**int thread_getProcId();**

Upon success, this function returns the process ID. When a process (say PID=2321) first starts running, we say it has a single "main" thread, and the `thread_getProcId` simply returns the process' pid (2321). If that process later creates new threads, those threads would also return its pid (2321) since they are all threads of the same simulated process. Notice that the existing system call `Sys getpid` that is already implemented should not be changed, as it is used for other purposes which we would not want to harm.

**int thread_join(int thread_id, void** **ret_val)***

Suspends the execution of the calling thread until the thread identified by `thread_id` terminates.
`Thread_id` is the identifier of the thread that needs to be waited for.
Calling `thread_join` on a thread `thread_id` on which another thread is already waiting for termination returns -1.
On error returns -1 as well. On success, returns 0 and sets the `ret_val`.

**void thread_exit(void* retval)**

Terminates the execution of the calling thread. It doesn’t terminate the whole process if called from the thread's main function.
Task 2: Binary Semaphores

In this part of the assignment you will add system calls that allow users to create and use binary semaphores. For this task you should examine the implementation of spinlocks in XV6's kernel (for example the scheduler uses a spinlock). Spinlocks are used in XV6 in order to synchronize kernel cores while they change the status of processes. Your task is to implement binary semaphores as a kernel service to users, via system calls. The locking and releasing can be based on the implementation of spinlocks (remember spinlocks are made for the kernel internal specific use). Notice spinlocks are NOT the same as binary semaphores, for example starvation can happen with spinlocks but not with semaphores due to the queue of waiting threads that the semaphore maintains. The semantics of binary semaphores has been described in class and include the two operations below.

Add the following system calls to the XV6 kernel:

**int binary_sem_create()**
which returns an identifier (binary_sem_ID) of a newly created and initialized semaphore.

**int binary_sem_down(int binary_sem_ID)**
Returns 0 if successful, -1 if an error occurred.

**int binary_sem_up(int binary_sem_ID)**
Upon success, 0 is returned. In case of error, -1 is returned.

*Note:* you will need to assign and manage the IDs of semaphores inside the kernel.

Task 3: Counting Semaphores

Implement counting semaphores. Your implementation should rely on the system calls you added in the previous task. Implementing counting semaphores using binary semaphores was taught in class. Your implementation should be done outside of the kernel, as a user-space program called `semaphore.c` and `semaphore.h` (not as system calls).

Implement the following functions:

**struct semaphore* semaphore_create(int initial_semaphore_value);**

**void sem_up(struct semaphore* sem );**

**void sem_down(struct semaphore* sem );**

You may want to write some code now to fully test your semaphores and make sure they work flawlessly.
Task 4: Synchronized Bounded Buffer

Implement a synchronized bounded buffer using semaphores. Implement a user space program, **boundedbuffer.c** and **boundedbuffer.h**. The buffer has a finite capacity defined at creation and behaves as expected (trying to enter an item when the buffer is full causes the thread to wait until there are free slots in the buffer. Trying to remove elements from an empty buffer causes the thread to wait as well). Implement the following functions:

```c
struct BB* BB_create(int max_capacity);
void semaphore_put(struct BB* bb, void* element);
void* semaphore_pop(struct BB* bb);
```

You might want to consider adding some auxiliary function to support your bounded buffer.

Task 5: Thread test

Write a program called **threadTest** which receives a positive integer argument `n` from the command line. The program creates a single binary semaphore called the lock, and `n` threads and runs them. Each thread runs in an infinite loop, printing a line three times (see below). However, a process must acquire the lock just before printing, and must release it immediately after printing the line three times. After releasing the lock the thread performs sleep(1).

The line each thread prints is:

```
Process <thread_getProcId()> Thread <thread_getid()> is running.
```

Task 6: Producer-Consumer

Write a "consumer – producer" application that will create new threads and put them to test. Your application needs to get its parameters from a configuration file (See below). All printing should go to a file named 'ass2_log.txt'.

A. Description:
"Roses" is a new pub that opened in Beer-Sheva that serves one of a kind drink. It hasn't made all the arrangements and only bought 100 cups. Despite that, the busboy has a unique way to wash the used cups – when he receives 85% used cups he washes them instantly and return them for reuse. "Roses" employs only one busboy.
"Roses" employs **B** bartenders that handle requests (consumer). In addition, there is **H** hostesses that take requests from customers and deliver them to the bartenders (producer). Each entity is described below.
B. Data Structures:

There are two bounded buffers of different sizes you need to use.

1. The first bounded buffer is used to hold clean cups and is called CBB.

The buffer contains $C$ slots, which determines the total number of cups that "Roses" has.

This bounded buffer must support the following synchronized actions:

- **Cup* getCleanCup()** – Used by the bartender to fetch a clean cup if one exists. The bartender must get the lock on this buffer and only then is he able to get a clean cup. In case there are no clean cups to use, the bartender will wait until the busboy will wash the cups. Cup should be defined as a struct with members as you find right.

- **int washCups()** – When the busboy is notified by a waiting bartender that the number of used cups has reached $85\%$, he attempts to fill CBB one cup at a time, until $85\%$ of the capacity of CBB is full. The busboy attempts to hold the lock and when he succeeds, he washes one cup and releases the lock. This action is repeated until $85\%$ of the capacity of CBB is full.

If the busboy has filled $85\%$ of CBB and number of used cups is lower than $85\%$ then the busboy can go back to sleep. Otherwise, you should decide what to do. On success, 0 is returned. On Error, -1 is returned.

2. The second bounded buffer is used to hold customers' requests and is called RBB.

This bounded buffer has $R$ slots and is consumed by the bartenders.

This bounded buffer must support the following synchronized actions:

- **void addNewRequest( Request* newReq)** – This function is called by a hostess when a new customer arrives in the pub. The hostess creates a new Request (will be explained later) and places it at the end of the buffer.

- **Request* getRequest()** – This function is called by the bartender whenever he is able to serve customers' requests. The request located at the beginning of the buffer is handled and removed. Request should be defined as a struct with members as you find right.

C. The Bartender

There are $B$ (see configuration file in section 6) bartenders where each is a thread. The bartenders work loop can be abstracted to the following actions:

- Call getRequest().
- Call getCLeanCup().
- Print (to a file) "Bartender <thread_getid()> completed request #<request_id>
- Sleep(10).
The above work loop doesn’t contain all details and describes only the main actions taken by a bartender. You should pay attention to synchronization issues (e.g. interacting with the busboy).

D. The Hostess

There are $H$ (see configuration file in section 6) hostesses where each is a thread. The hostess work loop can be abstracted to the following actions:

- Create a new request.
- Call addNewRequest(...)
- Print (to a file) "Hostess <thread_getid()> added a new request #<request_id>
- Sleep(10)

Hostesses create requests until they reach the total number of request to create as defined in the configuration file. Each request has a unique identifier (int).
Again, you should pay attention to synchronization issues (e.g. interacting with the bartenders).

E. The Busboy

There is a single busboy which is a thread. The busboy sleeps until it is notified that at least 85% of the cups were used by the bartenders. When it is awake it does the following:

- Attempts to get the lock on CBB.
- Fills 85% of the CBB capacity with cups
- Print (to a file) "Busboy <thread_getid()> added <cups_added()> clean cups.

Pay attention to synchronization issues (e.g. interacting with the bartenders).

F. Configuration File

You need to add a configuration file to xv6 and name it 'con.conf' (make the necessary changes in the Makefile). Your application must be able to read properties from the configuration file and start a simulation according to it.
The configuration file has the following properties:

- $B = value1$
- $H = value2$
- $R = value3$
- $C = value4$
- $totalRequests = value5$

value1 – number of working bartenders that need to be created
value2 – number of working hostess that need to be created
value3 – size of the requests' bounded buffer (RBB).
value4 – size of the cups' bounded buffer (CBB).
value5 – total requests that will be created by the hostesses
Submission guidelines

Assignment due date: 17/04/11, 24:00
Make sure that your Makefile is properly updated and that your code compiles with no warnings whatsoever. We strongly recommend documenting your code changes with remarks – these are often handy when discussing your code with the graders. Due to our constrained resources, assignments are only allowed in pairs. Please note this important point and try to match up with a partner as soon as possible. Submissions are only allowed through the submission system. To avoid submitting a large number of xv6 builds you are required to submit a patch (i.e. a file which patches the original xv6 and applies all your changes). You may use the following instructions to guide you through the process:

*Back-up your work before proceeding!*

Before creating the patch review the change list and make sure it contains all the changes that you applied and noting more. Modified files are automatically detected by svn but new files should be added explicitly with the ‘svn add’ command:

> svn add <filename>

In case you need to revert to a previous version:

> svn revert <filename>

At this point you may examine the differences (the patch):

> svn diff

Alternatively, if you have a diff utility such as kompare:

> svn diff | kompare -o -

Once you are ready to create a patch simply make sure the output is redirected to the patch file:

> svn diff > ID1_ID2.patch

**Tip:** Although graders will only apply your latest patch file, the submission system supports multiple uploads. Use this feature often and make sure you upload patches of your current work even if you haven’t completed the assignment.

Finally, you should note that graders are instructed to examine your code on lab computers only(!) - **Test your code on lab computers prior to submission.**