CS 416 - Operating Systems Design

Homework 2
Semaphores and Kernel-Level Threads in xv6

October 20, 2016
Due by Friday, November 18th at 11:55 PM

1 Introduction

In this homework, you will add semaphores to the xv6 operating system. A semaphore is a variable or abstract data type that is used for controlling access, by multiple processes or threads, to a common resource in a parallel programming or a multi-user environment.

A useful way to think of a semaphore as used in the real-world systems is as a record of how many units of a particular resource are available, coupled with operations to adjust that record safely (i.e. to avoid race conditions) as units are required or become free, and, if necessary, wait until a unit of the resource becomes available. Semaphores are a useful tool[1] in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems[2]. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores and are used to implement locks.

The semaphore concept was invented by Dutch computer scientist Edsger Dijkstra in 1962 or 1963, and has found widespread use in a variety of operating systems[3].

2 Implementing Semaphores in xv6

2.1 Requirements

In this section, you will extend the xv6 kernel to add support for semaphores. To achieve this, you will construct a system-wide array of 32 semaphores, addressed by semaphore id (offset into the array). Each entry in the array will be of type semaphore, which has the following members:

```
struct semaphore
{
        int value;
        int active;
        struct spinlock lock;
}
```

In addition, you will implement the following set of system calls to allow user programs to setup and use these semaphores. The system calls are as follows:

[1] For a list of classical problems solved by semaphores, check ***The Little Book of Semahores***

[2] Simply using a semaphore or any other synchronization primitive will not fix race conditions. Incorrect use of semaphores might lead to previously nonexistent deadlock

[3] This introduction is from Wikipedia, for further information check original article

```
int sem_init(int semId, int n);
int sem_destroy(int semId);
int sem_wait(int semId);
int sem_signal(int semId);
```

Each of these calls should behave as follows:

- **sem_init(int semId, int n)** : Activate the semaphore whose id is semId. Initialize its value to n. Note that a semaphore should be activated with sem_init before it can be used. Furthermore, after calling sem_init for a semaphore, no other process can (re)initialize the semaphore until it's destroyed by a call to sem_destory.
- **sem_destroy(int semId)** : Mark the semaphore whose id is semId as destroyed (non-active). After this call, other processes can reinitialize the semaphore again.
- **sem_wait(int semId)** : this operation decrements the semaphore's value by one. A semaphore's value cannot drop below zero. Calling wait on a semaphore with the value zero should block (put it to sleep) the calling process until another process increments (calls sem_signal) the semaphore.
- **sem_signal(int semId)** : this operation increments the semaphore's value by one. If there any processes that are waiting on this semaphore, unblock one (wakeup the waiting process).

Process blocking should be implementing the xv6 sleep(...)/wakeup(...) functions.

2.2 Testing

The starting branch you checkout for this homework contains a test program named **sem_test** that will test your semaphore implementation. Your final implementation should be able to pass the tests performed by this program.

By default, the program **sem_test** is commented out in xv6's makefile. It's so because prior to adding the 4 system calls mentioned earlier, building sem_test.c on xv6 will fail. Remove this comment once you add these system calls to xv6.

3 Implementing Kernel Level Threads in xv6

By default, xv6 does not implement support for threading. For this part, you will add kernel-level thread functionality to xv6.

3.1 Requirements

You must add kernel-level thread functionality to xv6 by implementing some new system calls, as well as making several other supporting modifications.

3.1.1 Kernel

Implement three new system calls to handle threading:

```
int clone(void *(*func) (void *), void *arg, void *stack);
int join(int pid, void **stack, void **retval);
void texit(void *retval);
```

The implementations of these system calls should be as follows:

- **int clone (void *(*func) (void *), void *arg, void *stack)** - This system call should behave as a lightweight version of **fork(...)**. In particular, the system call should create a new OS process (**struct proc**) that has its own kernel stack and other structures, but shares the memory address space of the parent. When a user calls **clone(...)**, they should provide a function for the thread to run, a single pointer to an argument, as well as a one-page region of memory to be used as a user stack. The **clone(...)** call should return the PID of the new thread to the parent, and immediately start execution of the function **func** in the new thread's context.
- **int join(int pid, void **stack, void **retval)** - This system call should behave similar to **wait()**, and cause the caller to sleep until the thread specified by the *pid* argument terminates. **join(...)** should return 0 on success (negative number if an error), as well as copy the address of the thread's user stack and the return value (passed to **texit(...)**) into the pointers provided as parameters.
- **void texit(void *retval)** - This system call should behave similar to **exit()**, but takes a pointer that should be passed to the caller of **join(...)**.

Additionally, you should consider the behavior of some some other system calls:

- **exit()** - If called by a process with active child threads, those threads should be killed and cleaned up before the parent exits.
- **kill()** - If called on a process with active child threads, those threads should be killed and cleaned up before the parent is killed.

3.2 Testing

The starting branch will contain a test program called **thread_test** that will test your new threading facility. Your implementation must pass the **sem_test** before it can pass the thread test.

By default, the program **thread_test** is commented out in xv6's makefile. It's so because prior to adding all the system calls mentioned earlier, building **thread_test** on xv6 will fail. Remove this comment once you add all these system calls to xv6.

4 Source Control

To start on this project, use the following commands in your xv6 git repository [4]

```
git checkout fa16-hw2-start
```

After checking out, use the following commands to create your working branch:

```
git branch fa16-hw2
git checkout fa16-hw2
```

While you don't need to use git features to complete this homework, we believe using git can help with writing the homework[5]. We suggest that you **git commit** often while you are working. We recommend that you commit each time you have made a complete, significant change, and use a descriptive commit message. Typically in systems development, one feature may be implemented over dozens of individual commits

5 Overall Requirements

- The code has to be written in **C** language. Ask your TA if you feel inline **asm** is necessary to write the homework.
- Do not copy the solution from other students. Use the course forum to discuss ideas on how to implement the homework. Do not post your solutions there. It is your task to debug your program. Do not post questions that involve fixing or debugging partial solutions.
- We suggest committing in your local git repository during your changes to xv6. This simplifies isolating problems in your program. And if a change breaks your program, you'll have the option to revert back to an older commit.
- Submit a report on **Sakai** named **report.pdf**, detailing what you accomplished for this project, including issues you encountered. This report *must* be named **report.pdf** and *must* be in PDF format.
- To submit your code on **Sakai**, go to your xv6 code. Run **make clean** to remove all the unnecessary files for building xv6. Then create a tarfile of the xv6 folder which contains all the remaining xv6 files.
- By convention, all the systems calls that you write for this homework that return an integer, should return 0 on success and a negative value on failure. For certain system calls such as **clon(...)**, more specific return values are specified in the requirements section

[4] You need to clone the repository from https://github.com/nbushehri/xv6.git

[5] There are many tutorials on git online. For a detailed reference, you can use the git book at https://git-scm.com/book/en/v2