CS 416 - Operating Systems Design

Homework 2
Implementing Semaphores in xv6

March 24, 2015
Due April 9th, 2015 at 11:59 PM

1 Introduction

In this homework, you will implement a semaphore facility in xv6. 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 is as a record of how many units of a particular resource are available, coupled with operations to safely (i.e., without race conditions) adjust that record as units are required or become free, and, if necessary, wait until a unit of the resource becomes available. Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. 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**.

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. [1]

2 Implementing Semaphores on xv6

2.1 Requirements

You need to extend the xv6 kernel to add support for semaphores. To do this, you will construct a **system-wide** table of 32 semaphores, addressed by semaphore ID (offset into the table). Each entry in the table can look something like this:

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

[1] This is from Wikipedia, see the article there for more information

Additionally, you will need to implement a set of system calls to allow user programs to manage and use these semaphores. These calls should be as follows:

```
int sem_init(int sem, int value);
int sem_destroy(int sem);
int sem_wait(int sem, int count);
int sem_signal(int sem, int count);
```

The implementations of these system calls should be roughly as follows:

- **sem_init(...)** - Initialize the value and mark the semaphore as active. No other process can (re)initialize the semaphore until the after it is destroyed.
- **sem_destroy(...)** - Mark the semaphore as destroyed (non-active). Other processes can reinitalize the semaphore at this point.
- **sem_wait(...)** - Try to obtain *count* instances of the semaphore. Equivalent to the classical **P()** operation. If *count* instances are not available, block the calling process until enough instances become available.
- **sem_signal(...)** - Release *count* instances of the semaphore. Equivalent to the classical **V()** operation. If any processes are waiting for instances of this semaphore, unblock one (if enough instances are available).

You should implement process blocking using the xv6 **sleep(...)/wakeup(...)** functions.

2.2 Testing

The starting branch mentioned above will contain a test program called **sem_test** that will test your new signal facility. Your implementation must be able to pass the tests performed by this code.

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.[2]

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.

Implement two new system calls to handle threading:

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

The implementations of these system calls should be as follows:

- **clone(...)** - 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.
- **join(...)** - This system call should behave similar to wait(), and cause the caller to sleep until one of the threads in the process terminates. **join(...)** should return the PID of whichever thread was joined, as well as copy the address of the thread's user stack into the pointer provided as a parameter.

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 mentioned above 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**.

*The **thread_test** program is commented out of the Makefile by default. If you do this section, you **must** uncomment this line in UPROGS.*

[2] Required for CS518, extra credit for CS416.

4 Source Control

To start on this project, use the following commands in your xv6 git repository (assuming you have checked it out as instructed in the document uploaded to Sakai):

```
git fetch origin
git checkout sp15-hw2-start
```

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

```
git branch sp15-hw2
git checkout sp15-hw2
```

Please ensure 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 Submission

To submit your work, please follow the following instructions:
Type the following (for each netids field, use both partner's netids, hyphenated like **NETID1-NETID2**):

```
git checkout sp15-hw2
make submit netids=<NETIDS> name=hw2 base=sp15-hw2-start
```

After this is complete, you should have a .tar.gz file containing individual patch file(s) for each commit. Ensure that the tar file actually contains the patches, then submit this tarball on Sakai.

Only ONE student from each group should commit, but you NEED to make sure you give the netids of both partners.

6 Overall Requirements

- The code has to be written in **C** language. You should discuss on Piazza if you think inline **asm** is necessary to accomplish something.
- Do not copy the solution from other students. Use Piazza to discuss ideas of how to implement it. Do not post your solution there. Use Sakai to submit it.
- Commit in your local git for each step of your solution to make sure you can isolate each step later in case you need to refer back.
- 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.