Prof. Ramadan, CS 240 , KAUST

Assignment #4 : " We have shared memory … now all we need are Semaphores."

This assignment is individual (no teams). This assignment is optional. It is a bonus assignment that replaces your lowest quiz score (assuming you get a higher score on the assignment).

It is due in one week, before our next class (November 28)

You can start with original xv6 sources, but are encouraged build on your previous assignment (so you can use both shared memory and semaphores in your test program). In all cases, try to isolate the semaphore code into its own implementation files so that you can easily add it to the original xv6 sources without a lot of work. (Your description of what files are added/modified, etc. should describe the semaphore changes only, without any shared memory changes).

See the general assignment instructions for submission instructions. Report should be the usual 1-2 pages, describing the key details and decisions of your implementation (esp. on how you use wait-channels, locks inside the kernel, etc.).

Your task is to implement semaphores. You're encouraged to review the textbook material on semaphores.

You are allowed to use the wait channel abstraction (sleep/wakeup) that is provided by xv6 scheduling code, to make your life easier.

The system calls you will need to add are as follows:

```
 int sem_get (uint name, int value);
 int sem_delete (int handle);
 int sem_signal (int handle);
 int sem_wait (int handle);
```

You are expected to test your implementation with a user program that uses "full" semaphores, "empty" semaphores, and "mutex-style" semaphores (count of 1).

The sem_get operation is used to create/get a semaphore, based on a name (we use numbers for simplicity), and the maximum semaphore value is specified. sem_get returns a handle (also an int) to the semaphore. The other functions take this handle, and either delete, signal, or wait on the semaphore.

Note that since we want to use semaphores with shared memory, two processes that call sem_get on the same name, should be using the same semaphore (i.e. semaphores should not be local to a single process). This also means you need to figure out what you will do with semaphores when you fork.

The functions in general should return 0 if everything is OK (except sem_get, which should return a positive handle). Negative numbers can be used for errors, such as trying to signal/wait/delete a non-existent semaphore.   If your implementation has a maximum limit on the number of semaphores in the system , make sure it is no less than 10, then define it in a constant, such as NSEM, and put it in some place param.h). You can also return a negative number to indicate if this limit has been exhausted (e.g. trying to create a new semaphore above this limit).

So basically, we can consider the following return values as part of our system call interface.

```
#define SEM_OK 0
#define SEM_DOES_NOT_EXIST -1
```

```
#define OUT_OF_SEM -2
```
EXAMPLE CODE

Below is a sample test program provided by a previous student in the class, that tests his implemenetation of semaphores. It does not use shared memory. You are free to use it (at your own risk) to test your code, or as an example of understanding the semaphore semantics. (If you write a test program that you feel is worthy of inclusion for future students, include the source in your submission (and make sure it builds and runs in qemu), and describe it briefly in your report).

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main(int argc, char *argv[]) {

  int semFull;
  int semEmpty;
  int semMutex;
  char buffer[10];
  int i, j;

  for(i = 0; i < 10; i++) {
    buffer[i] = 'E';
  }

  int pid;

  semEmpty = sem_get(100, 10);
  semFull  = sem_get(200, 0);
  semMutex = sem_get(300, 1);

  printf(1, "producer PID = %d\n", getpid());

  pid = fork();

  // Producer
  if(pid != 0) {
    //while(1) {
    for(j = 0; j < 100; j++) {
      printf(1, "\nProducer\n");
      sem_wait(semEmpty);
      sem_wait(semMutex);
      for(i = 0; i < 10; i++) {
        if (buffer[i] == 'E') {
          buffer[i] = 'F';
          break;
        }
      }
      for(i = 0; i < 10; i++) {
        printf(1, "%c ", buffer[i]);
      }
      printf(1, "\n");
      sem_signal(semMutex);
      sem_signal(semFull);
    }
  // Consumer
  } else {
    //while(1) {
    for(j = 0; j < 100; j++) {
      printf(1, "\n        Consumer\n");
      sem_wait(semFull);
      sem_wait(semMutex);
      for(i = 0; i < 10; i++) {
        if (buffer[i] == 'F') {
          buffer[i] = 'E';
          break;
        }
      }
      for(i = 0; i < 10; i++) {
        printf(1, "%c ", buffer[i]);
      }
      printf(1, "\n");
      sem_signal(semMutex);
      sem_signal(semEmpty);
    }
  }
  exit();
}
```