CS 4375 Fall 2025
Homework 5: Semaphores for xv6
125 points

MD Armanuzzaman

Dude Date: December 8, 2025 (11.59 PM))

For this lab, we will implement unnamed semaphores in xv6 that can be used by processes to synchronize access to shared memory. Doing so will allow us to program a general producer-consumer application with multiple producers and consumers operating on a shared buffer. You should implement the POSIX standard `sem_init()`, `sem_destroy()`, `sem_wait()`, and `sem_post()` semantics for unnamed semaphores shared between processes.

Pull the **hw5-init** branch from the instructor's xv6 repository. Add instructors repository as upstream (may not be necessary if the remote is already set):

```
$git remote add upstream https://github.com/Tomal-kuet/xv6-
   riscv-labs.git
```

Verify upsteam:

```
$git remote -v
origin  git@github.com:yourUSER/yourPRIVATErepo.git (fetch)
origin git@github.com:yourUSER/yourPRIVATErepo.git (push)
upstream https://github.com/Tomal-kuet/xv6-riscv-labs.git (
   fetch)
upstream https://github.com/Tomal-kuet/xv6-riscv-labs.git (push
   )
```

And then create a hw5 branch with:

```
$git fetch upstream hw5-init
$git checkout -b hw5 upstream/hw5-init
```

By doing this, you will have `mmap()` and `munmap` system calls and a `private` command to test the given implementation. You will also get lazy allocation implemented in this repository which are necessary prerequisites of this homework.

Task 1. System call declarations and preliminaries (20 pts)

Add `sem_init()`, `sem_destroy()`, `sem_wait()`, and `sem_post()` system call declarations with the same prototypes as the POSIX calls for unnamed semaphores to `user/user.h`. We will define a `sem_t` type so go ahead and use that type. Make the necessary modifications to `user/usys.pl`, `kernel/syscall.h`, and `kernel/syscall.c` for the new system calls. Add the following line to kernel/types.h:

```
typedef int sem_t;
```

Put the provided test program `prodcons-sem.c` in the user directory and add this program to UPROGS in Makefile. Type 'make qemu' and fix any compilation errors. You can try running `prodcons-sem` but it won't work yet since you haven't implemented the system calls.

Task 2. Data structures, semaphore allocation and deallocation (20 pts)

Add data structure definitions for semaphores to `spinlock.h` as follows:

```
// Counting semaphore
struct semaphore {
        struct spinlock lock; // semaphore lock
        int count; // semaphore value
        int valid; // 1 if this entry is in use
};
// OS semaphore table type
struct semtab {
        struct spinlock lock;
        struct semaphore sem[NSEM];
};
extern struct semtab semtable;
```

Add the following line to kernel/param.h:

```
#define NSEM 100 // maximum open semaphores per system
```

In a new file `semaphore.c`, create a data structure to keep track of the semaphores, which will be kernel-held counters. For simplicity, you may use a fixed-length table. In that case, define a value for `NSEM` in `kernel/param.h` and place the following in `semaphore.c`:

```
#include "types.h"
#include "riscv.h"
#include "param.h"
#include "defs.h"
#include "spinlock.h"

struct semtab semtable;
```

```
void seminit(void)
{
        initlock(&semtable.lock, "semtable");
        for (int i = 0; i < NSEM; i++)
        {
                initlock(&semtable.sem[i].lock, "sem");
        }

}
```

Add a function `semalloc()` to `semaphore.c` that returns the index of an unused location in the semaphore table, or returns -1 if there is no empty location. Add a function `semdealloc()` to `semaphore.c` that takes a semaphore index as an argument and invalidates that entry in the semaphore table. Remember to use concurrency control since more than one kernel thread of execution may be accessing the semaphore table. Add declarations for `seminit()`, `semalloc()` and `semdealloc()` to `kernel/defs.h`. You will need to add semaphore.o to OBJS in the Makefile in order for `semaphore.c` to get compiled. Call `seminit()` along with the other initialization routines in `main()` in `kernel/main.c`. Check that the code compiles and fix any compile errors.

Task 3. Implementing semaphore system calls (40 pts)

Add your code for `sys_sem_init()`, `sys_sem_destroy()`, `sys_sem_wait()`, and `sys_sem_post()` to `kernel/sysproc.c`. See sections 7.5 and 7.6 of the xv6 textbook for a hint on how to implement semaphores in xv6. Note that you will need to use `copyout()` in `sys_sem_init()` and `copyin()` in `sys_sem_wait()`, `sys_sem_post()`, and `sys_sem_destroy()` to access the user's `sem_t value`. Make all the necessary changes to the various user and kernel files to finish implementing the new system calls.
Sample output of the implementation:

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ prodcons-sem 1 1
producer 5 producing 1
producer 5 producing 2
producer 5 producing 3
consumer 4 consuming 1
producer 5 producing 4
consumer 4 consuming 2
consumer 4 consuming 3
producer 5 producing 5
consumer 4 consuming 4
```

```
producer 5 producing 6
consumer 4 consuming 5
producer 5 producing 7
consumer 4 consuming 6
producer 5 producing 8
consumer 4 consuming 7
producer 5 producing 9
consumer 4 consuming 8
consumer 4 consuming 9
producer 5 producing 10
consumer 4 consuming 10
producer 5 producing 11
consumer 4 consuming 11
producer 5 producing 12
consumer 4 consuming 12
producer 5 producing 13
consumer 4 consuming 13
producer 5 producing 14
consumer 4 consuming 14
producer 5 producing 15
producer 5 producing 16
consumer 4 consuming 15
producer 5 producing 17
consumer 4 consuming 16
producer 5 producing 18
producer 5 producing 19
consumer 4 consuming 17
producer 5 producing 20
consumer 4 consuming 18
consumer 4 consuming 19
consumer 4 consuming 20
total = 210
$ prodcons-sem 2 3
producer 10 producing 1
consumer 7 consuming 1
producer 10 producing 2
consumer 7 consuming 2
producer 10 producing 3
consumer 7 consuming 3
producer 10 producing 4
producer 10 producing 5
consumer 7 consuming 4
consumer 9 consuming 5
producer 10 producing 6
consumer 9 consuming 6
```

```
producer 10 producing 7
producer 10 producing 8
consumer 7 consuming 7
consumer 9 consuming 8
producer 11 producing 9
consumer 7 consuming 9
producer 11 producing 10
consumer 7 consuming 10
producer 10 producing 11
consumer 7 consuming 11
producer 10 producing 12
consumer 7 consuming 12
producer 10 producing 13
consumer 9 consuming 13
producer 10 producing 14
consumer 9 consuming 14
producer 11 producing 15
consumer 8 consuming 15
producer 10 producing 16
consumer 7 consuming 16
producer 10 producing 17
consumer 7 consuming 17
producer 11 producing 18
consumer 7 consuming 18
producer 10 producing 19
consumer 9 consuming 19
producer 11 producing 20
consumer 7 consuming 20
total = 210
$ prodcons-sem 5 2
producer 15 producing 1
consumer 13 consuming 1
producer 15 producing 2
consumer 13 consuming 2
producer 15 producing 3
consumer 14 consuming 3
producer 15 producing 4
consumer 13 consuming 4
producer 19 producing 5
consumer 13 consuming 5
producer 15 producing 6
consumer 13 consuming 6
producer 15 producing 7
producer 15 producing 8
consumer 13 consuming 7
```

```
producer 19 producing 9
consumer 13 consuming 8
consumer 14 consuming 9
producer 15 producing 10
producer 15 producing 11
producer 19 producing 12
consumer 13 consuming 10
consumer 14 consuming 11
producer 19 producing 13
consumer 14 consuming 12
consumer 13 consuming 13
producer 15 producing 14
consumer 13 consuming 14
producer 15 producing 15
consumer 14 consuming 15
producer 15 producing 16
consumer 13 consuming 16
producer 15 producing 17
consumer 14 consuming 17
producer 15 producing 18
consumer 14 consuming 18
producer 15 producing 19
consumer 14 consuming 19
producer 15 producing 20
consumer 14 consuming 20
total = 210
```

Task 4. Readers–Writers with xv6 Semaphores (40 pts)

You have implemented unnamed semaphores (`sem_init`, `sem_wait`, `sem_post`, `sem_destroy`) and used them in `prodcons-sem.c` with mmap. In this exercise, you are given an xv6 user program skeleton that sets up shared memory and semaphores for the classic **readers–writers** problem. The shared state and main function are already implemented for you:

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

#define NULL 0
#define READER_ITERS 50
#define WRITER_ITERS 100

typedef struct {
        int value;        // shared "database" value
        int readercount;  // number of active readers
        sem_t mutex;      // protects readercount
        sem_t wrt;        // writers' lock (and readers' first/
           last lock)
} rw_t;

rw_t *rw;

void
reader(void)
{
        /* implement your semaphore logic for reader */
}

void
writer(void)
{
        /* implement your semaphore logic for writer */
}

int
main(int argc, char *argv[])
{
        if (argc != 3) {
                printf("usage: %s <nreaders> <nwriters>\n",
                   argv[0]);
                exit(0);
        }

        int nreaders = atoi(argv[1]);
        int nwriters = atoi(argv[2]);
        int i;

        rw = (rw_t *) mmap(NULL, sizeof(rw_t),
        PROT_READ | PROT_WRITE,
        MAP_ANONYMOUS | MAP_SHARED, -1, 0);
        if (rw == (void *)-1 || rw == NULL) {
                printf("rw-sem: mmap failed\n");
                exit(1);
        }

        // initialize shared state
        rw->value = 0;
        rw->readercount = 0;
        sem_init(&rw->mutex, 1, 1);
        sem_init(&rw->wrt,   1, 1);

        /* fork readers */
        for (i = 0; i < nreaders; i++) {
                if (!fork()) {
                        reader();
                }
        }

        // fork writers
        for (i = 0; i < nwriters; i++) {
                if (!fork()) {
                        writer();
                }
        }

        // wait for all children
        for (i = 0; i < nreaders + nwriters; i++)
        wait(0);

        // check final value
        int final = rw->value;
        int expected = nwriters * WRITER_ITERS;
        printf("rw-sem: final value = %d, expected = %d\n",
           final, expected);

        // cleanup
        sem_destroy(&rw->mutex);
        sem_destroy(&rw->wrt);
        munmap(rw, sizeof(rw_t));

        exit(0);
}
```

Your task is to implement the bodies of `reader()` and `writer()` using the unnamed semaphores mutex and `wrt` stored in the shared `rw` object.

Implement **reader()** using the **readers–writers** algorithm from lecture:

- Multiple readers should be allowed to read `rw->value` concurrently.
- Readers must use `rw->mutex` to safely update `rw->readercount`.
- The first reader should acquire `rw->wrt` (blocking writers), and the last reader should release `rw->wrt`.

- Each reader should perform **READER_ITERS** iterations of: enter, read `rw->value`, exit, then eventually call `exit(0)`.

After completing the implementation make necessary changes to the `Makefile` to add it as a user program before you run `make qemu`.

Implement `writer()` so that writers have exclusive access to the shared value:

- Each writer should perform **WRITER_ITERS** iterations.
- For each iteration, it should acquire `rw->wrt`, increment rw-¿value by 1, then release `rw->wrt`.
- Writers must not execute their critical section concurrently with any other writer or any reader.

When you run the program as:

```
rwtest-sem <nreaders> <nwriters>
```

the final output:

```
rwtest-sem: final value = X, expected = Y
```

Sample test cases are below, you need include the out of you implementation in your report along with your `rwtest-sem.c` file.

```
$rwtest-sem 0 1
$rwtest-sem 3 0
$rwtest-sem 1 1
$rwtest-sem 3 2
```

Turn-in procedure and Grading

Please use the accompanying template for your report. Convert your report to PDF format and push your hw5 branch with your code and report to your xv6 GitHub repository by the due date. Also, turn in the assignment on Teams with the URL of your GitHub repo by the due date. Give access to your repo to the TA.

This assignment is worth 125 points. The breakdown of points is given in the task descriptions above. The points for each task will be evaluated based on the correctness of your code, proper coding style and adequate comments, and the section of your report for that task.

You may discuss the assignment with other students, but do not share your code. Your code and lab report must be your own original work. Any resources you use should be credited in your report. If we suspect that code has been copied from an online website or GitHub repo, from a book, or from another student, or generated using AI, we will turn the matter over to the Office of Student Conduct for adjudication.