Brian Duenas

CSE 460

Lab 4

20 points Total

1. Shared Memory

Q: Write a brief description on the usage of each semaphore.

A:

Shmget() : returns the identifier of the System V shared memory segment associated with the value of the argument key. (shared memory get)

Shmat() : attaches the System V shared memory segment identified by *shmid* to the address space of the calling process. (shared memory attach)

Shmdt() : detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process. (shared memory detach)

Shmctl() : performs the control operation specified by cmd on the System V shared memory segment whose identifier is given in *shmid*. (shared memory control)

Q: Type in some text at the terminals. What do you see? What text you enter will terminate the programs? Explain what you have seen.

Output:

```
[user@csusb.edu@jb359-1 lab7]$ shared1
Memory attached at ADFF6000
You wrote : testing
You wrote : end
[user@csusb.edu@jb359-1 lab7]$
```

```
[user@csusb.edu@jb359-1 lab7]$ shared2
Memory attached at 33C8C000
Enter some text : testing
waiting for client...
Enter some text : end
[user@csusb.edu@jb359-1 lab7]$
```

A: When typing into the terminal with the program *shared2* running the text appears in the terminal window with *shared1* running. Typing “end” will terminate both programs. Both programs are accessing the same memory space so when *shared2* writes (produces) program *shared1* reads the change (consumes).

Code for *shared1.cpp*:

```
        ...
        shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

        sem_t *sem = sem_open(NAME, O_CREAT, 0644, 3);

        if (shmid == -1) {
                fprintf(stderr, "shmget failed\n");
                exit(EXIT_FAILURE);
        }

        /* We now make the shared memory accessible to the program. */

        shared_memory = shmat(shmid, (void *)0, 0);
        if (shared_memory == (void *)-1) {
                fprintf(stderr, "shmat failed\n");
                exit(EXIT_FAILURE);
        }

        printf("Memory attached at %X\n", shared_memory);

        shared_stuff = (struct shared_use_st *)shared_memory;
        shared_stuff->written_by_you = 0;

        sem_init(sem, 1, 1);

        while (running) {
                sem_wait(sem);
                if (shared_stuff->written_by_you) {
                        printf("You wrote: %s", shared_stuff->some_text);
                        sleep(rand() % 4); /* make the other process wait for us ! */
                        shared_stuff->written_by_you = 0;
                        if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
                                running = 0;
                        }
                }
                sem_post(sem);
        }

        if (shmdt(shared_memory) == -1) {
                fprintf(stderr, "shmdt failed\n");
                exit(EXIT_FAILURE);
        }

        if (shmctl(shmid, IPC_RMID, 0) == -1) {
                fprintf(stderr, "shmctl(IPC_RMID) failed\n");
                exit(EXIT_FAILURE);
        }

        sem_close(sem);
        sem_unlink(NAME);
        exit(EXIT_SUCCESS);
}
```

Code for *Shared2.cpp*:

```
	...
	shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

	sem_t *sem = sem_open(NAME, 0);

	if (shmid == -1) {
		fprintf(stderr, "shmget failed\n");
		exit(EXIT_FAILURE);
	}

	shared_memory = shmat(shmid, (void *)0, 0);
	if (shared_memory == (void *)-1) {
		fprintf(stderr, "shmat failed\n");
		exit(EXIT_FAILURE);
	}

	printf("Memory attached at %X\n", shared_memory);

	shared_stuff = (struct shared_use_st *)shared_memory;

	//initialize semaphore
	sem_init(sem, 1, 1);

	while (running) {
		sem_wait(sem);
		while (shared_stuff->written_by_you == 1) {
			sleep(1);
			printf("waiting for client...\n");
		}
		printf("Enter some text: ");
		fgets(buffer, BUFSIZ, stdin);

		strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
		shared_stuff->written_by_you = 1;

		if (strncmp(buffer, "end", 3) == 0) {
			running = 0;
		}
		sem_post(sem);
	}

	if (shmdt(shared_memory) == -1) {
		fprintf(stderr, "shmdt failed\n");
		exit(EXIT_FAILURE);
	}
	sem_close(sem);
	sem_unlink(NAME);
	exit(EXIT_SUCCESS);
}
```

2. POSIX Semaphores

Q: Try the server-client example and explain what you observe. You have to start the server first why?

Outputs:

```
[user@csusb.edu@jb359-1 lab7]$ server
[user@csusb.edu@jb359-1 lab7]$

[user@csusb.edu@jb359-1 lab7]$ client
ABCDEFGHIJKLMNOPQRSTUVWXYZ[user@csusb.edu@jb359-1 lab7]$
```

A: The server starts and writes to memory. The client reads from memory and signals to server that it has finished reading. Server must starts first because it creates the memory space with the flag *O_CREAT*, while the client opens a created memory space.

Q: Modify the programs so that the server sits in a loop to accept string inputs from users and send them to the client, which then prints out the string.

Code for *server.cpp*:

```
//function to make string to char stream
char* changer(string s) {
	char* writable = new char[s.size() + 1];
	copy(s.begin(), s.end(), writable);
	writable[s.size()] = '\0';
	return writable;
}

int main()
{
	...
	char* cc;

	while (*shm != '*')
	{
		s = shm;
		sem_wait(mutex);
		printf("Enter some text: ");
		getline(cin, in);
		cc = changer(in);
		for (int i = 0; cc[i] != '\0'; i++)
		{
			*shm++ = cc[i];
		}
		*shm = '\0';

		sem_post(mutex);
	}

	//the below loop could be replaced by binary semaphore
	sem_close(mutex);
	sem_unlink(SEM_NAME);
	shmctl(shmid, IPC_RMID, 0);

	_exit(0);
}
```

Code for *Client.cpp*:

```
int main()
{
	...
	//start reading
	while (*shm != '*')
	{
		s = shm;
		sem_wait(mutex);
		if (*shm == '/')
		{
			for (int i = 0; shm[i] != '\0'; i++)
			{
				std::cout << *shm++;
			}
			std::cout << '\n';
		}
		sem_post(mutex);
	}
	//once done signal exiting of reader:This can be replaced by another semaphore
	sem_close(mutex);
	shmctl(shmid, IPC_RMID, 0);
	exit(0);
}
```

3. XV6 System Calls

Q: Modify the files and show the output for *ps*.

Output:

```
$ ps
name     pid     state
init     1       SLEEPING
 sh      2       SLEEPING
 ps      4       RUNNING
 $
```

Q: Modify ***cps()*** in *proc.c* so that it returns the total number of processes that are SLEEPING or RUNNING.

Code:

```
Int sleepCount = 0;
Int runCount = 0;
...
int
cps()
{
struct proc *p;


// Enable interrupts on this processor.
sti();

// Loop over process table looking for process with pid.
acquire(&ptable.lock);
cprintf("name \t pid \t state \n");
for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
       if (p->state == SLEEPING)
       {
              cprintf("%s \t %d  \t SLEEPING \n ", p->name, p->pid);
              sleepCount++;
       }
       else if (p->state == RUNNING)
       {
              cprintf("%s \t %d  \t RUNNING \n ", p->name, p->pid);
              runCount++;
       }
}
cprintf("Total RUNNING: %d", runCount);
cprintf("\nTotal SLEEPING: %d\n", sleepCount);
runCount = 0;
sleepCount = 0;
```

Output:

```
$ ps
 name     pid     state
 init     1       SLEEPING
 sh      2       SLEEPING
 ps      4       RUNNING
 Total RUNNING : 1
 Total SLEEPING : 2

$
```