HW4 Linux Semaphore

Files provided：proj3.zip (*sem.c prog1.c prog2.c p1.c p2.c p3.c awk_sem.h)*
Files to upload for grading：
1.All of needed files, include source code.
2.README：Explain how to compile and execute your program.
Compression the files and named "Student ID-HW4.zip" then upload to LMS.
Environment： Linux only

Introduction

The goal of this lab is make you familiar with semaphores and use them to coordinate and synchronize among processes. The concept introduced in this lab is also applied to multithreads. Every operating systems provides a set of synchronization commands (system calls), called IPC (Inter Process Communication) Although the functionality and signatures are different in different O.S., they are basically similar. Please note that, this lab can only be done in Linux. TA will provide you the Linux environment for the labs.

In the provided files, there are two sample programs, *prog1.c* and *prog2.c.* Please compile them and make them executable in the Linux. *prog1.c* is very simple. It starts by creating a semaphore. The name of semaphore is composed by two parts. One is path and another is project id. In *prog1.c,* it uses create_sem to create a semaphore with name "."+"S" and its initial value is 0.

IMPORTANT： In principle, a semaphore is created for any other processes (including other users) to access it. Any processes can access the semaphore once they know the path+name. So, if you plan to run this prog1 and prog2 in the same operating systems with other, you better clear it before doing so. Another better way is to rename the semaphore so that no one can access the same semaphore with you.

After the creation of a semaphore, *prog1* calls *P(semid)*, which in some OS books called *wait(semid)*. However, the original Linux system calls of semaphore are not *P(s)* nor *wait(s).* Interested readers can open *sem.c* to see how the real Linux system calls are wrapped into *P(s)* and *V(s).*

Because *semid* is initialized as 0, so according to the semantics of a semaphore, *prog1* should block on the *P(semid).* After *prog1* enters a blocking state, you can run *prog2*. The code of *prog2* is very simple too. It uses the name "."+"S" to find the semaphore created by *prog1* and then call *V(semid)* to release a blocked process from semaphore *semid*. In this case, it is *prog1* to be woken up. This is the *signal(S)* in OS

text books. This call will wake up *prog1* and then both *prog1* and *prog2* run concurrently to the end.

Sample Runs：

To run *prog1* and *prog2*, please compile the program as follows:

```
> gcc –o  prog1  prog1.c sem.c
> gcc –o  prog2  prog2.c sem.c
```

Please run prog1 in Linux as follows:

```
> prog1 &
```

The purpose of "&" is to tell your shell to run your program in background without blocking your shell. So, although prog1 is immediately blocked by semaphore, your shell is still alive for you to continue entering commands.

Next, please run

```
> prog2 &
```

Prog2 should wake up prog1 and then both run to the end concurrently.

After *prog1* and *prog2* finished, remember that semaphore still exist. You can use *ipcs* command to list the IPC resources you own. If you want to delete the semaphore manually, you can use `icprm sem semid`, where *semid* is the semaphore id listed by `ipcs`.

You can try a series of runs as follows

- `prog1 &`
- `prog1 &`
- `prog1 &`

After that, you block three prog1 on the semaphore. You can use `jobs` to list how many background processes are in the background. Next, once you execute `prog2 &`, it will wake up one earlier blocked *prog1*. So you need to run *prog2* three times to wake up all the blocked *prog1*. When *prog1* is woken up, it shall print some message and their process id. So you should observe which process is woken up.

Project goal：

In this lab, three files p1.c p2.c and p3.c are provided. They are incomplete but simple. Each one will print a message. Please assume p1.c is always execute first. That is, p1.c is responsible for creating semaphore and p2, p3 are not responsible for creating semaphores.

Your goal is to use semaphores to coordinate p1,p2,p3 so that p1 prints message once, p2 prints message once, and then p3 prints message twice. They loop forever until loop exists. That is, suppose you run `p1 & ; p2 &; p3&`, your program output should be

```
P1111111
P2222222
P3333333
P3333333
P1111111
P2222222
P3333333
P3333333
…….
```

In order to achieve the goal, you need to think of some ways to use semaphore to coordinate the processes. Try to let p1 prints message once first. After that, p2 is allowed to print message. Next, p3 is allowed to print message twice.

NOTE THAT, TA may test your program by different order such as:

```
p1 & ; p3 & ; p2 &
```

But your program output should be the same.

NOTE THAT, when TA tests your program, he will use `ipcrm` to clear any existing semaphores. So, you should do so as well. Clear all the semaphores by `ipcrm` before running your program.

IMPORTANT CONSTRAINTS

To coordinate p1,p2 and p3, **you are only allowed to use P(), V()** for the lab. You are not allowed to use any other statements such as IF or use additional variables, etc. If you use other commands other than P() and V(), you will get 0

points.

(1) remember to killed bad processes when your experiment fails
(2) remember to clear your semaphore by `ipcrm`.

HINT：

In order to achieve the synchronization, please think how many semaphores should be used. Then use `create_sem()` in p1.c to create all the semaphores and their initial values. Then, in p2, p3, before the loop, use `get_sem()` to read the semaphores created by p1.

The initial values of semaphores are very important. Semaphore values must be greater than 0. There are no points in setting a negative semaphore value. Before and after the `printf` statements, please use P(), V() to achieve your goal. Good luck.

In `sem.c`, the following two functions are provided for you to observe and debug. You cannot use them in the final program. If you use, 0 point will be graded.

get_blocked_no(semid): return the number of processes blocking on the semaphore
get_sem_val(semid): return the semaphore value of semid. In UNIX semaphore, the semaphore value always greater than 0 but in OS text book, the semaphore value can be negative.

May force be with you.

~ Yoda，the Master