CSE 2431 **LAB 3** **AUTUMN 2018**

Due: Tuesday, October 9, 11:59 p.m.
3 pts + 2 extra pts

The source code and Makefile must be packed as a single zip file and submitted electronically online in Carmen Dropbox by 11:59 pm Tuesday 10/09/18. Late penalty will be 25% per day. Please complete your solution in the virtual machine you installed in Lab 0. The grader will compile and test your solution under the same environment. Any program that does not compile will receive a zero. The grader will NOT spend any time fixing your code. It is your responsibility to leave yourself enough time to ensure that your code can be compiled, run, and tested well before the due date.

Up to 2 extra points can be given to the implementation of a kernel module in the tasks below. In your submission, please create a subfolder for the kernel module. The code should compile in either Ubuntu 16 or Ubuntu 18. Please specify which version you tested successfully in your readme.txt.

1. Goal

Understand thread synchronization problems and practice the use of POSIX semaphores.

2. POSIX Semaphores

POSIX provides two types of semaphores - named and unnamed. For this project, we use *unnamed* semaphores to synchronize threads of the same process. To synchronize different processes, named semaphore can be used. If you are interested, you can try to learn by yourself.

The code below illustrates how an unnamed semaphore is created:

```
#include <semaphore.h>
sem_t sem;

/* create the semaphore and initialize it to init_value */
sem_init(&sem, 0, init_value);
```

The *sem_init*(…) creates a semaphore and initializes it. This function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

In this example, by passing the flag 0, we are indicating that this semaphore can only be shared by threads belonging to the same process that created the semaphore. A nonzero value would allow other processes to access the semaphore as well.

For the semaphore operations wait () and signal () discussed in class, POSIX names them sem_wait(…) and sem_post(…), respectively. The code example below creates a binary semaphore sem_mutex with an initial value of 1 and illustrates its use in protecting a critical section:

```
#include <semaphore.h>
sem_t sem_mutex;

/* create the semaphore */
sem_init(&sem_mutex, 0, 1);
/* acquire the semaphore */
sem_wait(&sem_mutex);

/*** critical section ***/

/* release the semaphore */
sem_post(&mutex);
```

3. Tasks

You are given the pthread implementation of the bounded-buffer problem you implemented in lab 1, where there are 1 producer and 1 consumer that communicate through a bounded buffer. As in lab 1, the producer will read from a file "input.txt" and put each line into one element of the shared buffer, and the consumer will read from the buffer and write each element into another file "output.txt" so that it will look exactly the same as "input.txt".

In this lab, you are expected to convert it into 3 producers and 3 consumers implementation using semaphores. The 3 producers will read a shared input file concurrently and write the line it reads into the bounded buffer; the 3 consumers will concurrently read from the bounded buffer and write the content into an output file that is shared by all consumers. The output file is expected to have exactly the same content as the input file, including the order of the lines (you should pay close attention to this requirement). Note: you should use three semaphores as suggested in the slides.

4. Kernel module (extra credit: 2pts)

You have been provided with two files, process.c and Makefile. Specially, process.c will list every process in the operating system by printing out their command line name and pid. Note, they are not printed to your terminal, rather, you have to run "dmesg" to see the results. Everything you "printk" since the beginning of system bootup will be shown using "dmesg", so if you want to clean up the previously printed data, reboot your virtual machine.

You are expected to extend the functionality of process.c. Instead of printing out all processes line by line, print the family tree of all processes. For example:

<pid_1> <command_line_name_1>
        <pid_2> <command_line_name_2>
        <pid_3> <command_line_name_3>
                <pid_4> <command_line_name_4>
                        <pid_5> <command_line_name_5>
        <pid_6> <command_line_name_6>
<pid_7> <command_line_name_7>

In this example, process 1 is the parent process of process 2,3,6. Process 3 is the parent process of process 4, and process 4 is the parent process of process 5. Process 7 is a sibling of process 1.

Therefore, if there are N processes in the operating system, your printed data will comprise N lines, with some lines indented to the proper position to indicate parent-child relationship.

Hint: You should work inside your virtual machine and reboot your virtual machine whenever the kernel crashes.