Elec. 377 - Operating Systems
Lab 3 - Process Communication

Lab Dates: October 17/23 and 30/31, 2018, Due: Nov 4, 2019 at 9:00am

Objectives

- Implement the pass the key algorithm for the critical section problem, allowing communication between user level processes using shared memory.
- Learn about separate compilation and common code used between multiple programs.

Introduction

In this lab we will not be programming inside of the kernel. Instead we will be using the virtual memory capabilities of Linux to map a single frame of physical memory into multiple processes virtual address space.

Most modern operating systems have shared memory capability. A process requests shared memory with a key, and then other processes may access the same shared memory given permission and the key. The same memory is mapped into each of the processes' address space, but it may not be mapped in at the same location in memory. So, the start of the shared memory may be at location `0x000c4628` in one process and the location `0x000c472c` in another process.

Synchronization

You will implement the pass the key algorithm to allow synchronization between two processes. You will write both a producer program that copies its standard input one byte at a time into a shared buffer and a consumer that retrieves a single character at a time from the buffer and writes it to the standard output. Your git repository contains 6 files to start with. These are:

1. `Makefile` The `makefile` to build the system
2. `meminit.c` This will compile to `meminit` You do not need to modify this file. Once compiled, running it will create the shared memory segment (200 bytes) and initialize its `null` values. It must be run between each test to reinitialize the shared memory segment.
3. `producer.c` This is the skeleton code for the producer. It contains the code to convert the command line to access the shared memory segment and map it into memory.
4. `consumer.c` This is the same as the producer, but for the consumer. It has the same code to convert the command line and to map the shared memory segment into memory.
5. `common.h` This will contain the definitions of the struct declaration used to

impose structure on the shared memory. The size of the data structure must be less than, or equal to, the size of the shared segment (200 bytes). It is included (using the `#include` directive) into both `producer.c` and `consumer.c` and into `common.c` Keep the buffer small (say 5 bytes) to assist you in your testing (see below)

6. `common.c` This file contains the skeletons for the *`getMutex`* and *`releaseMutex`* routines. You will implement the entry and exit code for the critical section in these two routines.

Rather than trying to map Linux PIDs, we will just assign the pids by passing an argument on the command line. The producer should contain a loop that reads each character (using *`getchar`*`()`) until it reaches the end of file (*`getchar`*`()` returns `EOF`). Since the solution uses busy waiting and does not put a process to sleep, you will have to use an inner loop that requests access to the critical section (using *`getMutex`*) then checks to see if there is room in the queue. If there is, it adds the character to the queue. It then releases the mutex. The inner loop is controlled by a flag that is set whenever the producer succeeds in adding a byte to the buffer. Remember that the `EOF` condition is an integer, not a character. So you will have to add a flag to the buffer that the producer can set to indicate that the end of the data has been reached.

Like the producer, the consumer will transfer a single character at a time from the queue to the output. It must also use nested loops, with a flag to indicate when it has successfully retrieved a byte from the buffer.

C has the ability to insert assembly langauge inline. On the x86 architecture, the atomic instruction is called cmpxchg which exchanges a value and tests it at the same time. An inline macro is defined in the include file <asm/system.h>.:

```
__cmpxchg(lock address, test value,  new value, size)
```

Note that there are two underscores at the beginning. It swaps the new value for the value in the address given by the first parameter and checks if the value that was retrieved is the test value. The macro returns the value retrieved. So the following implements test and set:

```
int test_and_set(int * lock){
   return __cmpxchg(lock,0,1,sizeof(int));
}
```

This function will provide the atomic test and set instruction needed by the pass the key algorithm.

Testing

Unlike previous labs, you have no oracle (an independent program that generates the right answers) with which to compare your results. Instead you will have to show the input for the producers and the output of the consumers to show that all of the values that are written to the shared buffer by each producer are in fact read once, and only once, by one of the consumers.

For testing, you can use multiple consoles (ALTF# to switch) to run multiple producers and consumers at the same time. You will have to devise a set of test data that will show

each byte is transferred a single time in the proper order from the producers to the consumers. In your implementation, limit the buffer to 5 bytes to ensure that a single producer cannot fill the buffer and exit in a single time slice.
You should confirm that the basics work simply by using "./producer 0" in one window or console and "./consumer 1" in another. What you type as input to the producer should show up in the consumer. You can use redirection in order to run the final test data. For example, you could run the three following commands in three separate windows (or three separate consoles)

```
./producer 0 < test00in.txt
./producer 1 < test01in.txt
./consumer 2 > test02out.txt
```

What to hand in

At the end of the first lab period, save your work so far to your git repository account as a backup.

After the end of the second lab period save your program and your test results to your git account. You have until 9:00am Nov 4 to write up your lab reports.

Your lab report must be comprised of two files. The first contains:

- A description of your program and how you solved problems.

The second file contains

- A description and explanation of your testing.
- Reference to your input and output files (which also need to be added to your git repository)

Both the program and lab report are to be checked into your git account. They must be plain text or pdf (Adobe Acrobat) files. Also, any test files you have generated (input and output) should also be checked into the repository.

Make sure that both your name and your partner's name are in the documentation.