Name: ____________________ **Student ID #:** ____________________

Lab #2 Solutions: COMP 3000 (Operating Systems)

Please answer all questions below. Part A is designed to be done in the lab while part B is designed to be done on your own time. Many of the questions require you to compile and run sample programs. These programs can be compiled using either the Corel Lab computers or your SCS Linux account.

For all questions asking you to modify source code, you should always start with a clean version of the source code and modify that (unless otherwise directed in the question). Don't continue modifying your solution to a previous question in order to answer the next question.

All programs given in this assignment are written in C. A makefile is provided to compile the core programs given by the assignment. If you wish to rename files, you will need to either edit the makefile or run GCC to compile the programs on your own.

1 Part A

This section is designed to be completed in the lab. You get 10% of the total marks for attempting to do part A during assigned lab hours. Please ensure that one of the Lab instructors takes your attendance.

1. **[8]** Have the instructor mark down that you were present and attempted part A during lab hours.

1.1 Processes and Threads

1. **[1]** The program `threads.c` is a multithreaded producer/consumer program. Unfortunately it consumes faster than it produces, resulting in an error. Why does it not print the same number every time?

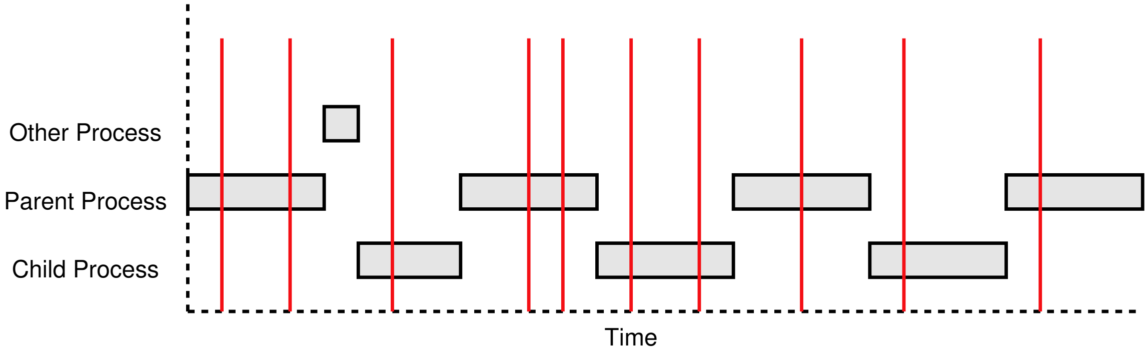
   This has to do with the fact that we have two processes running on a system without any sort of locking mechanism. This results in a race condition. If both processes ran for *exactly* the same amount every time you ran them, then the race condition would always appear at the same value. Unfortunately, this is not the case. There are a number of factors in a modern system which can affect the running time of a process. They include:

   (a) Other processes on the system

   (b) Hardware (which interrupts the CPU to get serviced)

   (c) The CPU, memory, cache

   Because of this, the scheduling of the two processes may look more like the diagram below, where the read lines indicate the servicing of an interrupt request. In order to always stop at the same number every time, each interrupt service would have to take *exactly* the same amount of time, and the processes would never be allowed to have their running time shortened or lengthened by even a few clock cycles. Being strict on execution time down to the clock cycle is not feasible on modern systems.



2. **[2]** The program `passstr.c` is a multithreaded program using the *clone* function call. What is wrong with the way this program blocks, waiting for the string to arrive in the buffer?

   The program is blocking by busy waiting on the first character in the string. Busy waiting is bad because it ties up system resources needlessly. Furthermore, it is looking at the first character in the string, which means if a process switch happened after the first character was written but before the rest of the string was written, then only the first part of the string would be displayed on the string (since the consumer would read the string before it was all completely there).

1.2 Fork & Exec

1. **[2]** What is the difference between the clone and the fork function call?

   The Clone function call is a much more powerful version of the fork system call. In fact, the functionality of the fork function call can be implemented with the clone function call. The clone function call, however, contains additional options which make it possible to create threads. The threads have the option of sharing file handles, memory spaces and other kernel resources.

1.3 IPC

Examine the program given in wait-signal.c. It multiplies two matrices together using the standard trivial algorithm (which also happens to be a n^3 algorithm). It spawns off a child process to compute the value of each element in the resulting matrix. The program has a problem, however, in that it fails to pass the resulting values back to the parent process in order to give the right result. In this section, we will examine various methods for passing data between processes.

1.3.1 Signals

Signals can be sent to each process running on the system. Signals, however, don't allow the passing of any data along with the signal. Therefore, they are most useful for triggering actions.

1. **[1]** The `kill` command actually sends signals to processes. What signal does the `kill` command by default send to a process?

The signal **SIGTERM** signal is normally sent unless a different signal is specified on the command line.

2. **[2]** Modify the *wait-signal-1.c* file to use the `signal` function to install the signal handler instead of the `sigaction` function call. You can have it install the `child_handler_alt` signal handler instead of the `child_handler` signal handler. What line did you add to install the signal handler to `child_handler_alt`?

```
signal( SIGCHLD, child_handler_alt );
```

3. **[2]** Modify the *wait-signal.c* file to ignore the abort signal. What line did you have to add to do this?

```
signal( SIGABRT, SIG_IGN );
```

1.3.2 Pipes

Pipes (also called FIFO's) allow two processes to communicate through a file handle. One process writes data into a file handle and the other process can then read that data out through a related but different file handle.

1. **[2]** Compile and run *pipe.c*. Notice how data is sent through the pipe by writing to one end of the pipe in the child and reading from the other end of the pipe in the parent. Also notice how the message *Finished writing the data!* is never displayed on the screen. The problem has to do with the SIGPIPE signal. What is the problem?

   When the parent closes the pipe, the child is sent a sigpipe signal when it tries to write to the pipe and the default action for a sigpipe signal is to terminate the program, hence the printf never runs.

   The program had another bug in that there was a `while(1)` loop which would have run forever had the process not been killed by the **SIGPIPE** signal. The `while(1)` loop, however, does not explain why the child died at all and the fact that the child exited at all should have been an obvious clue as to what was happening.

1.3.3 Shared Memory

1. **[1]** Shared memory regions are controlled by the kernel to prevent other processes from accessing the memory without permission. Like files in Unix, the shared memory regions are given read, write and execute permission. These permissions are specified in the call to `shmget`. Where in the arguments to `shmget` are the permissions specified?

   The lower 9 bits of the shmflg value.

2. **[1]** The permissions must be specified as a value. By reading the manpage of *chmod*, determine what the permission 0760 means.

- Read, Write and Execute for the Owner
- Read and Write for the Group
- No permissions for everyone else

3. **[1]** What number is going to be required in order for two processes owned by the same user to be able to read and write to the shared memory?

   0600 Any additional bits set represent a potential security issue in that additional processes not owned by the user may be able to interfere with the shared memory. The execute permission is not required in this question.

2 Part B

2.1 Processes

1. **[3]** From class, you know that the process descriptor contains numerous information about a running process on the system. The task structure in Linux is called `struct task_struct`. By examining the source of the Linux kernel, determine what source file this structure is defined in. The `grep` command may be useful in locating the correct file.

   include/linux/sched.h

2. **[7]** Figure 6.3 (page 213) in your textbook contains a list of common elements found in a process table. Determine at least one variable in the Linux task structure which is related to each element listed in Figure 6.3. You may omit address space and stack.

Internal process name	pid
State	state
Owner	uid, euid, suid, fsuid, gid, egid, sgid, fsgid
Execution Statistics	sleep_avg, last_ran, utime, stime, nvcsw, nivcsw
Thread	thread_info
Related Processes	parent, real_parent, group_leader
Child Processes	children

2.2 Fork & Exec

1. **[10]** Examining the flags that can be passed to the clone function call. Choose 5 flags and describe a situation in which each of them would be useful.

   - **CLONE_PARENT** - Useful when one child wants to create another process but have it be a sibling instead of a child. In this way, a complex application can have a parent which takes care of administration and any of the children would be able to create additional processes that are also taken care of by the parent. If this flag did not exist, the child would have to ask the parent to create the new process.

- **CLONE_FS** - chroot is often useful when you want to have processes only have access to a part of the file system. It restricts within the kernel what directories the program can access. If this flag is set, a server could be started with access to the entire file system and then after initialization access could be restricted to just those directories needed by the server during operation. As an example, a web server could be started and after it is running the parent could perform a chroot and the child would then be restricted to only the part of the file system containing files which should be displayed by the web server (there are security problems associated with open file handles when doing a chroot, but we do not discuss them here).
- **CLONE_FILES** - If the child was to do work on a file handle, the parent (or dispatcher) could open files and then tell the child to operate on a specific file handle. Since file handles are simply integers, passing a file handle back and forth between processes only works if this option is enabled.
- **CLONE_NEWNS** - Useful if you only wanted a new mounted drive to be accessible to certain processes. You could mount the drive in a new name-space and then all processes not in that name-space would not be able to access the mounted drive (since the drive would not be mounted in their name-space). It gives a way of protecting data on the drive when access permissions are not sufficient.
- **CLONE_SIGHAND** - Allows a parent to catch signals that should be destined for the child process. This is especially useful when creating threads, where only one signal handler should be present for the group of threads (you don't want different things happening depending on which thread generated the signal).
- **CLONE_PTRACE** - Allows debugging of children created by a parent process which is also being debugged.
- **CLONE_VFORK** - If a process wants to spawn off another program, it can use this option to block until the child has actually executed the new process. This prevents possible race conditions where the parent spawns off the child and then runs before the child has an opportunity to execute the replacement program.
- **CLONE_VM** - Sharing memory space is one of the basic desired traits when creating threads. It allows two threads to operate on the same data (with appropriate locking mechanisms of course).
- **CLONE_PID** - As it says, this is a hack. It allows creating a new process with the same PID so that when the parent exits, all other processes which refer to the process based on it's pid will still be talking to the correct process.
- **CLONE_THREAD** - This results in a thread which shares the PID of the process. It is useful in conjunction with the CLONE_VM function to create multiple threads which all share the same process ID and memory space.
- **CLONE_SETTLS** - Used by threading libraries. The description of this in the man page is not very clear.
- **CLONE_PARENT_SETTID** - Used by threading libraries, it makes the kernel write information on the thread to a specific location in memory, which is useful for keeping track of the threads.

- **CLONE_CHILD_SETTID** - Used by threading libraries, it makes the kernel write information on the thread to a specific location in memory, which is useful for keeping track of the threads.
- **CLONE_CHILD_CLEARTID** - Used by threading libraries, it forces the kernel to write information to the thread data structure in the parent when the child exits.

2. **[5]** Normally, file descriptor 0 is associated with standard in, file descriptor 1 is associated with standard out, and file descriptor 2 is associated with standard error. In your own words, explain what happens to file descriptors across an *exec* call and use this to explain how the vulnerability discussed in `http://security.nnov.ru/news1956.html` works.

   To produce the vulnerability, the parent closed the file descriptor associated with one of the outputs (either standard out or standard error). Across a standard exec call, open and closed file descriptors are maintained and so the new process would start with either standard error or standard out closed. When the process attempted to open a file, the opened file would be assigned the file descriptor typically associated with standard error or standard out. Then, if the program could be made to print information to either stderr or stdout (through the use of debugging messages as an example), the actual debugging messages would be written to the file opened by the process. If the process was running as a different user than the user who started it (which can be the case if the setuserid bit is set) then this would result in the process writing to files which the user did not have access to. Because a user can often control debugging output, a user could control what data is written to the file.

2.3 IPC

In this section, you will be modifying the program wait-signal to correctly compute the value of the matrix multiplication.

2.3.1 Signals

1. **[5]** Describe in words how you might modify the *wait-signal* program to correctly pass back the value computed in the child to the parent using only signals. Remember that signals do not allow data to be passed back and forth. Also keep in mind that there are only around 32 signals that can be sent to a process. You do not have to implement your answer, only describe what you would do.

   There are a few ways of answering this:

   - Count the number of signals received of a specific type and that is the value being passed back.
   - As an improvement, use two signals, one which increments the placeholder and the other which flips (or increments) the value pointed to by the placeholder. This can be done for any base (but base 2 is most convenient).

- As an even better solution, use two signals, one represents a 0 bit being passed and the other represents a 1 bit. So, a sequence of SIGUSR1, SIGUSR1, SIGUSR2, SIGUSR1 may be translated as 1101.

2.3.2 Pipes

1. **[10]** Modify the *wait-signal.c* program to pass the appropriate matrix data back to the parent via a pipe. Remember that you will also have to pass back the x and y locations that the data should be put in. What is your updated main function?

```
--- ../src/wait-signal.c        2007-01-22 18:34:35.706009269 -0500
+++ wait-signal-pipe.c  2007-01-22 18:32:52.395846640 -0500
@@ -78,6 +78,13 @@
     int matrixA[MATRIX_SIZE * MATRIX_SIZE];
     int matrixB[MATRIX_SIZE * MATRIX_SIZE];
     int matrixC[MATRIX_SIZE * MATRIX_SIZE];
+    int mypipe[2];
+
+    struct {
+        int x;
+        int y;
+        int value;
+    } retData;

     /* Initialize the matricies. */
     init_matrix( matrixA, 0 );
@@ -95,13 +102,21 @@
     printf( "Matrix B:\n" );
     print_matrix( matrixB );

+    pipe( mypipe );
+
     /* Start those children running... */
     for( y = 0; y < MATRIX_SIZE; y++ ) {
         for( x = 0; x < MATRIX_SIZE; x++ ) {
             pid = fork();
             if( pid == 0 ) {
                 /* We are the child. */
+                close( mypipe[0] );
+
                 int value = calc_elem( matrixA, matrixB, x, y );
+                retData.x = x;
+                retData.y = y;
+                retData.value = value;
+                write( mypipe[1], &retData, sizeof(retData) );
                 return value;
             } else {
                 /* We are the parent. */
@@ -109,9 +124,17 @@
             }
         }
     }
+    close( mypipe[1] );

     /* Now, we wait for all the children to finish. */
-    while( active );
+    do {
+        int datalen = read( mypipe[0], &retData, sizeof(retData) );
+        if( datalen == sizeof(retData) ) {
+            matrixC[retData.y * MATRIX_SIZE + retData.x] = retData.value;
+        }
+        if( datalen == 0 )
+            break;
+    } while( 1 );

     printf( "Result:\n" );
     print_matrix( matrixC );
```

2.3.3 Shared Memory

1. **[10]** Modify *wait-signal.c* to send data back to the main process using shared memory. You will need to use the functions `shmget` and `shmat`.

```
--- ../src/wait-signal.c        2007-01-22 18:34:35.706009269 -0500
+++ wait-signal-shm.c   2007-01-22 19:00:26.003349128 -0500
@@ -3,6 +3,8 @@
 #include <sys/wait.h>
 #include <signal.h>
 #include <unistd.h>
+#include <sys/ipc.h>
+#include <sys/shm.h>

 #define MATRIX_SIZE 5

@@ -77,9 +79,12 @@
     int pid, x, y;
     int matrixA[MATRIX_SIZE * MATRIX_SIZE];
     int matrixB[MATRIX_SIZE * MATRIX_SIZE];
-    int matrixC[MATRIX_SIZE * MATRIX_SIZE];
+    int * matrixC;

     /* Initialize the matricies. */
+    int id_shm = shmget( IPC_PRIVATE, sizeof(matrixC), IPC_CREAT | 0600 );
+    matrixC = shmat( id_shm, 0, 0 );
+
     init_matrix( matrixA, 0 );
     init_matrix( matrixB, 1 );
     memset( matrixC, 0x00, sizeof(matrixC) );
@@ -102,6 +107,7 @@
             if( pid == 0 ) {
                 /* We are the child. */
                 int value = calc_elem( matrixA, matrixB, x, y );
+                matrixC[y * MATRIX_SIZE + x] = value;
                 return value;
             } else {
                 /* We are the parent. */
```

2.4 Virtual Memory

1. **[10]** Describe how shared memory works at a virtual memory level.

   The same region of physical memory is mapped into the virtual address space of two different processes. The mapping in is done by updating the page tables of the processes involved. Permissions to the shared memory segment are enforced by setting the access permissions for the page in the page table.