Bicol University
College of Science
Computer Science Department

CS 114 – Operating Systems

Machine Exercise No.1

In Lecture 2, we described a program that copies the contents of one file to a destination file. This program works by first prompting the user for the name of the source and destination files. Write two versions of this program, one using the POSIX API and the other using standard C library functions only. Be sure to include all necessary error checking, including ensuring that the source file exists.

Once you have correctly designed and tested the programs, run the program using a utility that traces system calls. Linux systems provide the **strace** utility. Using **strace**, generate the trace files for these programs and compare the system calls used.

Machine Exercise No.2

A. Read the manual (`man`**) about the** `fork()` **,** `getpid()` **,** `getppid()` **UNIX/Linux API**

functions. Then write a program which creates one child process, and then both parent and

child are running in blind loops displaying their PIDs and PPIDs. This way you will have the

models of typical I/O-oriented processes. Perform some observations, how the CPU time is

shared between these two processes and how they are reacting for **`Ctrl+C (SIGINT)`** or default **`SIGTERM`** signal (sent from command line by kill shell command when running in

background – can be some fun). The output screen may look like this:

```
[PARENT]: PID 2374, PPID 2360

[PARENT]: PID 2374, PPID 2360

[CHILD]: PID 2375, PPID 2374

[PARENT]: PID 2374, PPID 2360

[CHILD]: PID 2375, PPID 2374

[CHILD]: PID 2375, PPID 2374

[PARENT]: PID 2374, PPID 2360
```

…

B. Read the manual about the wait() and exec() families of functions. Then write a program which creates one child process, loads new binary code (compiled from other source) for the child, and then parent waits (without using CPU nor I/O) until child finishes his task (counting from 1 to 10 and displaying counter status for example). The output screen may

look like this:

```
[PARENT]: PID 2421, waits for child with PID 2422

[CHILD]: PID 2421, starts counting:

[CHILD]: i = 1

[CHILD]: i = 2

...

[CHILD]: i = 10

[PARENT]: Child with PID 2422 finished and unloaded.
```

Machine Exercise No.3

Choose one problem to solve.

A. Design a program using ordinary pipes in which one process sends a string message to a second process, and the second process reverses the case of each character in the message and sends it back to the first process. For example, if the first process sends the message `Hi There`, the second process will return `hI tHERE`. This will require using two pipes, one for sending the original message from the first to the second process, and the other for sending the modified message from the second back to the first process. You may write this program using either UNIX pipes.

Sample run of the program:

```
$ ./me3.exe

Input string message: “Hi There”

PARENT(2506): Sending [Hi There] to Child

CHILD(2507): Recieved message

CHILD(2507): Reversing the case of the string and sending to
Parent
```

```
PARENT(2506): Received [hI tHERE] from Child
```

B. The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, ... Formally, it can be expressed as:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

Write a program that will have the child process output the Fibonacci sequence, by establishing a shared-memory segment between the parent and child processes. This technique allows the child to write the contents of the Fibonacci sequence to the shared-memory segment and has the parent output the sequence when the child completes. Because the memory is shared, any changes the child makes will be reflected in the parent process as well.

The program first requires creating the data structure for the shared-memory segment. This is most easily accomplished using a struct. This data structure will contain two items: (1) a fixed-sized array of size `MAX_SEQUENCE` that will hold the Fibonacci values and (2) the size of the sequence the child process is to generate- sequence_size, where `sequence_size` $\leq$ `MAX_SEQUENCE`. These items can be represented in a struct as follows:

```
#define MAX_SEQUENCE 10

typedef struct {

     long fib_sequence[MAX_SEQUENCE];

     int sequence_size;

} shared_data;
```

The parent process will progress through the following steps:

a. Accept the parameter passed on the command line and perform error checking to ensure that the parameter is $\leq$ `MAX_SEQUENCE`.
b. Create a shared-memory segment of size shared_data.
c. Attach the shared-memory segment to its address space.
d. Set the value of `sequence_size` to the parameter on the command line.
e. Fork the child process and invoke the `wait()` systen1. call to wait for the child to finish.
f. Output the value of the Fibonacci sequence in the shared-memory segment.
g. Detach and remove the shared-memory segment.

Because the child process is a copy of the parent, the shared-memory region will be attached to the child's address space as well as the parent's. The child process will then write the Fibonacci sequence to

shared memory and finally will detach the segment.

One issue of concern with cooperating processes involves synchronization issues. In this exercise, the parent and child processes must be synchronized so that the parent does not output the Fibonacci sequence until the child finishes generating the sequence. These two processes will be synchronized using the `wait()` system call; the parent process will invoke `wait()`, which will cause it to be suspended until the child process exits.

C program illustrating POSIX shared-memory API.

```
#include <stdio.h>

#include <sys/shm.h>

#include <sys/stat.h>

#include <sys/ipc.h>



int main()

{

      /* the identifier for the shared memory segment */

      int segment_id;



      /* a pointer to the shared memory segment */

      char *shared_memory;



      /* the size (in bytes) of the shared memory segment */

      const int size = 4096;



      /* allocate a shared memory segment */

      segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

```
            /* attach the shared memory segment */
            shared_memory = (char*) shmat(segment_id, NULL, 0);


            /* write a message to the shared memory segment */
            sprintf(shared_memory, "Hi there!");


            /* now print out the string from shared memory */
            printf (" *%s \n" , shared_memory) ;


            /* now detach the shared memory segment */
            shmdt(shared_memory);


            /* now remove the shared memory segment */
            shmctl(segment_id, IPC_RMID, NULL);


            return 0;
}
```