Operating Systems

Homework #2
Due: 2007/4/10

For the following programming exercises, please finish them by using C language and use the workstations at IM workstation laboratory to compile/execute them.

1. **The producer-consumer problem:** The source code of a producer-consumer program implemented using shared-memory can be got via

   http://graphics.im.ntu.edu.tw/~robin/courses/os07/code/03proc/producer-consumer.c

   Ensure that you can compile and execute the code and know the meanings of it. The producer-consumer algorithm described in the source code allows only *n*-1 buffers to be full at any one time. Modify the algorithm to allow all buffers to be utilized fully and also modify the source code to let the consumer consume the items in the buffer only if the buffer is full.

   Hint: Only some minor tweaks in the source code are needed.

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

   $fib_0 = 0$
   $fib_1 = 1$
   $fib_2 = fib_{n-1} + fib_{n-2}$

   Write a C program using the `fork()` system call that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child process. Because the parent and child processes have their own copies of the data, it will be necessary for the child to output the sequence. Have the parent invoke the `wait()` call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

3. **The Fibonacci sequence problem (again):** In the previous exercise, the child process must output the Fibonacci sequence, since the parent and child have their own copies of the data. Another approach to designing this program is to establish 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 to the shared memory will be reflected in the parent process as well.

   This program will be structured using POSIX shared memory as the example at

http://graphics.im.ntu.edu.tw/~robin/courses/os07/code/03proc/shm-posix.c

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` ≤ `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 flowing steps:

a. Accept the parameter passed on the command line and perform error checking to ensure that the parameter is ≤ `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()` system 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. The child process will then write the Fibonacci sequence to shared memory and finally will detach the segment.

4. **The popen function:** Write a C program using the `fork()` system call to produce a child process, which can execute a command given in the argument like the `popen()` system call and also print out the `PID` of the child process the parent process forked. For example, suppose your program is named by "`forktest`", this program could be run as:

```
./forktest ls
```

The child process then executes the specified command and the output will be like:

```
fork_sample.c  forktest  Makefile
child pid = 29745
```

Note : Your program must be able to take command argument**s**. For example, after

executing the program like "`./forktest ls -a -l`", your child process must execute the command "`ls -a -l`" instead of only "`ls`". Furthermore, if some of the commands may not be executed correctly in your program, it is okay as long as most basic commands (like `ls`) works.

BONUS PROBLEM

The mini-mini-shell: Write a C program to "replace" the original shell by extending the previous exercise, i.e., you can execute commands in this new shell interface like in a normal shell. It will print out the output normally after each execution, then prompt user about the next command without exiting the program. Of course, you must fork a new process to execute each command. For example, suppose your program is named by "`bonustest`", if the original shell looks like:

```
[b91056@mercury ~/homework]$ cd bonus
[b91056@mercury bonus]$ ls
bonus_sample.c  bonustest  Makefile
```

This is what your shell should look like:

```
[b91056@mercury bonus]$ ./bonustest
myshell> ls -al
total 24
drwxr-xr-x 2 b91056 student 4096 Mar 21 22:15 .
drwxr-xr-x 4 b91056 student 4096 Mar 21 22:00 ..
-rw-r--r-- 1 b91056 student 1979 Mar 21 22:15 bonus_sample.c
-rwxr-xr-x 1 b91056 student 6533 Mar 21 22:15 bonustest
-rw-r--r-- 1 b91056 student   46 Mar 21 22:00 Makefile
child pid = 31456
myshell> ps
  PID TTY          TIME CMD
31377 pts/0    00:00:00 tcsh
31454 pts/0    00:00:00 bonustest
31458 pts/0    00:00:00 ps
child pid = 31458
myshell> exit
[b91056@mercury bonus]$
```

Besides executing each command in your new shell, the `PID` of the child process should also be printed out, just like the previous exercise. Furthermore, your shell should support the following commands:

`pid`: print out the PID for this (parent) process.
`exit`: end the shell program and return to the normal shell environment.
`<enter>`: print out a new prompt line just like the original shell.

Note: You can add more useful commands to earn more bonuses. Since this bonus problem is considered simple, no part credit will be given. You have to meet all requirements in order to get the bonus.

Submission & Grading

The deadline is **4/10**. Please send your source code to one of the TAs by E-mail before 11:59PM of that day. The file name should be your student id like `B91705056.rar`. If you want to submit a newer edition of your code, please rename it like `B91705056_1.rar`. In the `rar` file, please include 1) the source codes, 2) an brief report about how you wrote your code, your personal understanding about the mechanism behind your work, and a series of test input/output.

Grading: Programs: 70%; Documentation: 30%.

NOTE

a. You can use any resources you found on the Internet, or discuss with others. However, you should not use any code or library which is not written by you, except for the system calls and build-in functions. **Any kind of copying will result in zero points.** (including the one who "shared" his/her exercises)
b. You can ask Robin or TAs if you have any question, but we will not help you to do the exercises anyway.
c. Any exercise after the deadline is not acceptable.