Sean Hinchee
CprE 308: Section G
Project 02
3/28/18

IPCS

Unnamed Pipe

Output of pipe_test.c

```
% ./a.out

My child asked "Are you my mummy?"

And then returned 42
```

What do you notice about the timing of the printing?

Timing had a slight delay, then, both print statements display.

What happens when more than one process tries to write to a pipe at the same time? Be specific: using the number of bytes that each might be trying to write and how that effects what happens.

POSIX.1 dictates that writes of less than PIPE_BUF bytes must be atomic. The output data is thusly written to the pipe as a continuous sequence. Writes greater in size than PIPE_BUF may not be atomic. In the case of nonatomic writes, the kernel may interleave the data written between processes. POSIX.1 dictates that PIPE_BUF is at least 512 bytes (on linux this is 4096 bytes). Blocking can be disabled or enabled if desired.

How does the output of pipe_test.c change if you move the sleep statement from the child process before the fgets of the parent?

The output with the sleep statement moved from the child to before the fgets of the parent:

```
% ./a.out
My child asked "Are you my mummy?"
And then returned 42
```

The original (unmodified) output:

```
% ./a.out
My child asked "Are you my mummy?"
And then returned 42
```

Thusly, there is no change between the outputs (tested on linux-7 server).

What is the maximum size of a pipe in linux since kernel 2.6.11?

The size is located in /proc/sys/fs/pipe-max-size and is 1048576 (16 pages).

Named Pipe (FIFO)

What happens when you run the echo command?

Text is written into the fifo and the write is blocked until something opens the fifo for reading.

What happens when you run the echo first and then the cat?

The cat reads and outputs the text being blocked on the echo write, releasing the echo writer.

Look at the man page fifo(7). Where is the data that is sent through the FIFO stored?

Data sent through the FIFO is stored in the kernel. Said data does not get written to the filesystem.

Write a short program that uses named FIFO (mkfifo(3)) to print any line entered into the program on one terminal out on the other terminal.

```
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#define PIPE_READ_END 0
#define PIPE_WRITE_END 1

int
main()
{
   FILE* f;

   mkfifo("./myfifo", 0666);
```

```
    while(1){
        f = fopen("./myfifo", "w");
        char usrin[512];
        scanf("%511s", &usrin);
        fprintf(f, "%s\n", usrin);
        fflush(f);
        fclose(f);
    }

    return 0;
}
```

Then use: **tail -f myfifo** to get output in another terminal.

Socket

What are the six types of sockets?

STREAM, DGRAM, SEQPACKET, RAW, RDM, PACKET.

What are the two domains that can be used for local communications?

AF_UNIX, AF_LOCAL

Message Queues

What is the output of mq_test1?

```
% ./mq_test1

Received message "I am Clara"
```

What is the output of mq_test2?

```
% ./mq_test2

Received message "I am the Doctor"

Received message "I am the Master"
```

Change mq_test2.c to send a second message which reads "I am X" where "X" is your favorite companion. Change mq_test1.c to wait for and print this second message before exiting.

```
% ./mq_test1

Received message "I am Clara"

Received message "I am Rose"



% ./mq_test2

Received message "I am the Doctor"

Received message "I am the Master"
```

Shared Memory Space

What is the output if you run both at the same time calling shm_test1 first?

```
-bash-4.2$ ./shm_test1
a_string = ""
an_array[] = {0, 1, 4, 9, 16}
a_ptr = 140725867627600 = "I am a string allocated on main's stack!"

-bash-4.2$ ./shm_test2
a_string = "I am a buffer in the shared memory area"
an_array[] = {42, 0, 0, 0, 0}
a_ptr = NULL
```

What is the output if you run both at the same time calling shm_test2 first?

```
-bash-4.2$ ./shm_test2
a_string = "I am a buffer in the shared memory area"
an_array[] = {42, 0, 0, 0, 0}
a_ptr = NULL

-bash-4.2$ ./shm_test1
a_string = ""
an_array[] = {0, 1, 4, 9, 16}
a_ptr = 140723208429184 = "I am a string allocated on main's stack!"
```

What if you run each by themselves?

```
-bash-4.2$ ./shm_test1
a_string = ""
an_array[] = {0, 1, 4, 9, 16}
a_ptr = 140728388814576 = "I am a string allocated on main's stack!"

-bash-4.2$ ./shm_test2
```

```
a_string = "I am a buffer in the shared memory area"
an_array[] = {42, 0, 0, 0, 0}
a_ptr = NULL
```

Why is shm_test2 causing a segfault? How could this be fixed?

shm_test2 did not segfault on the linux-7 server I tested on. The output when running it solo is shown above.

What happens if the two applications both try to read and set a variable at the same time?

They would overwrite each other in some manner, most likely.

How can a shared memory space be deleted from the system?

Close(2) called on the fd allocated by shm_open(3).

Change the code to share some useful piece of information?

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>

#include "shm_test.h"

int main(int argc, char** argv)
{
   int fd;
   fd = shm_open("/308LabIPC", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR | S_IRGRP); // open the shared memory area creating it if it doesn't exist
   if(!fd)
   {
         perror("shm_open\n");
         return -1;
   }
```

```
   if(ftruncate(fd, sizeof(struct SHM_SHARED_MEMORY)))
   {
         perror("ftruncate\n");
         return -1;
   }
   struct SHM_SHARED_MEMORY* shared_mem;
   shared_mem = mmap(NULL, sizeof(struct SHM_SHARED_MEMORY), PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0);
   if(!shared_mem)
   {
         perror("mmap\n");
         return -1;
   }
   if(close(fd))
   {
         perror("close\n");
         return -1;
   }
   int i;
   for(i = 0; i < 5; i++)
   {
         shared_mem->an_array[i] = i*i;
   }


   time_t rawtime;
   time (&rawtime);
   char my_string[512];
   sprintf(my_string, "%s", ctime(&rawtime));


   shared_mem->a_ptr = my_string;
   sleep(5);
   printf("a_string = \"%s\"\n", shared_mem->a_string);
```

```
   printf("an_array[] = {%d, %d, %d, %d, %d}\n", shared_mem->an_array[0],
shared_mem->an_array[1], shared_mem->an_array[2], shared_mem->an_array[3],
shared_mem->an_array[4]);

   if(shared_mem->a_ptr > 0)

   {

         printf("a_ptr = %lu = \"%s\"\n", shared_mem->a_ptr, shared_mem-
>a_ptr);

   }

   else

   {

         printf("a_ptr = NULL\n");

   }

}
```

This version of shm_test1.c provides the time to memory space.

Unnamed Semaphores

What is the function call that would be needed to create an unnamed semaphore in a shared memory space called shared_mem->my_sem and assign it an intial value of 5?

```
sem_init(&shared_mem->my_sem, 1, 5);
```

Named Semaphores

How long do semaphores last in the kernel?

POSIX semaphores are kernel persistent, meaning that even if no process has the semaphore open, the value is held.

What causes them to be destroyed?

`sem_unlink()` is the function by which to fully remove a semaphore (once the reference count is 0) from the kernel. Alternatively, the system could reboot.

What is the basic process for creating and using named semaphores? (List the functions that would need to be called, and their order).

```
sem_t *sem;

unsigned int myvalue = 1;

sem = sem_open("mysem", O_CREAT | O_EXCL, 0644, myvalue);

sem_wait(sem);

sem_post(sem);

sem_unlink("mysem");
```

Signals

What happens when you try to use CTRL+C to break out of the infinite loop?

```
% ./a.out
.....^?
Ah Ah Ah, you didn't say the magic word
```

What is the signal number that CTRL+C sends?

SIGINT.

When a process forks, does the child still use the same signal handler?

If fork() is called without a further exec (or similar), then then the child will use the same signal handler as the parent.

How about during a exec call?

If exec is called after fork, the child will use a different signal handler than the original parent.

Write two programs. One which will send a signal of number 42 to the other process. The other program should catch that signal and print out the message "I got the signal!"

sig0.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int main(int argc, char** argv)
{
```

```
   int pid;
   printf("pid: ");
   scanf("%d", &pid);

   kill(pid, 42);

   return 0;
}
```

sig1.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void my_quit_handler(int i)
{
   printf("\nI got the signal!\n");
   fflush(stdout);
}

int main(int argc, char** argv)
{
   printf("my pid: %d\n", getpid());
   signal(42, my_quit_handler);
   while(1)
   {
         printf(".");
         fflush(stdout);
         sleep(1);
   }
   return 0;
```

```
}
```

Dynamically / Statically Linked Libraries

First output of lib_test:

```
% ./lib_test

./lib_test: error while loading shared libraries: libhello.so: cannot
open shared object file: No such file or directory
```

Second output of lib_test after exporting the library:

```
sh-4.2$ export =`pwd`

sh-4.2$ $LD_LIBRARY_PATH

sh: /home/seh/cpre308/proj2/Project2-1/library: Is a directory

sh-4.2$ ./lib_test

Hello

World

World

World

i=42
```

Project 2

If you worked with someone else – who was it?

N/A.

Summary

The implementation was designed around a socket interface. The library has the address (127.0.0.1) and the port (13337) hardcoded (as does print-server). Each communication is processed into a generalized two step process: a command message (one of MKJOB or GETDRIVERS) followed by a data message. Data messages are formatted as ~-separated tuples (as a result, ~ is a restricted character in these transmissions).

If you did extra credit – tell us what the functionality and how to use it here:

Cli-printer supports all proposed options (as per run-*.sh scripts). Print-server supports daemonization (-d flag).

How to run Project 2

Terminal 1: ./src/print-server/printer

```
rm /drivers/*
make clean
make
./virt-printer –n printer0
./virt-printer –n printer3
```

Terminal 2: ./src/print-server

```
make clean
make
./main -d
```

Terminal 3: ./src/libprintserver

```
make clean
make
```

Terminal 4: ./src/cli-printer

```
make clean
make
export LD_LIBRARY_PATH="../libprintserver/"
LIST FUNCTUION:
   ./cli-printer –l file_name
PRINT FUNCTION
   ./cli-printer –d driver –s description –o output_name file_name
```

Notes:

A nice demonstration is available for cli-printer via run-cli-printer-print.sh

NOTE: The file_name argument for cli-printer **must** be an absolute path. That is, as demonstrated in `run-cli-printer-print.sh`, the format `$(pwd)/file.ps` is recommended for the argument.

An example run of cli-printer is shown below:

```
% ./run-cli-printer-list.sh
Data: samplec.ps
LIB: Connected to server socket.
LIB: Waiting on server to write list…
LIB: Reply received.
printer_name=./drivers/printer0-r
printer_name=./drivers/printer3-r


% ./run-cli-printer-print.sh
Data: /home/henesy/repos/cpre3##
Driver: color
Description: description
Output: first_test.pdf
LIB: Connected to server socket.
LIB: Jobs successfully transmitted to server.
Data: /home/henesy/repos/cpre3##
Driver: black_white
Description: more exposition
Output: second_test.pdf
LIB: Connected to server socket.
LIB: Jobs successfully transmitted to server.
```

Here we see that for the list command script, printer0 and printer3 are the loaded printers for the server (accurate).

We also see in the print command script, in both cases a file (with a truncated path in this case, it’s samplec.ps) is transferred to the server for printing. If we check the server console (running without daemonization in this case, the server is capable of being a daemon), we see that both files printed:

```
...
consumed job second_test.pdf
```

```
consumed job first_test.pdf
...
```

To be absolutely sure we can reference the virtual printer console:

```
./start
PRINTERNAME: ./drivers/printer0-r
Switching to background
PRINTERNAME: ./drivers/printer3-r
Switching to background
% ##NAME##
##DESCRIPTION##
##LOCATION##
##NAME##
##DESCRIPTION##
##LOCATION##
##NAME: second_test.pdf##
##NAME: first_test.pdf##
##END##
##END##
```

Here we see that both files were processed by the virtual printer. A quick ls (in this case lc) confirms that the files exist:

```
% lc
.nfs000000000001112500000491  run-virt-printer.sh
.nfs000000000001119900000048a  second_test.pdf
.nfs000000000001133200000489  setup-virt-printer.sh
.nfs0000000000011e56000001d5  start
.nfs000000000001200e000001e1  virt-printer
Makefile                                virt-printer.c
drivers                                 virt-printer.o
```

```
first_test.pdf
%
```

We can see that both files were created!