CPRE 308 Lab 4

Reid Schneyer

Summary

In this lab, I learned about how different processes can communicate with eachother, such as with signals, piping, and shared memory.

3.1: Introduction to Signals

1. When you type `^C`, the interrupt signal is sent to the running program. This usually terminates the running program, but the code is set up so that typing `^C` instead runs `my_routine()`, thanks to the `signal(SIGINT, my_routine)` line in the code.
2. The program exits normally when the `signal(...)` statement is removed. By default, this causes the program to terminate.
3. Replacing `signal(...)` with `signal(SIGINT, SIG_IGN)` causes the interrupt signal (`^C`) to be ignored, so nothing happens.
4. Similar to the original code, we have "overridden" the behavior of `SIGQUIT`, so that it now runs `my_routine` instead of killing the process.

3.2: Signal Handlers

1. `^C`'s int value is 2, and `^\`'s int value is 3. `^C` generates `SIGINT`, and `^\` generates `SIGQUIT`.

3.3: Signals for Exceptions

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

void div_by_zero_catch();

int main(){
    signal(SIGFPE, div_by_zero_catch);
    int a = 4;
    printf("Going to divide %d by 0\n", a);
    a = a/0;
}

void div_by_zero_catch(){
    printf("Caught a SIGFPE!\n");
    exit(1);
}
```

The `signal()` statement should come before the divide by 0 code. If it comes after, then the code for overriding the `SIGPFE` won't get run until after you try to divide by zero.

3.4: Signals using `alarm()`

1. The input params to this program are `argc` and `argv`. The second argument (`argv[2]`) is converted to an integer with `atoi()`, and then passed as an argument to `alarm()`. The first argument (`argv[1]`) is copied into the `msg` variable with `strcpy()`, and is later printed using `msg` in the `my_alarm()` function.
2. `alarm(time)` causes the `SIGALRM` signal to be generated after `time` seconds. Since we "overrode" the `SIGALRM` signal in our code, `my_alarm()` is run instead.

3.5: Signals and fork

1. Two processes are running
2. The forked process prints out the `Return value from fork = 0` message, and the parent process prints out the `Return value from fork = n`, where n is a non-zero number.
3. Two processes recieved signals

3.6: Pipes

1. Two processes are running, the parent process is the `if` block, and the child process is the `else` block.
2. The message is initialized in the array `msg`, then written to the pipe in the parent's `if` block. The child then runs the `else` block, which waits, then reads `msg` from the pipe, and prints it with `printf()`.
3. Without the `sleep()` statement, the program pipes and prints the message almost immediately, before the program ends.

3.7 Shared Memory Example

1. Both programs have a `key_t key` variable that they both set to `5678`, and then use `shmget()` to get that shared memory.
2. If the client is started before the server, or if the server takes too long to generate the message, an incorrect message will be recieved by the client.
3. Running the client without the server results in the message that the client reads is similar, but with a '*' replacing the first character.
4. After adding those two lines, the programs function normally when `server` is started, then `client`, but if only `client` is ran, it prints: `shmget failed: No such file or directory`. The two added lines detach `shm` from the shared memory, and `shmctl()` with the `IPC_RMID` argument marks the shared memory segment to be destroyed.

3.8 Message Queues and Semaphores

1. The amount of data that can be sent in a single message depends on the value of the `msgsz` argument passed to the function.
2. A process that tries to read from a message queue that has no messages of the requested type will
3. Shared memory can be destroyed with the `shmdt()` function, and message queues can be destroyed with `msgctl(queueID, IPC_RMID, NULL)`
4. These are counting semaphores.