Quiz #3, on Wednesday, October 3rd, will consist of questions taken from or inspired Parts I and II of this homework.

Part I — Questions

1. How many parents do a process have? Generally, can a process have more than two children?

2. What are the steps performed by an OS to create a new process? Do their order matter?

3. What is a `pid`?

4. What does the `getppid()` function return?

5. What is an orphan? What is a zombie?

6. What does it mean to preempt a process?

7. What is swapping and what is its purpose?

8. Does message-passing involve the kernel?

9. Name two ways of sending a message from a process P_1 to several processes $P_2, \ldots, P_n$.

10. What is a rendez-vous between two processes?

11. Why does a system have many ports?

12. What are the differences between named pipes and ordinary pipes?

13. What does it mean for a process to *lock* shared data? How is it useful?

14. Name the two common models of communication between processes. If I want to transfer large data between two processes, which one is the fastest?

Part II – Problems

I'll assume that you will have successfully completed those tasks by Wednesday, October 3rd, so don't wait and let me know if you had difficulties solving them. Make sure you completed Problem 1 before the first exam.

You can find the code given below (as well as a possible solution to Problem 3) at `http://spots.augusta.edu/caubert/teaching/2018/fall/csci3271/code/hw3.zip`.

Problem 1

Look at the code displayed in Listing 1, and answer the following:

1. What is the datatype `pid_t` used for?
2. If you execute this program how many processes would be created? How many children would have the process you created by executing the program?

3. Will "Where am I?" ever be printed? If yes, how many times, and by who: the parent, a child, some other process?
4. Will "Hi Dad!" ever be printed? If yes, how many times, and by who: the parent, a child, some other process?
5. Will "Hi Mom!" ever be printed? If yes, how many times, and by who: the parent, a child, some other process?
6. If you want to use the `execl` function instead of `execlp`, what would you need to change?
7. If you want to load `ls` with the argument `../` instead of simply `ls` without arguments, what do you need to change in that program?

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(){
        pid_t pid;
        pid = fork();
        if (pid == 0)
        {
                fork();
        }
        else
        {
                printf("Where am I?\n");
                execlp("ls", "ls", NULL);
                printf("Hi Mom!\n");
        }
        printf("Hi Dad!\n");
        exit(0);
}
```

Listing 1: fork.c

Problem 2

Run your virtual machine, created a "03" folder in your "Desktop/HW/" folder, and copy the code located at "Desktop/osc9e-src/ch3/unix_pipe.c" in it, or copy-and-paste it from Listing 2. Optionally, you may want to install a nicer IDE using

```
sudo apt-get install geany
```

and typing "osc" as password.

1. Examine the code `unix_pipe.c`, compile it, and run it. What do you observe?
2. Now, let us look at the code again:
   (a) What does `pipe(fd)` do? What value does it return? Use `man pipe` to answer those questions.
   (b) Change the value of the constant `BUFFER_SIZE` to 6, compile and execute the program. What do you observe? Change it back to 25.
   (c) Switch the values of `READ_END` and `WRITE_END`. Compile and execute the program: what happened?

(d) (optional) Try to modify your program, so that is actually check that the pipe is still open before writing in it or reading from it. You can get inspiration from this question: `https://stackoverflow.com/a/19020926/`.

```c
/**
* Example program demonstrating UNIX pipes.
*
* Figures 3.25 & 3.26
*
* @author Silberschatz, Galvin, and Gagne
* Operating System Concepts  - Ninth Edition
* Copyright John Wiley & Sons - 2013
*/

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

#define BUFFER_SIZE 25
#define READ_END        0
#define WRITE_END        1

int main(void)
{
        char write_msg[BUFFER_SIZE] = "Greetings";
        char read_msg[BUFFER_SIZE];
        pid_t pid;
        int fd[2];

        /* create the pipe */
        if (pipe(fd) == -1) {
                fprintf(stderr,"Pipe failed");
                return 1;
        }

        /* now fork a child process */
        pid = fork();

        if (pid < 0) {
                fprintf(stderr, "Fork failed");
                return 1;
        }

        if (pid > 0) {  /* parent process */
                /* close the unused end of the pipe */
                close(fd[READ_END]);

                /* write to the pipe */
                write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
```

```
                /* close the write end of the pipe */
                close(fd[WRITE_END]);
        }
        else { /* child process */
                /* close the unused end of the pipe */
                close(fd[WRITE_END]);

                /* read from the pipe */
                read(fd[READ_END], read_msg, BUFFER_SIZE);
                printf("child read %s\n",read_msg);

                /* close the write end of the pipe */
                close(fd[READ_END]);
        }

        return 0;
}
```

Listing 2: Desktop/osc9e-src/ch3/unix_pipe.c

Problem 3

Run your virtual machine, and open two terminal.

1. In the first one, execute `top`. In the second terminal, find the `pid` of `top`. The right combination of `ps` and `grep` should give you that answer easily.
2. In that second terminal, execute `man kill`, and find the instruction to list all the signal names. Once you found it, exit the manual by typping `q`.
3. Send the signal 26 to top using `kill`. What do you observe?
4. Run `top` again in the first terminal, and send it another signal.

Problem 4

Copy the code displayed in Listing 3. Do the following:.

1. Compile and run the program, then send it the `SIGINT` signal, using `Ctrl` + `C` or the `kill` program.
2. Add the statement

   ```
   signal(29, sig_handler);
   ```

   after "`signal(SIGINT, sig_handler);`", compile and run the program, then send it the signal 29.
3. Modify your program so that it would print "Time's over" then quit when it received the signal `SIGALRM`.
4. Now, write a handler for `SIGKILL`, compile and run your program, and send it a `SIGKILL` signal. You should get a "Illegal instruction" error message printed (or, at least, your handler won't be executed). Can you guess why? Look for the answer if needed.

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

void sig_handler(int signo)
{
        printf("Caught signal %d, coming out...\n", signo);
        exit(1);
}

int main(void)
{
        printf("My pid is %d.\n", getpid());
        signal(SIGINT, sig_handler);
        while(1)
        {
                sleep(1);
        }
        exit(0);
}
```

Listing 3: signal.c