Computer Science 315
Laboratory 2 - Creating Processes
Due: 7:00am Monday February 6, 2023

A process can create or "spawn" other processes. This is the mechanism that Unix uses to activate the various resource management and process management processes which make up the operating system. Process creation is also available to user programs via several methods.

The system function:

One way to create a new process is with a call of the system function. It can be used to run a process which the shell program might normally activate.

For example:

```
#include <stdio.h>
#include <stdlib.h>   // For system function
int main() {
  system("date -u");
  return 0;
}
```

A single string argument is passed from the user program to the shell program. The string may be a literal string (as above), or a variable string in which we have stored the appropriate request. The shell program is simply a command interpreter with which the user interfaces when they log in. In this example, we are passing the shell the string "date -u", which the shell recognizes as a request to display the current date and time in GMT. The shell causes a new process to be spawned to execute a program which displays the date. When the date program is finished, the spawned process is terminated, and execution continues in the original process.

Try out the above code.

The execl function:

The execl function can be used to spawn another process without returning to the original process. A programmer might use it as the last action in a running program. For example, to display the date using execl we would write:

```
#include <stdio.h>
#include <unistd.h>   // For execl function
int main() {
  execl("/usr/bin/date", "date", "-u", NULL);
  return 0;
}
```

The first argument of execl is the filename of the command you want executed. You have to know where it is found in the file system. Before writing this program, you would probably check to see where the date program is stored by using the which program:

```
$ which date
/usr/bin/date
$
```

which tells us that there is a file called date in the /usr/bin directory (that's the date program).

The second argument of execl is the program name (that is, the last component of the file name). If this seems redundant to you, you're right. If the command takes arguments, they follow the program name. The end of the arguments is marked by a NULL pointer.

When you call execl, the new spawned process (in this example, date) will actually overlay the existing process which called it, and then begin executing. That's why there is no return from this call. The code of the original calling process has actually been overwritten! The only exception to this is if there is an error in calling the program specified in the execl call. To see this, when the code below runs, the printf statement doesn't execute:

```
execl("/usr/bin/date", "date", "-u", NULL);
printf("Someone stole 'date'\n");
```

Remeber that you have to include stdio.h in order to have access to the printf() function.

The fork function:

Most of the time, we don't want our original process to be overwritten. Instead, what we would like to do is to spawn a process that continues on its own without interfering with the calling process. To accomplish this we use the fork function, which splits a process into two separate processes:

```
int proc_id;
......
......
proc_id = fork();
```

From this point on, there are two copies of the same program running. The only difference between the two is the value of the integer returned by fork. In the original program (the *parent* process), the value of proc_id above will be the Process ID number which was assigned to the new process (the *child* process) by the operating system. In the child process, the value of proc_id will be 0. In all other respects, the two processes are identical. The child gets an exact copy of all variables which the parent declared, including their values. To illustrate this, consider the following program:

```
#include <stdio.h>
#include <unistd.h>
int main() {
  int x = 5;
  printf("Before fork\n");
  fflush(stdout);
  fork();
  printf("The value of x is %d\n", x);
  return 0;
}
```

The output from this short program should be:

```
Before fork
The value of x is 5
The value of x is 5
```

Each of the two processes (the child and the parent) will execute code which follows the fork, since there are now two separate processes running. The call of fflush was to ensure that the output buffer was flushed before we got to the fork. If the output buffer hadn't had time to be flushed, then the child process would also output the phrase "Before fork", since it inherits an exact copy of the output buffer too. (see Experiment 3)

More commonly, we want the child to do something different from what the parent is doing. Therefore, we embed the child's code within an if structure:

```
proc_id = fork();
if (proc_id == 0) {
  child's processing goes here
  ...
  return 0;
}
parent's processing is here
```

Recall that to the child process the value returned by fork is 0, and that the value is something other than 0 to the parent (it would be the actual system Process ID of the child). The call of return 0 in the child's code terminates the child process and properly closes any buffers that the child might have been using (for example, the output buffer for stdout).

If the child is not terminated, it will continue execution of the code after the if statement. So it must be explicitly closed.

Communication between processes (the pipe function):

Since a call of fork results in an exact copy of the program being activated, processes don't actually share storage. The child gets an exact copy of the variables declared by the parent in the exact state they were in when the fork was executed. Thus, if a child modifies a variable, it has no effect on the parent's copy of the variable.

Sometimes we would like to be able to have a child send a result to the parent process. To do this, it will have to send a message containing the result in the form of a character string to the parent. This is accomplished through the use of *pipes*. A pipe is a communication path between processes. Processes can read from pipes and write to pipes. The pipe function establishes a pipe:

```
int p[2];
pipe(p);
```

A pipe is actually implemented as an array of two integers. Each integer represents a so-called file descriptor, one for reading from, and one for writing to. The call of pipe sets the values of p[0] and p[1] to the numbers of the next two available file devices on the system. The file descriptor 0 is already assigned to the file stdin, the default input device. The descriptor 1 is, by default, assigned to the file stdout. The descriptor 2 is assigned to the standard error device, stderr, which usually is the screen. Thus, a call of pipe usually results in p[0] getting the value 3 and p[1] getting the value 4. If we have more than one pipe active, then the file descriptors would be greater than 4.

A process can read a message from another process by performing a read operation on the file designated by the file descriptor stored in p[0]. A process can send a message to another process by writeing to the file designated by the file descriptor stored in p[1]. For example:

```
char buf[10];
int p[2], proc_id;
pipe(p);
proc_id = fork();
if (proc_id == 0) {
  sprintf(buf, "%s", "Hi Mom!");
  write(p[1], buf, 10);
  ...
  return 0;
}
read(p[0], buf, 10);
printf("Message received was '%s'\n", buf);
...
```

A message buffer is normally used by each of the processes to build and receive messages (buf in this example). As discussed in class, the sprintf function is useful in building messages - it is a function that does a formatted print (like printf), but to a string rather than to standard output. The string that is the target of the print in the above example is buf. Remember that after the fork, each process has its own copies of all variables, including buf. The write function takes as arguments a file descriptor, followed by a string, followed by the length of the string. The read function takes similar arguments. If a process executes a read and nothing has arrived yet, it waits until something does arrive. Similarly, if a process writes to another process and the other process isn't reading yet, it simply waits until the other process does its read operation. This waiting is called *blocking*. In this case it is important that the writer waits, because the writer process may later change the contents of buf, which would mean that the writer wouldn't be sure which value of buf would be read by the reader.

Pipes are actually implemented at a lower level by a structure called a *socket*. We will be looking at sockets in a later lab.

Add some code to the program above to print the value of the file descriptors assigned to the pipe. Are the values 3 and 4 as expected?

Note: To make sure that a parent process waits for its child, provide the wait() function call before returning from the parent.

Experiments:

Place all your answers and any code that you write in a report file called lab2.txt, to be submitted when you have completed all the experiments.

1. Does a call of system overwrite the original process or not? Construct a small test program to find out, and explain how your program answers the question.
2. Rewrite 1) using a call of execl. Does execl overwrite the original process?
3. Try the program (mentioned earlier in this lab, in the material about fork) which outputs the value of x. Test the effect of removing the call of fflush. Do two copies of the phrase "Before fork" appear? Try redirecting the output of this program to a file. Remember that in bash, you redirect the output of a command to a file by ending the command line with "> myfile". For example,

   ```
   a.out > foo.txt
   ```

   What do you observe? Put your observations in your lab2.txt file.
4. Write a small program which forks into two processes. Have each process output its own process number, the Process ID of its parent, and the Process ID of its child (if it has a child process). A process can get its process number from the operating system with a call of the function getpid(). There is also a system function for finding the Process ID of a process' parent. (Hint: check out the man getpid pages to find out what it is.)

   Note: Make sure you add some text to your output so that it would be clear to anyone reading your output, which process is responsible for printing each line of output.
5. Write a small program which sets up a pipe. What file descriptors are assigned to the pipe? Print their values out.
6. Write a program which sets up communication between a parent and child. Have each process send its relative a message. Have the parent send the child the message "*Have you finished your history homework?*" and the child respond with the message "*Almost. How did the Vikings send secret message?*". Finally, have the parent respond with "*I think it was by Norse Code*". Ensure that the messages are arriving by printing them out after they are received. Use two pipes.

7. Examine the code shown below. Run this code and examine the output. You should see something in the output that is unexpected. (Hint: examine the address and value of the variable x in the *parent* and *child* processes.) Later we will learn about how memory is managed, which will explain the unexpected output, but for now your job is to just describe the apparent contradiction. Specifically, does there appear to be any memory location that stores two different values? Explain. **Do not include the C code in your lab2.txt file - just the output**.

   Note: the %p format specifier is used for printing an address value. Use it to determine the value of a pointer, or the address at which a variable is stored.

```
#include <stdio.h>
#include <unistd.h>     // for fork()
#include <sys/wait.h>   // for wait()

int main() {
  int x = 88888;
  int *z = &x;
  printf("x = %d,  Address of x = %p\n\n", x, &x);
  if (fork() == 0) {
    *z = 12345;
    printf("In child:  x = %d,  Address of x = %p\n", x, &x);
    printf("The child process is ending.\n\n");
    return 0;  // Forces the child process to end here
  }
  wait(NULL);  // Ensures output is sequenced
  printf("In parent: x = %d,  Address of x = %p\n", x, &x);
  printf("The parent process is ending.\n\n");
  return 0;
}
```

Submit:

Place all your test programs (and output) and answers to the questions in a file called lab2.txt on Moodle. This is a HARD deadline.