

COMP2401A—Tutorial 8

Processes

Learning Objectives

After this tutorial, you will be able to:

- Clone a program (create child processes using `fork()`)
- Gain insight into basic management of processes
- Send signals between processes
- Use `wait` and `waitpid`

Download the file `t8.tar` and extract the tutorial files.

1 Spawning new Processes

Purpose: Creating new processes

When a program is invoked then the OS is starting a new execution process. The `fork` command clones the original program resulting in two identical programs – a parent process/program and a child process/program. The two programs are identical (as if they are identical twins) and the execution thread of each (instruction set) continues from the same location in the code – namely immediately after the `fork` command. Note, that the cloning includes the memory where all variables are cloned. In particular note that dynamic memory allocations are cloned as well. This is because the memory heap is being cloned.

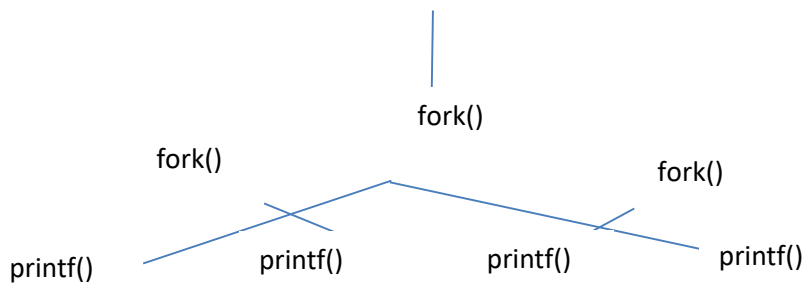
1. Creating multiple children

Review the file `fork0.c`. The file has one call to `fork()`. When the function is executed a cloned process is created. Therefore, after the call there will be two programs that execute the same code.

- a. Compile the code in `fork0.c` and execute it. How many times does the statement “after `fork()`” appears on the screen?
- b. Modify the program in the file to call the function `fork()` twice as
`fork();`
`fork();`

Before compiling the determine how many times will the statement “after `fork()`” be printed? Why?

Test your answer by compiling the file.



The diagram above illustrates what happens at each line of code. The first `fork()` creates a clone and the two programs continue at the next line of code which is `fork()`. Again the code duplicates each process and therefore there are four processes each executing the code line `printf()`.

2. Fork process and parent child relationship overview

The `fork()` function instructs the OS to clone the program. The `fork` command returns either 0 in the cloned program, termed child process, and the process id of the newly created cloned one, to original program, termed parent process. If an error has occurred the system returns a negative number. For example:

```

pid = fork();
if (pid == 0) {
    // child process instructions
} else if (pid > 0) {
    // parent process instructions
} else {
    // error pid < 0
}

```

3. Fork a process

Study the file `fork1.c`. You can compile it and run it. Note that the parent and the child execute the same code after the `fork`. Augment the code in `fork1.c` so that the child and the parent execute different code. Use `printf` statements to print the parent process id and the child process id using the functions `getpid()` and `getppid()`;

The function `getpid()` returns the process id (process number) of the process. The function `getppid()` returns the process id of the parent process. Recall that in a tree-like parent-child relationship a parent may have many children but each child can have only one parent (hence the two functions).

```

pid = fork();
if (pid == 0) {
    // child process instructions
}

```

```

    printf("Child process pid=%d  parent process id=%d \n",getpid(),
getppid());
    return(55);

} else if (pid > 0) {
    //  parent process instructions
    printf("\t Parent process pid=%d  child process id=%d
\n",getpid(), pid);
    return(0);
} else {
    //  pid <  0 - an error occurred during the fork operation
    printf("\t ERROR - parent process pid=%d  could not fork a child
process \n",getpid());
    return(1);
}

```

Note, remove the for loop at the end of the code.

2 Managing Processes - Viewing and Controlling Processes

When a program is invoked by the user (e.g., a shell command), the operating system allocate resources to the new program – memory space for the program instructions, memory space for the data (e.g., static and dynamic), stack space for function calls, and heap space. Once the space is allocated the program is loaded and into the memory and program starts its execution. The loading of the program onto memory and starting the execution is termed a process. Modern operating systems are multitasking allowing more than one program running simultaneously.

1. View processes

- a. Viewing processes - In a new terminal type *ps*. This command lists the processes that are currently executed by the OS. You will probably see only a few processes.
- b. Viewing all processes – type in the command line *ps -ef*. This command will print all processes that are currently active. The list will most likely be quite long.

View the list by piping the output of the above command into “more” - ps -ef | more. View the different columns. Important columns to view are: user (who is the owner), process id (the process number), parent process id (the id of the parent process), and execution time.

2. Managing Processes

- a. Review the files *while1.c* and *loop.c* – these files are executing an infinite loop. The purpose of the file is to ensure that they can continue to run indefinitely.
- b. Compile the files and create corresponding executable files (e.g., *gcc -o while1 while1.c*)
- c. Start the program *while1*. The shell will be occupied executing your program. Open another shell (terminal) and type *ps -ef* to view it. Note that the shell process is the parent of your

- program process. If you wish to see then find the process id of the parent (e.g., ppid is 1234) and then type `ps -ef | grep 1234`.
- d. Suspend the program while1 by typing `<ctrl> z` in the terminal where while1 is executing
 - e. View the processes – type `ps`. You will see the process of your program but the time will not advance.
 - f. Start the program loop. However this time execute it in the background. This is accomplished by using the “&” as the last command line parameter – e.g., `loop &`. This will run the program in the background.
 - g. Type `ps` again to see that the two programs while1 and loop are listed. Note that the program while1 is stopped and program loop is running. This can be inferred by looking at the time that each process is executing.
 - h. **Jobs** – you can the status of the programs that you have started by typing `jobs`. Here you will see the two jobs (while1 and loop) one as stopped and one as running. Note that each listed job has a number
 - i. **Restarting the suspended job** – find the job id of the suspended program by typing `jobs`. Here you will see the id of the job (e.g., job id is 2). Once you have the program id (which is not a process id) then type `bg 2` and the job will resume execution in the background.
 - j. **Moving a program to the foreground** – type `fg <job id>` in the previous example it will be `fg 1`
 - k. **Stopping a program** – you can stop a program by using the kill command. The kill command tells the system to send a signal to the process. If the program runs in the foreground typing `<ctrl>c` will kill the program. If the program runs in the background you can do one of three things:
 - i. Find the job number when using the command `jobs`. Use the kill command as follows: `kill -9 %jobid` where jobid is the number of job.
 - ii. Find the process id of the program and then use the kill command as follows `kill -9 pid` where pid is the process id of the program.
 - iii. Bring the program to the foreground and then using `<ctrl>c`

3 A Forking Exercise

Given the following code:

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

int main(int argc, char *argv[])
{
    int pid;
    int cpid;
    int numForks;
    int i;
```

```

if (argc == 1) {
    numForks = 2;
} else {
    numForks = atoi(argv[1]);
}
pid = getpid();
printf("Parent=%d\n",pid);
for (i = 0; i < numForks; i++) {
    fork();
}

pid = getpid();
printf("My process id=%d\n",pid);
return(0);
}

```

How many times will the output “My process id...” be executed?

What would happen if numForks is set to 3?

How would you set the program so that only the parent program will spawn the child process?

Namely, if numForks=2 then only two child processes will be spawned. Hint: ensure that child process will abort the “for” loop.

4 Sending signals

1. Compile **send.c** and **handle.c** with targets **send** and **handle**.
2. Run **handle** in the background, you should see something like:

```
student@COMPBase:~$ ./handle &
```

```
[1] 2329
```

where 2329 is an example of a process ID. In this case, the ID of the running handle process is 2329, but it will be different for your process.

3. Run **send**, enter the PID of **handle** and enter 1, 2, then 0.

5 Wait() and waitpid

1. Get **wait.c** from Chapter 5 (code part of the Notes). What should the output be? Try it.
2. Get **multiChildWait.c** from Chapter 5 (code part of the Notes). What should the output be? Try it.

3. Get **waitpid.c** from Chapter 5 of the Notes. What should the output be? Try it. Is the figure next to the code on p189 accurate?

6 Submit the tutorial

Create a tar file t8.tar and submit all the files that you created.

End of tutorial