COMP2401B—Tutorial 9
Cloning, Morphing, and Process management

Learning Objectives

After this tutorial, you will be able to:

- Clone a program (create child processes using fork())
- Morph programs into new programs – using exec()
- Receiving a return/exit code from a child process
- Do basic management of processes

Download the file t9.tar and extract the tutorial files.

As always, at the end of the tutorial submit your work!

1 Spawning new Processes

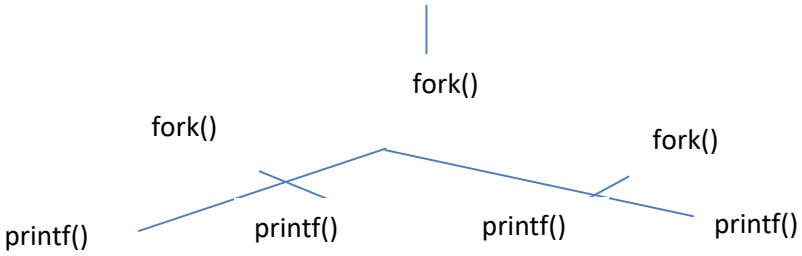
Purpose: Creating new processes

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 program 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 a process overview parent child relationship overview

   The following program clones itself once using the fork() function.

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

int main()
{
      int pid;
     pid = fork();
     printf("after fork \n");
}
```

3. Fork a process overview 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
}
```

4. 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() return the process id of the parent process. Recall that the 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 form 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 Waiting for a child process to terminate

Purpose: receiving the return/exit code of a child process

When a process forks a child (spawns a child) the system maintains a parent child relationship between the parent process and the child process.
The wait() function blocks the parent from continuing its execution until the child process completes.
Use "man wait" from the command line in order to read about the wait command.

1. Using the code from 1. above, add a wait() to the <u>parent code</u> as cpid = wait(&status); Print the process id that is returned from the wait() function.
2. The function has one parameter, int *status, which is used by the OS for:
   a. Informing the parent whether the child has exited normally – this is using WIFEXITED(status) macro. If the answer is true then the program terminated as expected (e.g., using the exit() function or a return() function).

The statement is checked by using
If (WIFEXITED(status) { … do something…}

b. Informing the parent program what is the return/exit code that the child has exited with, use the WEXITSTATUS(status) macro get the return code).
For example:
The statement below prints the return/exit code of the child program.
printf("Child has returned the value %d \n",WEXITSTATUS(status));
Use the WIFEXITED and WEXITSTATUS to check and print the return code of the child process.

3 Morphing a program

Purpose: transforming a program to another program
Morphing is an operation that allows a program to transforms itself to another program. Namely, the OS is loading a new instruction set for the new program and terminates the old program.

1. Create a program morphed.c with the following code. Compile it with the executable name *morphed*.

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

int main(int argc, char *argv[])
{
   int i;

   printf("I am a morphed program \n");
   for (i = 0; i < argc; i++) {
      printf("argv[%d] = %s \n", i, argv[i]);
   }
   Return(5*argc);
}
```

The morphing is carried out with the command execv. The command accepts two parameters: the name of the program (including path) and list of command line parameters to be transferred to the program. Note that the list of parameters uses the sentinel NULL to indicate the end of the parameter list. Also note that the first parameter in the command line parameter is the program name.

Use the following program to create a program that forks a child process and the child process is morphing itself. The file name is fork2.c

Note that the child code after the *execv*() function is not being executed. This is because the child process is starting a new program and is oblivious to what it was doing before. As a result no child code will be executed once the execv() function is successful.

```
 #include “stdio.h”
 #include “unistd.h”
int main(int argc, char *argv[])
{
      char *param[4];
      param[0]  = “morphed”;
      param[1]  =  “55”;
      param[2]  = “Second Command Parameter”;
      param[3]  = NULL;

      pid = fork()
      if (pid == 0) {
            // child process instructions
            int rc;
            rc = execv(“./morphed”,param);
            printf(“ This should not be printed \n”);          // this should not be executed because
                                                               // the child is executing a new program
      } else {
            //  parent process instructions
            printf(“Parent program \n”);
      }

}
```

To do:

a. Modify the number of parameters that are passed to the program as follows: 1. Increase the number of parameters by adding the parameter 5 as the last parameter to be passed (param[3]). Note that you still have to provide the NULL as the sentinel param[4].
b. Add a wait() command to receive the main program in and print the return code from the morphed program.

4 Managing Processes - Viewing and Controlling

Purpose: Managing 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 loop loop.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 inferred by looking that 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

5 Submit the tutorial

Create a tar file t9.tar and submit all the files that you created.

End of tutorial

6 Additional Exercises

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.