Operating Systems

Fall 2019

**Lab Exercise #4**

*A C program is like a fast dance on a newly waxed dance floor by people carrying razors."*

* Waldi Ravens

**Name: Bethany George / 6**

**Goals:** The intention of this lab is to better understand how an Linux processes work and programming with the fork(), wait(), exec(), and exit() C commands.

**Environment:** The GNU GCC C compiler on your Ubuntu virtual machine.

**Slack Channel:** All questions, problems and announcements for this lab should be directed to the course’s Lab 4 Blackboard discussion forum.

**Submission:** All files, including this document with the required screenshots, should be submitted via Blackboard and GitLab as indicated. ***Lab submissions not following this convention may not be graded.***

**Contents:** Submitted via GitLab with all source code and developed files checked into your repository. Create a new project (repository) named *cosc439-lastname-lab4* (where *lastname* is your last name) and add me ([ctessler@towson.edu](mailto:ctessler@towson.edu)) as a developer.

Do **not** add object (.o) or executable files such as collatz, or collatz.o to your repository.

Included in the project

* All source and makefiles
* This completed word document with screenshots

After pushing your contents to your project, include a link to the GitLab URL of the repository (e.g. <https://gitlab.com/s-mlr/AutowareAuto.git>) as the text submission on Blackboard for this lab.

**External References:** To understand how command line arguments are passed to C programming, use the following link for a *brief* description:

<https://en.wikibooks.org/wiki/A_Little_C_Primer/C_Command_Line_Arguments>

The getopt library is designed to handle command line arguments and may ease your implementation, start with the example linked below and examine the remaining documentation.

<https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html>

1. [**fork()**](https://www.youtube.com/watch?v=hiX9nbI56DI) **& wait().** (3 points) fork() is a system call that creates a new process under the Linux operating systems. It takes no arguments. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

 If fork() returns a negative value, the creation of a child process was unsuccessful.

 fork() returns a zero to the newly created child process.

 fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid\_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use the function getpid() to retrieve the process ID assigned to this process.

Therefore, after the system call to fork(), a simple test can tell which process is the child. Note that Linux will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

As an example, consider the following:

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

int main(){

printf(“Before Forking”);

fork();

printf(“After Forking”);

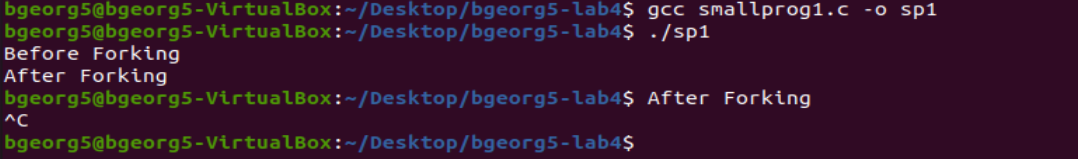
return 0;

}

If the call to fork() is executed successfully, Linux will:

* Make two identical copies of address spaces, one for the parent and the other for the child.
* Both processes will start their execution at the next statement following the fork() call.

**Implement this small program, compile it and run it and provide your screen shot here.**

****

As an explanation for your output, the printf() statement after fork() system call executed by parent as well as child process. Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

A program can differentiate between child and parent process and send each process to their own tasks, as shown in the following code snippet:

#include <stdio.h>

#include <sys/types.h>

void ChildProcess(); /\* child process prototype \*/

void ParentProcess(); /\* parent process prototype \*/

int main(){

pid\_t pid;

pid = fork();

if (pid == 0)

ChildProcess();

else

ParentProcess();

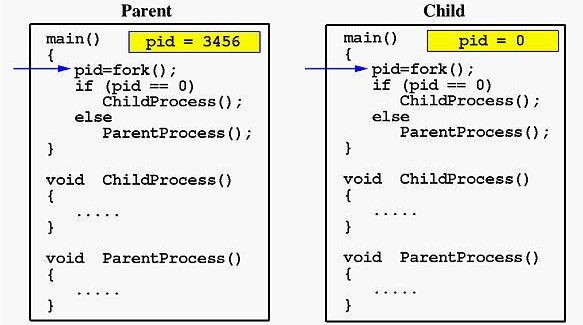
return 0;

}

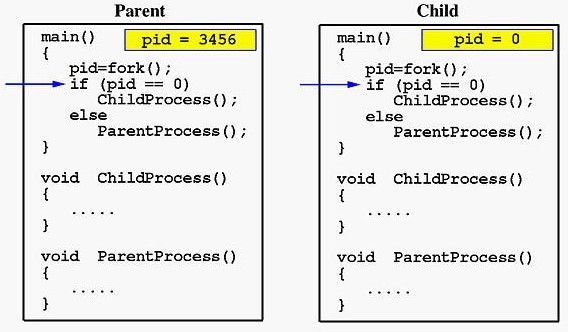
void ChildProcess() { }

void ParentProcess() { }

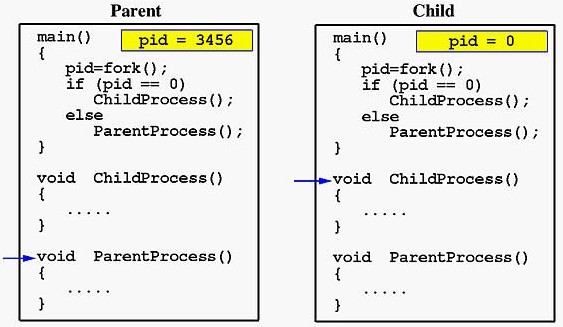
When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable pid. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the parent, since pid is non-zero, it calls function parentprocess(). On the other hand, the child has a zero pid and calls childprocess() as shown below:



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process.

If a parent needs to pause until its child process completes its execution, a call to wait() is needed. wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent *continues* its execution after wait system call instruction.

With this, **develop a C program, named collatz.c, that an integer, *n,* at the command line and prints out each number in the** [**collatz sequence**](https://en.wikipedia.org/wiki/Collatz_conjecture)**, starting with *n* on a new line. The collatz sequence, starting with *n* can be determined as follows: if *n* is even, divide it by 2 to get *n* / 2, if *n* is odd multiply it by 3 and add 1 to obtain 3*n* + 1; repeat the process until you reach 1. For example, the collatz sequence starting with 5 is 5, 16, 8, 4, 2, 1.**

To implement this, your [collatz](https://www.youtube.com/watch?v=RzQE2Q7lsT4).c program should first print out START on a new line in the parent process and then use the fork() to have a child process to calculate and print the collatz sequence and then the parent process print out END on a new line. To accomplish this, you will need to use the wait() call and you should do proper error checking in the parent process to ensure *n* is a positive number.

A correct output, with n=5 passed in at the command line to collatz, would print the following:

START

5

16

8

4

2

1

END

**Include a screenshot of your output here. Account for incorrect input**

****

1. **wait() & exec().** (3 points) The wait() system call blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.

The execution of wait() could have two possible situations:

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.
2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

# A *zombie process* occurs when a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.

# An *orphan process* is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under Linux, the latter kinds of processes are typically called daemon processes. The Linux *nohup* command is one means to accomplish this.

# The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program. The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways.

Within the exec family, there are functions that vary slightly in their capabilities:

1. execl() & execp()

execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL. For example:

# execl("/bin/ls", "ls", "-l", NULL);

execlp**():** It does same job except that it will use environment variable PATH to determine which executable to process. Thus, a fully qualified path name would not have to be used. The function execlp() can also take the fully qualified name as it also resolves explicitly. For example:

execlp("ls", "ls", "-l", NULL);

1. execv() & execvp()

execv(): It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string. For example:

char \*argv[] = {"ls", "-l", NULL};

execv("/bin/ls", argv);

execvp**():** It does same job expect that it will use environment variable PATH to determine which executable to process. Thus, a fully-qualified path name would not have to be used. For example:

execvp("ls", argv);

1. execve(): It executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form:

int execve(const char \*filename, char \*const argv[ ],

char \*const envp[ ]);

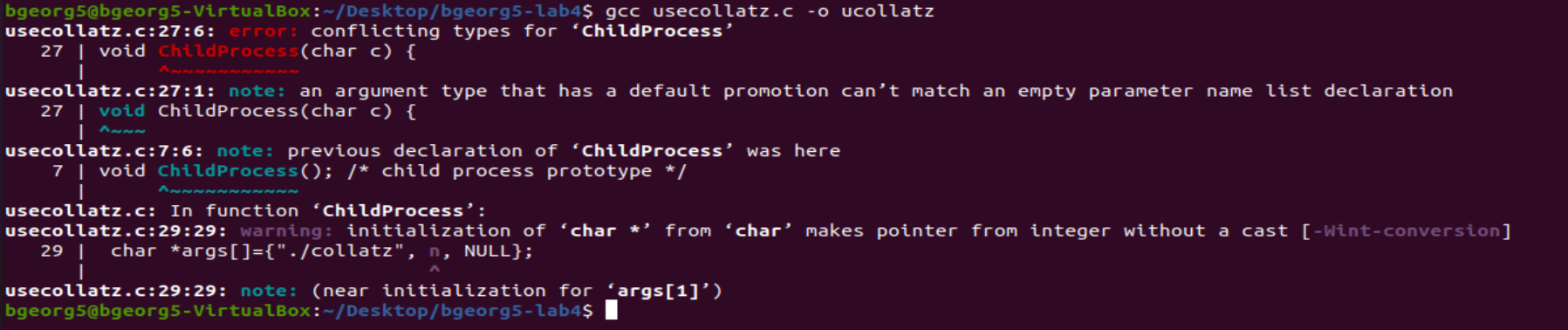
argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both argv and envp must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

int main(int argc, char \*argv[ ] , char \*envp[ ])]

execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

Develop a C program, named usecollatz.c, that asks the user to enter an integer between 1-100, prints to the console “You entered *n*.*”* (where *n* is the user provided integer) and then uses the fork() system call to create a child process that uses one of the exec() family calls to load the previously developed collatz program passing it the user-entered integer *n*.

**Include a screenshot of your output here.**

****