西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab 2 Report

Report Subject: OS Experiment - Lab 2

Student ID: 2021380101

Student Name: Rheina Trudy Williams

Data：2023/11/4

I. Introduction

In this laboratory session, we focus on practical aspects of Linux process creation and control. Beginning with the core concept of parent-child relationships and shared data, we examine the execution order between parent and child processes. The lab progresses to creating a defined number of child processes, offering hands-on experience with Linux's process management capabilities. Emphasis is placed on understanding process termination, as well as the concept of zombie processes and resource management in a Linux environment.

The final segment of the lab introduces participants to the creation of child processes that load new programs, providing a comprehensive overview of fundamental system calls in Linux process management. Through these exercises, participants will gain valuable proficiency in system-level programming.

II. Objectives

Learn to work with Linux system calls related to process creation and control.
Including:

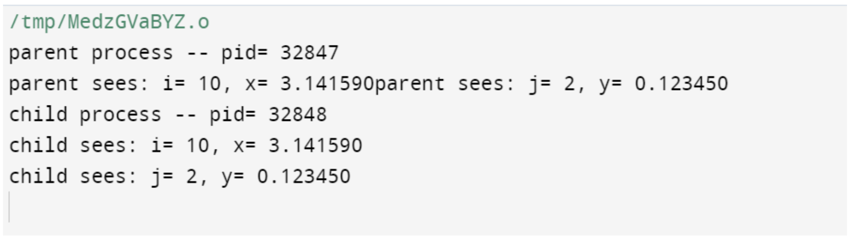
- process create and data sharing between parent and child
- the execution order of parent and child process
- Create the specified num of child processes
- Process termination
- Zombie process
- process create a child process and load a new program

III. Tools

Virtual box Ubuntu Linux
Programmiz online C compiler

IV. Experiment

Experiment 1: Process creation



```
/tmp/MedzGVaBYZ.o
parent process -- pid= 32847
parent sees: i= 10, x= 3.141590parent sees: j= 2, y= 0.123450
child process -- pid= 32848
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
```

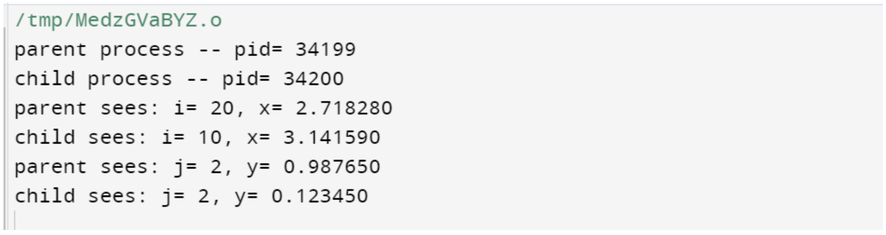
1) If you change the values of variable x ,y and i in parent process, do the variable in the child process will affected? Please give the reason.

Answer: No, the variables in the child process will not be affected because each process have its own independent set of variables. The variables x, y , and I are not shared between the parent and child process. Any changes made to variables in one process will not affect variables in other process.

Modified variables:

```
    if (pid > 0) {
        // parent code
        // Modify the variables after the fork
        i = 20; // Changed value of i
        x = 2.71828; // Changed value of x
        y = 0.98765; // Changed value of y
```

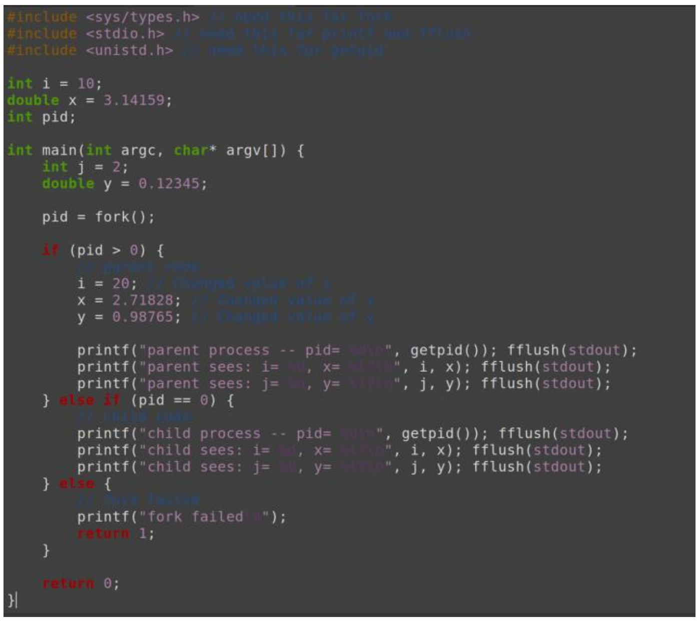
Output:



```
/tmp/MedzGVaBYZ.o
parent process -- pid= 34199
child process -- pid= 34200
parent sees: i= 20, x= 2.718280
child sees: i= 10, x= 3.141590
parent sees: j= 2, y= 0.987650
child sees: j= 2, y= 0.123450
```

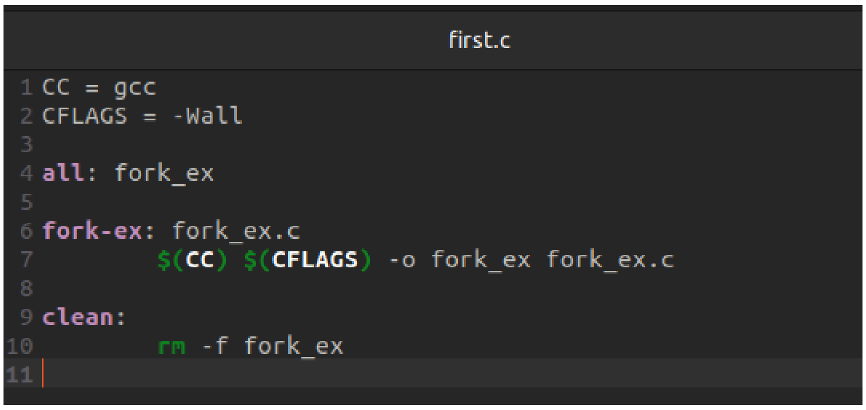
2) Please modify the fork_ex.c, and create a Makefile that builds all the programs you created. Test your expectation.

Modified program:



```c
#include <sys/types.h> // need this for fork
#include <stdio.h> // need this for printf and fflush
#include <unistd.h> // need this for getpid

int i = 10;
double x = 3.14159;
int pid;

int main(int argc, char* argv[]) {
    int j = 2;
    double y = 0.12345;

    pid = fork();

    if (pid > 0) {
        // parent code
        i = 20; // Changed value of i
        x = 2.71828; // Changed value of x
        y = 0.98765; // Changed value of y

        printf("parent process -- pid= %d\n", getpid()); fflush(stdout);
        printf("parent sees: i= %d, x= %lf\n", i, x); fflush(stdout);
        printf("parent sees: j= %d, y= %lf\n", j, y); fflush(stdout);
    } else if (pid == 0) {
        // child code
        printf("child process -- pid= %d\n", getpid()); fflush(stdout);
        printf("child sees: i= %d, x= %lf\n", i, x); fflush(stdout);
        printf("child sees: j= %d, y= %lf\n", j, y); fflush(stdout);
    } else {
        // fork failed
        printf("fork failed\n");
        return 1;
    }

    return 0;
}
```

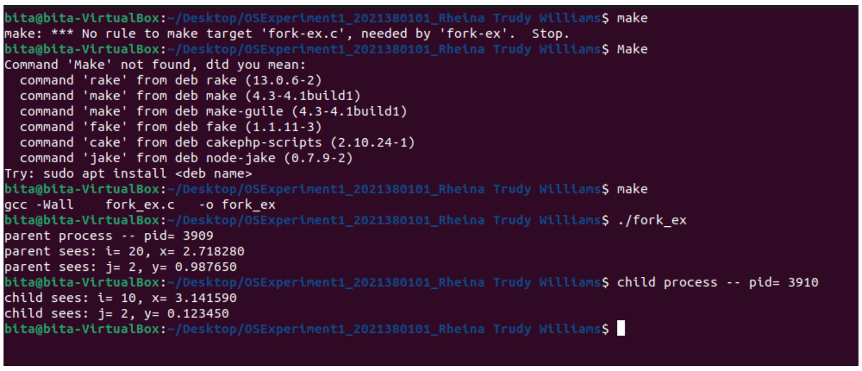
Makefile:



first.c

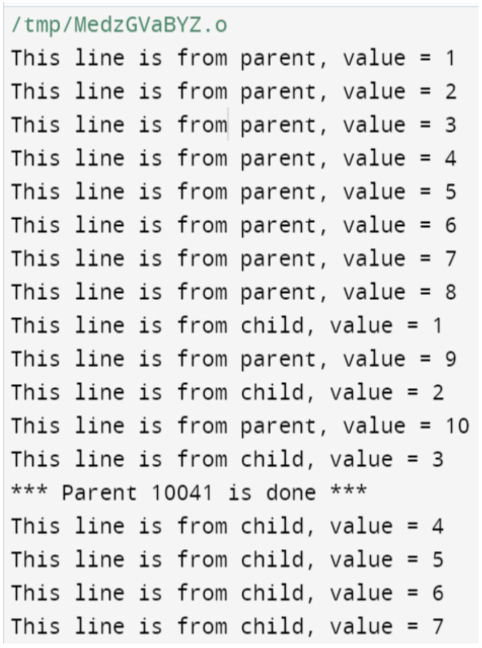
```
CC = gcc
CFLAGS = -Wall

all: fork_ex

fork-ex: fork_ex.c
	$(CC) $(CFLAGS) -o fork_ex fork_ex.c

clean:
	rm -f fork_ex

```

Tested:

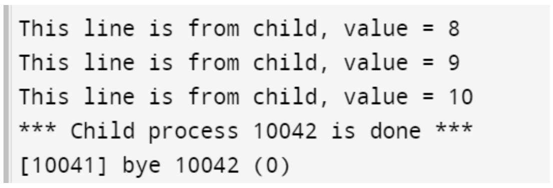


```
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ make
make: *** No rule to make target 'fork-ex.c', needed by 'fork-ex'.  Stop.
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ Make
Command 'Make' not found, did you mean:
  command 'rake' from deb rake (13.0.6-2)
  command 'make' from deb make (4.3-4.1build1)
  command 'make' from deb make-guile (4.3-4.1build1)
  command 'fake' from deb fake (1.1.11-3)
  command 'cake' from deb cakephp-scripts (2.10.24-1)
  command 'jake' from deb node-jake (0.7.9-2)
Try: sudo apt install <deb name>
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ make
gcc -Wall    fork_ex.c   -o fork_ex
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ ./fork_ex
parent process -- pid= 3909
parent sees: i= 20, x= 2.718280
parent sees: j= 2, y= 0.987650
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ child process -- pid= 3910
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ 
```

Experiment 2: The execution order of parent and child process
Let’s start slowly by investigating what a child process may be inheriting from its parent process. First, let’s get this code to compile!

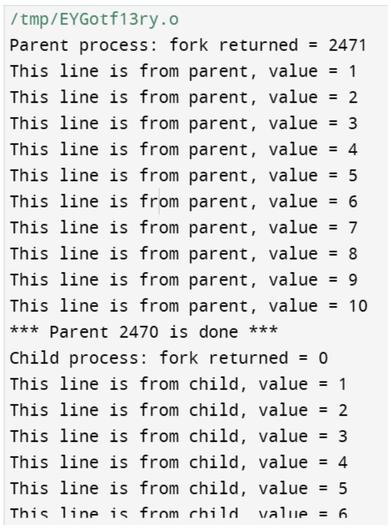


```
/tmp/MedzGVaBYZ.o
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
This line is from parent, value = 4
This line is from parent, value = 5
This line is from parent, value = 6
This line is from parent, value = 7
This line is from parent, value = 8
This line is from child, value = 1
This line is from parent, value = 9
This line is from child, value = 2
This line is from parent, value = 10
This line is from child, value = 3
*** Parent 10041 is done ***
This line is from child, value = 4
This line is from child, value = 5
This line is from child, value = 6
This line is from child, value = 7
```

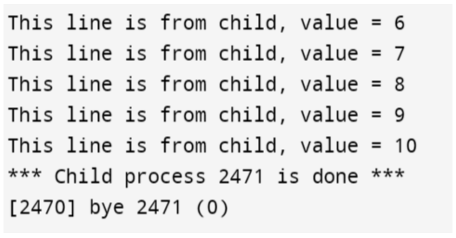


```
This line is from child, value = 8
This line is from child, value = 9
This line is from child, value = 10
*** Child process 10042 is done ***
[10041] bye 10042 (0)
```

After modification:



```
/tmp/EYGotf13ry.o
Parent process: fork returned = 2471
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
This line is from parent, value = 4
This line is from parent, value = 5
This line is from parent, value = 6
This line is from parent, value = 7
This line is from parent, value = 8
This line is from parent, value = 9
This line is from parent, value = 10
*** Parent 2470 is done ***
Child process: fork returned = 0
This line is from child, value = 1
This line is from child, value = 2
This line is from child, value = 3
This line is from child, value = 4
This line is from child, value = 5
This line is from child, value = 6
```



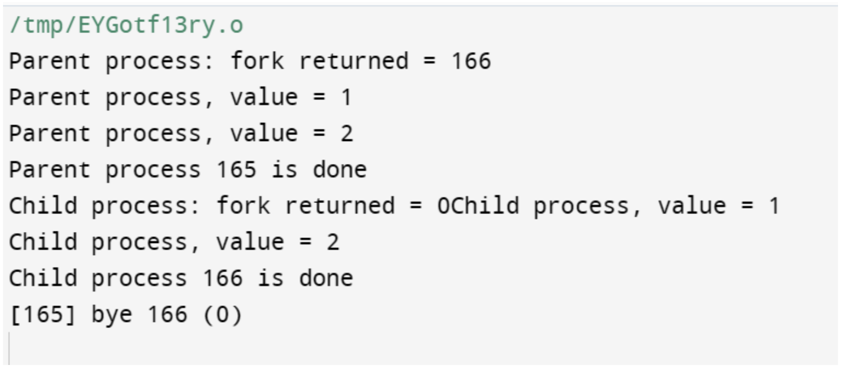
```
This line is from child, value = 6
This line is from child, value = 7
This line is from child, value = 8
This line is from child, value = 9
This line is from child, value = 10
*** Child process 2471 is done ***
[2470] bye 2471 (0)
```

1. The global variable num is declared before the call to fork() as shown in this program. After the call to fork(), when a new process is spawned, does there exist only one instance of num in the memory space of the parent process shared by the two processes or do there exist two instances: one in the memory space of the parent and one in the memory space of the child?
Answer:
After the fork() call, both parent and the child processes have their own separate memory spaces. The fork() system call creates a new process by duplicating the calling process. In the modified code, the 'num' variable is a global variable, and its value is the same for child and parent processes after fork() calling. Changes made to 'num' in one process will not affect the value of 'num' in the other process. Therefore, each copy has its own copy of the global variables.
The two instances of 'num' exists:

- In the memory space of the parent process.
- In the memory space of the child process.

And modification to 'num' in one process will not affect the value of 'num' in other process.

2. Can you infer the order of execution of these lines? Please try to decrease the num, If the value of num is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.
Answer: Yes the order of execution of these lines can be inferred based on the behavior of the parent and child process.
In this example, the num is decreased to 2 and the output is:



```
/tmp/EYGotf13ry.o
Parent process: fork returned = 166
Parent process, value = 1
Parent process, value = 2
Parent process 165 is done
Child process: fork returned = 0Child process, value = 1
Child process, value = 2
Child process 166 is done
[165] bye 166 (0)
```

We can see the two groups of lines

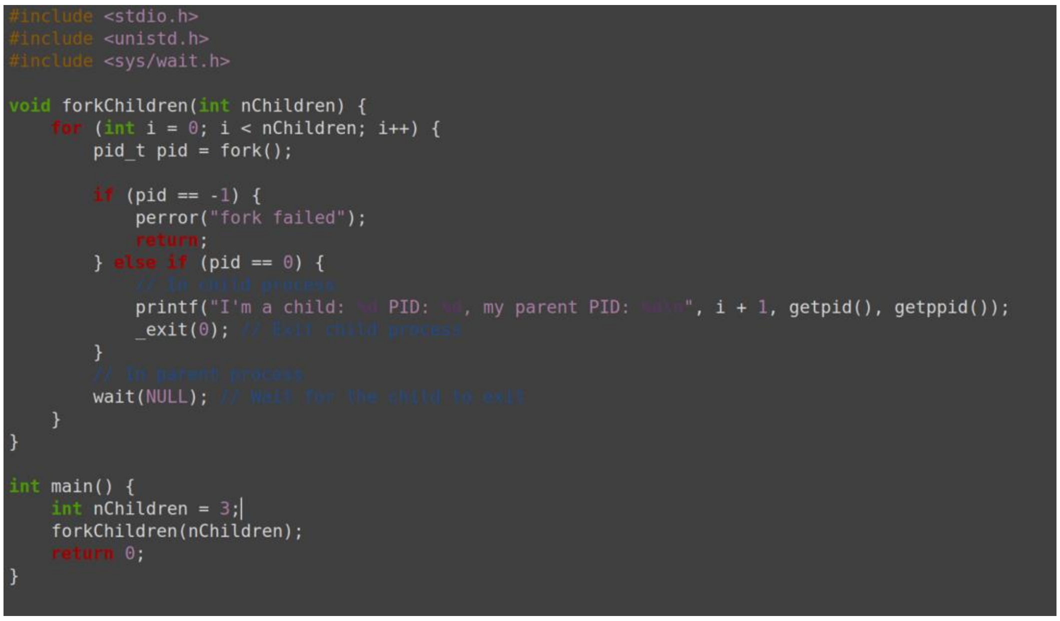
- Lines printed by the parent process:

Parent process: fork returned = 166
Parent process, value = 1
Parent process, value = 2
Parent process 165 is done

- Lines printed by the child process:

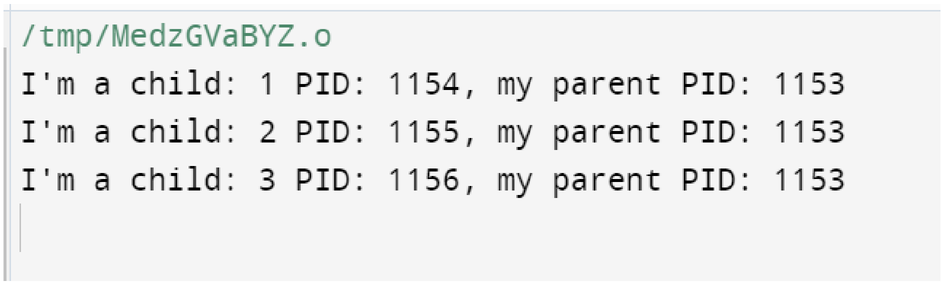
Child process: fork returned = 0Child process, value = 1
Child process, value = 2
Child process 166 is done
[165] bye 166 (0)

Experiment 3: Create the specified num of child processes
void forkChildren (int nChildren) in main function the forkChildren will be called, and each Children output their pid and parent pid.
childprocess.c:



```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

void forkChildren(int nChildren) {
    for (int i = 0; i < nChildren; i++) {
        pid_t pid = fork();

        if (pid == -1) {
            perror("fork failed");
            return;
        } else if (pid == 0) {
            // In child process
            printf("I'm a child: %d PID: %d, my parent PID: %d\n", i + 1, getpid(), getppid());
            _exit(0); // Exit child process
        }
        // In parent process
        wait(NULL); // Wait for the child to exit
    }
}

int main() {
    int nChildren = 3;
    forkChildren(nChildren);
    return 0;
}
```

Output:



```
/tmp/MedzGVaBYZ.o
I'm a child: 1 PID: 1154, my parent PID: 1153
I'm a child: 2 PID: 1155, my parent PID: 1153
I'm a child: 3 PID: 1156, my parent PID: 1153
```

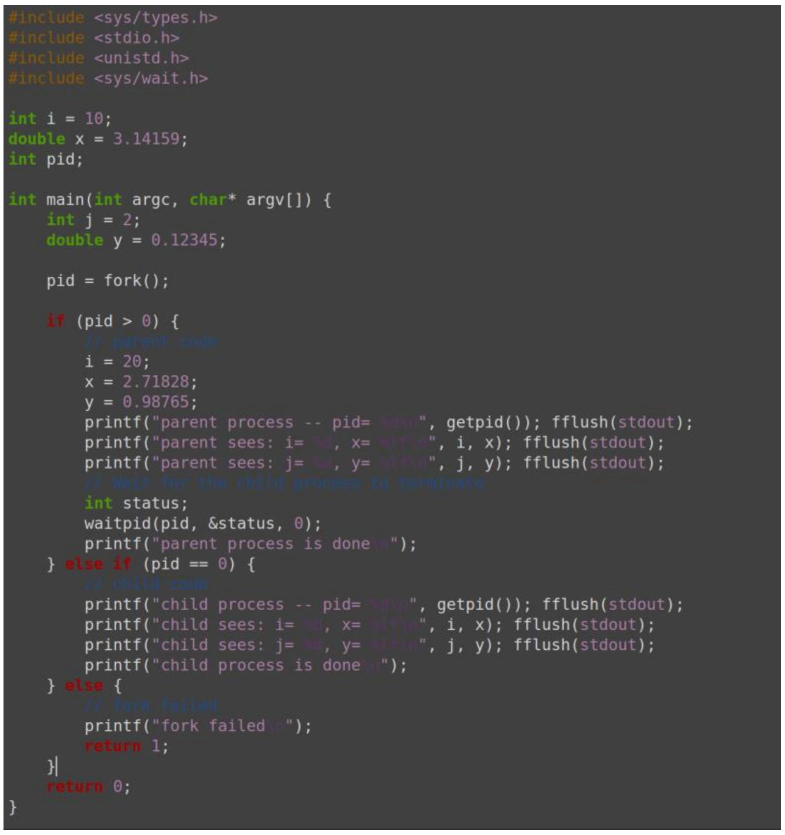
Question:
1. please observe the pid and can you tell the policy about pid allocation? Can you determine the parent process and child process from the pid?

a. Pid allocation policy:
- Pid allocation policy in Linux is incremental. When a new process is created, it is assigned a pid that is one higher than the highest pid currently in use.
- Pids are recycled. When the maximum Pid value is reached, the system starts reusing Pids from terminated processes.
- In the childprocess.c code, the pid alocation is managed by the operating system’s kernel.

b. Determining parent process and child process
- In the childprocess.c code, the parent-child relationship is determined based on the return value of fork(). Parent process gets the PID of the child and the child process gets 0 as the return value from fork()
- Based on the output:

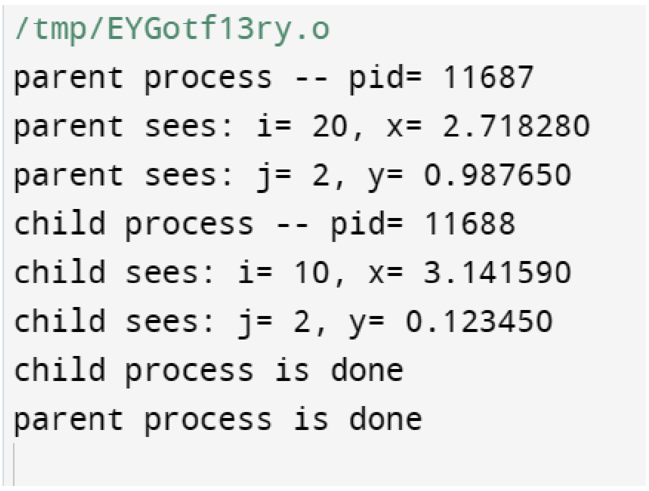
The parent process is 1153
The child process 1 is 1154
The child process 2 is 1155
The child process 3 is 1156

Experiment 4: Process termination
Copy fork_ex.c to file fork-wait.c and modify it so that you can guarantee that the parent process will always terminate after the child process has terminated. Your solution cannot rely on the termination condition of the for loops or on the use of sleep. The right way to handle this is using a syscall such as wait or waitpid – read their man pages before jumping into this task. One more thing: Modify the child process so that it makes calls to getpid(2) and getppid(2) and prints out the values returned by these calls.
Answer:
fork_wait.c:



```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int i = 10;
double x = 3.14159;
int pid;

int main(int argc, char* argv[]) {
    int j = 2;
    double y = 0.12345;

    pid = fork();

    if (pid > 0) {
        // parent code
        i = 20;
        x = 2.71828;
        y = 0.98765;
        printf("parent process -- pid= %d\n", getpid()); fflush(stdout);
        printf("parent sees: i= %d, x= %lf\n", i, x); fflush(stdout);
        printf("parent sees: j= %d, y= %lf\n", j, y); fflush(stdout);
        // wait for the child process to terminate
        int status;
        waitpid(pid, &status, 0);
        printf("parent process is done\n");
    } else if (pid == 0) {
        // child code
        printf("child process -- pid= %d\n", getpid()); fflush(stdout);
        printf("child sees: i= %d, x= %lf\n", i, x); fflush(stdout);
        printf("child sees: j= %d, y= %lf\n", j, y); fflush(stdout);
        printf("child process is done\n");
    } else {
        // fork failed
        printf("fork failed\n");
        return 1;
    }
    return 0;
}
```

Output:

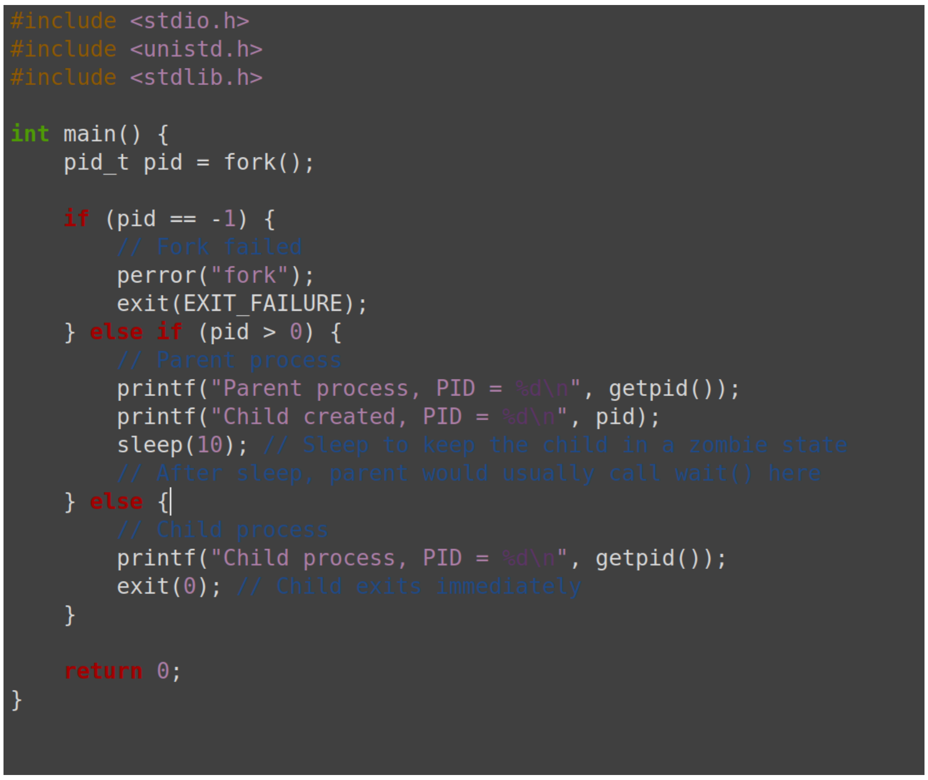


```
/tmp/EYGotf13ry.o
parent process -- pid= 11687
parent sees: i= 20, x= 2.718280
parent sees: j= 2, y= 0.987650
child process -- pid= 11688
child sees: i= 10, x= 3.141590
child sees: j= 2, y= 0.123450
child process is done
parent process is done
```

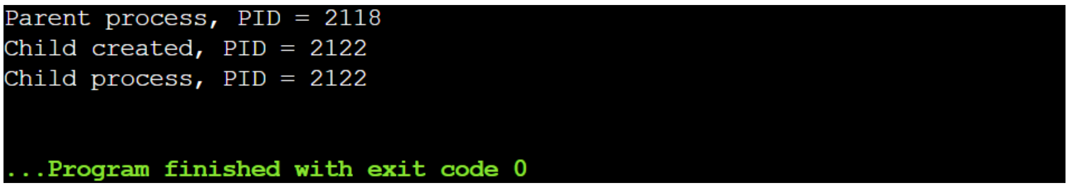
To guarantee the parent process always terminate after the child process has terminated, ‘waitpid’ is used in the parent process. The ‘waitpid’ system call suspends the execution of the calling process until one of its child processes terminates. This ensures the parent waits for the child process to complete before continuing its own execution.

Experiment 5: Zombie process

Zombie process occurs when a child process has completed execution but its parent has not yet read its exit status. Create a zombie process and use ps command to get the status of the process.



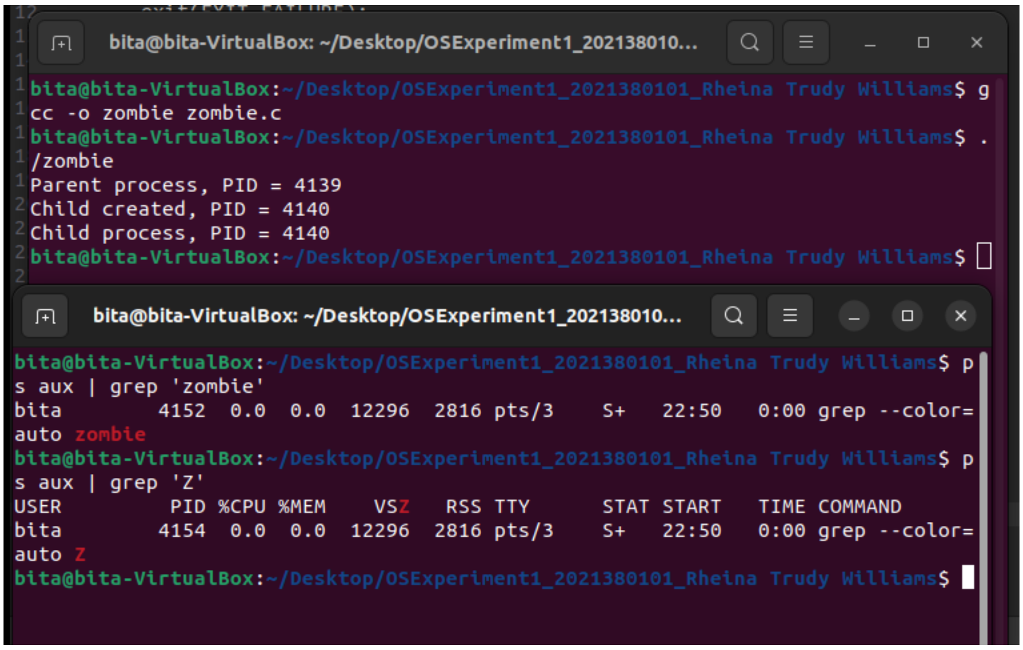
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        // Fork failed
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process, PID = %d\n", getpid());
        printf("Child created, PID = %d\n", pid);
        sleep(10); // Sleep to keep the child in a zombie state
        // After sleep, parent would usually call wait() here
    } else {
        // Child process
        printf("Child process, PID = %d\n", getpid());
        exit(0); // Child exits immediately
    }

    return 0;
}
```



```
Parent process, PID = 2118
Child created, PID = 2122
Child process, PID = 2122


...Program finished with exit code 0
```

1. Can you use kill command to kill the zombie process? If not, how can you reap the zombie process?

- Kill command cannot be used to terminate the zombie process because zombie processes are not running but entries in the process table, waiting to have their exit status collected by their parent. The kill command does not have any effect, the process itself is not executing.
- To reap the zombie process, it needs the parent process to read the child's exit status, using wait() or waitpid() system call.
- In zombie.c, wait() is added after sleep(10) to reap the zombie state.
- If the parent process ends before it can wait for the child process, the child process is adopted by the 'init' process (pid 1), calling wait() to reap any zombie processes it inherits.



```
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ g
cc -o zombie zombie.c
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ .
/zombie
Parent process, PID = 4139
Child created, PID = 4140
Child process, PID = 4140
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$
```

```
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ p
s aux | grep 'zombie'
bita        4152  0.0  0.0  12296  2816 pts/3    S+   22:50   0:00 grep --color=
auto zombie
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ p
s aux | grep 'Z'
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
bita        4154  0.0  0.0  12296  2816 pts/3    S+   22:50   0:00 grep --color=
auto Z
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$
```

Modified code to reap the zombie process:

```
//zombie process after modifying for reaping the process
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h> // Include for wait()

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        // Fork failed
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (pid > 0) {
        // Parent process
        printf("Parent process, PID = %d\n", getpid());
        printf("Child created, PID = %d\n", pid);
        sleep(10); // Sleep to keep the child in a zombie state temporarily

        int status;
        pid_t waitedPid;
        while ((waitedPid = wait(&status)) > 0) { // Reap the zombie process
            if (WIFEXITED(status)) { // Check if the child process exited normally
                printf("Child %d terminated with exit status %d\n", waitedPid, WEXITSTATUS(status));
            }
        }
    } else {
        // Child process
        printf("Child process, PID = %d\n", getpid());
        exit(0); // Child exits immediately
    }

    return 0;
}
```

Experiment 6: Create a child process and load a new program

Description: a shell interface accepting user commands and executes each command in a separate process. The program allows users to enter commands, executes them in separate processes and supports running commands in the background. In this terminal output, I use ls -l to list the files in the current directory.

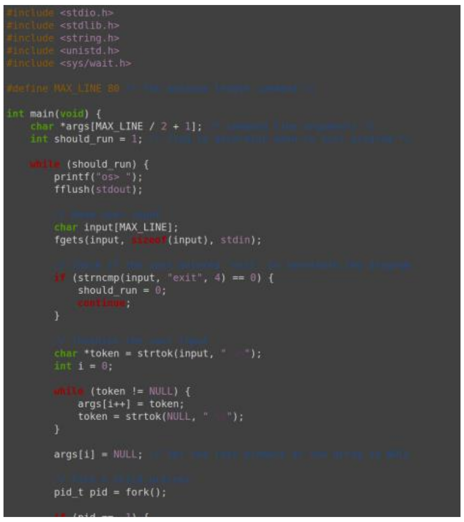
Other commands are:

date = printing the current date and time

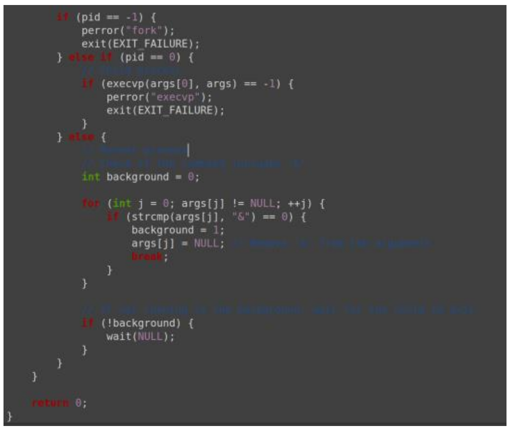
echo "hi" = printing the specified message

exit = terminating the shell program

Code:

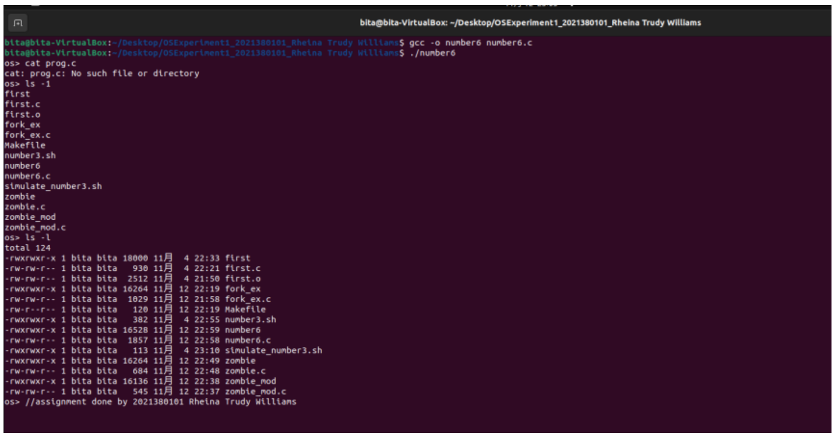


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX_LINE 80 [illegible]

int main(void) {
    char *args[MAX_LINE / 2 + 1]; [illegible]
    int should_run = 1; [illegible]

    while (should_run) {
        printf("os> ");
        fflush(stdout);

        [illegible]
        char input[MAX_LINE];
        fgets(input, sizeof(input), stdin);

        [illegible]
        if (strncmp(input, "exit", 4) == 0) {
            should_run = 0;
            continue;
        }

        [illegible]
        char *token = strtok(input, " \n");
        int i = 0;

        while (token != NULL) {
            args[i++] = token;
            token = strtok(NULL, " \n");
        }

        args[i] = NULL; [illegible]

        [illegible]
        pid_t pid = fork();
```

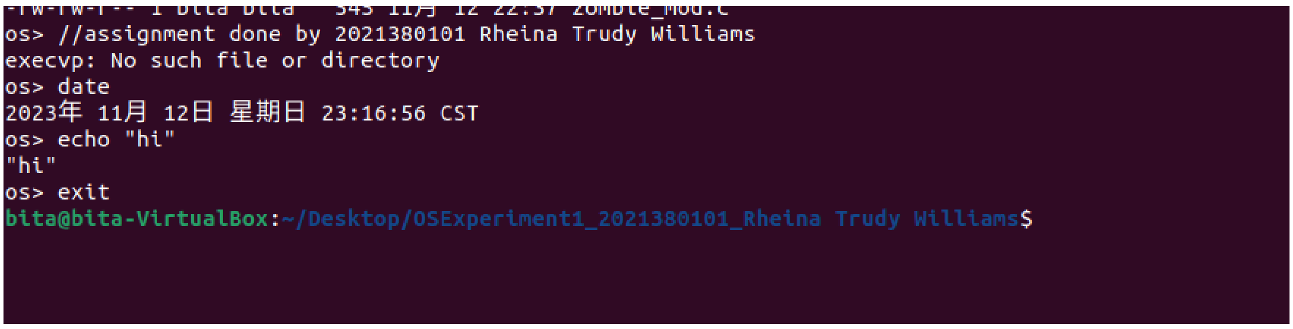


```c
        if (pid == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        } else if (pid == 0) {
            [illegible]
            if (execvp(args[0], args) == -1) {
                perror("execvp");
                exit(EXIT_FAILURE);
            }
        } else {
            [illegible]
            [illegible]
            int background = 0;

            for (int j = 0; args[j] != NULL; ++j) {
                if (strcmp(args[j], "&") == 0) {
                    background = 1;
                    args[j] = NULL; [illegible]
                    break;
                }
            }

            [illegible]
            if (!background) {
                wait(NULL);
            }
        }
    }

    return 0;
}
```

Output terminal:



```
bita@bita-VirtualBox: ~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ gcc -o number6 number6.c
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$ ./number6
os> cat prog.c
cat: prog.c: No such file or directory
os> ls -1
first
first.c
first.o
fork_ex
fork_ex.c
Makefile
number3.sh
number6
number6.c
simulate_number3.sh
zombie
zombie.c
zombie_mod
zombie_mod.c
os> ls -l
total 124
-rwxrwxr-x 1 bita bita 18000 11月  4 22:33 first
-rw-rw-r-- 1 bita bita   930 11月  4 22:21 first.c
-rw-rw-r-- 1 bita bita  2512 11月  4 21:50 first.o
-rwxrwxr-x 1 bita bita 16264 11月 12 22:19 fork_ex
-rw-rw-r-- 1 bita bita  1029 11月 12 21:58 fork_ex.c
-rw-r--r-- 1 bita bita   120 11月 12 22:19 Makefile
-rwxrwxr-x 1 bita bita   382 11月  4 22:55 number3.sh
-rwxrwxr-x 1 bita bita 16528 11月 12 22:59 number6
-rw-rw-r-- 1 bita bita  1857 11月 12 22:58 number6.c
-rwxrwxr-x 1 bita bita   113 11月  4 23:10 simulate_number3.sh
-rwxrwxr-x 1 bita bita 16264 11月 12 22:49 zombie
-rw-rw-r-- 1 bita bita   684 11月 12 22:48 zombie.c
-rwxrwxr-x 1 bita bita 16136 11月 12 22:38 zombie_mod
-rw-rw-r-- 1 bita bita   545 11月 12 22:37 zombie_mod.c
os> //assignment done by 2021380101 Rheina Trudy Williams
```



```
-rw-rw-r-- 1 bita bita   545 11月 12 22:37 zombie_mod.c
os> //assignment done by 2021380101 Rheina Trudy Williams
execvp: No such file or directory
os> date
2023年 11月 12日 星期日 23:16:56 CST
os> echo "hi"
"hi"
os> exit
bita@bita-VirtualBox:~/Desktop/OSExperiment1_2021380101_Rheina Trudy Williams$
```

V. Epilogue

After several trials and errors, I have written the codes for testing processes of the operating system. Some of the codes cannot be run in Linux Mini because of library restrictions. Therefore, I borrow my friend’s laptop and compile the codes in C online compiler. After several analyzing, the experiment is finished successfully.