The Study of Processes in UNIX
Lab. #2

Arif A. Balik

Undergraduate Student
Sytstems Programming
Department of Computer Science
Arel University
Büyükçekmece, İstanbul 34537
Email: arifbalik@outlook.com

March 14, 2019

Abstract

This report include analysis of some system calls in UNIX environment based on a program which provided in System Programming course. First section explains some system calls, second section answers some questions asked in the experiment paper, and third section suggests and implements some improvements and optimisations on the code.

This report is powered by LaTeX

Explanation of Some System and Library Functions

Difference between system and library functions

A system call is provided by the kernel and directly handled there, and a library functions are functions within programs, usually they are in the same level as shells.

write()

write is a system call used to output data to given channel (terminal, file etc.)

It has the prototype;

```
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

where *fildes* is the channel that is where to write, *buf* is which to write and *nbytes* is how many bytes to write.

fork()

The system call *fork* creates an exact copy of current program. Takes no parameters and returns *0* for *child* process and child's id for *parent* process.

getpid()

The system call *getpid* returns the process id of currently running process.

getppid()

The system call *getppid* returns the process id of currently running process's parent process.

execl()

The system call *excl* replaces the current program with given program. It has the form;

```
int execl(const char *path, const char *arg, ...);
```

where *path* is the path for new executable file and *arg* is arguments to pass to the new process. Function only returns *-1* if there is an error. By convention the first argument to pass into new program should be the name of the program and list of arguments should end with *NULL*.

sleep()

In the labsheet, instructor specified the function *sleep* as a library function, but in most of places it reffered as a system function.[i]

But then again looking at the **unistd.h** where sleep defined, it can bee seen that it uses a well documented system call **nanosleep**.

```
unsigned int sleep(unsigned int seconds)
{
        struct timespec ts;

        ts.tv_sec = seconds;
        ts.tv_nsec = 0;
        if(nanosleep(&ts, &ts) == 0)
                return 0;
        return ts.tv_sec;
}
```

It delays the system by given amount in seconds and returns *0* if the amount of time elapsed otherwise (in case of interruption) returns the amount of time left.

[i] http://www.cs.miami.edu/home/geoff/Courses/CSC521-04F/Content/UNIXProgramming/UNIXSystemCalls.shtml

Theory of Operation

Part 1

mainprog.c runs first and immediately creates three childs and childs overlays themselves with the program **child.c** with the arguments *1*, *2* and *3* respectively. Then parent process prints the ID of childs and wait them until they exit.

In the **child.c** each child prints their pid values along with a *char*, that is 0 - 256 number created from its own pid value. Then it sleeps random amount of time (0 - 5 sec). Finally if the value that have been passed trough child via *argv* is checked whether it is an even or odd number, if it is even then child kills itself with the signal value *9*, if not it exits with the char value mentioned above.

Part 2

In the **procident.c** program first creates a child process than terminates itself after 3 seconds of sleep (thus child becomes orphan). In the child process, after 10 seconds of sleep, child overlays itself with **simple.c**. Also as mentioned in the previous lab report, the variables can not effect each other after fork unless intended.

In the **simple.c** child reports its parent (**init()**) and exits.

In the **procmemory.c** program creates a bunch of global and local variables and shows their virtual addresses along with the starting addresses of the program, that is *etext*, *edata* and *end.*

INFORMAL NOTE : Honestly I did not understand why the variables have the same addresses in both child and parent processes. I tried to change the value of *i* in the child process and read it both from child and parent, and as expected they yield different results, but then again their addresses are same. My expertise is on mostly embedded systems and bare-metal C, so I can't understand the concept of Virtual Memory how matter I try. I will read more documents thou.

etext	0x100000DF4
edata	0x100001070
end	0x100003000
main	0xD4EDB90
showit	0xD4EDD30
cptr	0xD4EE068
buffer1	0xD4EE070
i	0x52712A4C
buffer2	0x527129D8

Q&A

Q1. *What is the purpose of this laboratory work?*

The purpose is to get an understanding of some UNIX system calls, dynamics between child and parent processes during switching between programs and understanding the Virtual Memory.

Q2. *How mony child processes created by the parent process in* ***mainprog.c****?*

Three child processes created.

Q3. *In* ***mainprog.c****, a few statements are executed by each created child process. Show these statements.*

```
sprintf(value, ”%d”, i);
execl(”child”, ”child”, value, 0);
```

Q4. *In* ***mainprog.c****, to what program is child process switched? Show and explain the corresponding statement. Does a child process return to the* ***mainprog*** *after finishing another program (explain, why yes/no)?*

Child processes switched to **child.c**.
Child does not returns to **mainprog.c** because when **execl** called, it replaces the current program with **child.c**, there is no way that child can return.[ii]
Also, in the program either process kills itself or exits.

Q5. *Which statements are NOT executed by the parent process in* ***mainprog.c****?*

Statements shown in **Q3.**

Q6. *What is the purpose of system call* ***wait()*** *in* ***mainprog.c****? What results are extracted by the parent process from this system call?*

The function **wait** puts parent process into sleep until child processes exits. It returns the ID of child process who terminated and fills the variable which given as a parameter with the exit status given by the child when terminating.

[ii] Unless **mainprog.c** called again in **child.c** using **execl**

Q7. *Is it possible for a child process to continue its work after the parent process terminates? What will the parent be for such a child? Prove your answer based on the results of the Part 2 of this lab work.*

Yes. When parent terminates before child process, then that process called *orphan*, and **init** function takes over the role of parent which has a pid of 1.

As we can see from the output of the program **procident.c**, parent terminates before the child, and child still runs and outputs its parent process id which is 1.

```
Child: my ID = 5219, i = 2
Child: my parent ID = 5218
Parent: my ID = 5218, i = 0
Parent terminating...
Arif-MacBook-Pro:lab2 celimless$
 Child after sleping: my ID = 5219

 Child after sleping: my parent ID = 1
NEW PROGRAM simple IS STARTED BY TH CHILD PROCESS
Child: my ID = 5219
Child: my parent ID = 1
Child: terminating...
```

Q8. *What is the purpose of directive* ***#define*** *in the program* ***procmemory.c****?*

#define is a preprocessor, it has no use as a compiled program but it gives developer the ability to create more modular and readable code. In the **procmemory.c** compiler simply pastes the contents wherever it sees the definition **SHW_ADR**. Also the definition behaves as a function but has no effect on program as such.

For example, if developer calls this definition *SHW_ADR(”x”,x)*, compiler changes that line as follows;

```
printf(”ID %s \t is at virtual address: %8X\n”, ”x”, &x)
```

Q9. *What is the meaning of the variables* ***etext, edata*** *and* ***end*** *in* ***procmemory.c****? Why are these variables declared with the word* ***extern****?*

The variables **etext**, **edata** and **end** represents segments of the program, such as variable environment, text segment etc. They are provided by linker and dangerous to use as stated by Linux man page. For instance macOS does not allow to access does variables directly, it rather provides some functions which returns the values of those variables.

extern keyword gives variable the ability to extend itself throughout the source files. But it only declares the variable, not defines it. In order to use the variable, it has to be defined somewhere in the source program, or in a library.

Q10. *Suppose that a variable i was declared and assigned some value in the parent process before the creation of a child process. Will this variable be accessible to the child process? Will the parent process see the change made in this variable by the child process?*

Every variable is copied and transferred to new locations with their last values in the process of fork. So the child could access to the variable but it would be isolated from parent's variable even thou they are in the same source file, have the same name, and same definition etc.

Q11. *Is* ***malloc()*** *a system call or a library function? Can you guess it looking at your program?*

Knowing whether is a function a system call or not is not possible by looking at the source file but few comments can be made.

malloc used to allocate memory, so it has to deal with low level sources but as in **printf - write** relationship, **malloc** may use some other system calls to accomplish the task of allocation.

According to the *Linux Programmer's Manual* **malloc** is not a system call.[iii]

Q12. *What is the meaning of two parameters of the function* ***main()*** *in the program* ***child.c****? What is* ***argv[1]*** *in this program?*

The first parameter **argc** gives the number of arguments given as an input when executable file started, and second one gives the values of those parameters.

argv[1] is the number of the child which is the variable i in the **mainprog.c**

Q13. *What is the purpose of system calls* ***getpid*** *and* ***getppid****?*

getpid returns the value of current process and **getppid** returns the value of parent process of the current process. It is usually used to create unique file names, but can be used to make child and parent processes to do different things in the same source file.

Q14. *Is it possible for a process to use more than one program?*

No. A process can run one program at a time. But it can use multithreading which is not same as running multiple programs. It can also trigger other programs to run them, but then again it does not mean the process runs that program.

[iii] http://man7.org/linux/man-pages/man2/syscalls.2.html

Q15. *Is it possible for two or more processes to use the same program?*

Yes but by saying *same* it should be noted that two processes will use same source code and will run two identical programs at the same time separately.

Q16. *What is the purpose of the system calls of the* ***exec*** *family? Does this system call return any result (in case it succeeds or fails)?*

exec system calls used to alter the current program and replace it and naturally they have no return values unless there is an error, which will return -1.

Optimization

Programs are well written, not much to be done.

In the **mainprog.c** there is an unnecessary use of if and while condition.

Listing 1: Original

```
while((w = wait(&status)) &&
                        w != -1){
          if(w != -1) ...
}
```

Listing 2: Optimised

```
while(!(w = wait(&status))){
    ...
}
```

Because return of the function **wait** will be always positive when child successfully terminates there is no need to use *!= -1.* Also the condition of *w != -1* is checked three times, two inside the parenthesis of while loop and one inside the loop.