

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

PROGRAM: SOFTWARE ENGINEERING

OPERATING SYSTEMS LAB

Lab_Task 07

SUBMITTED BY:

Name: Ahmed Ali

Roll No: 22P-9318

Section: BS(SE)-5B

INSTRUCTOR NAME: Sir Saad

A DEPARTMENT OF COMPUTER SCIENCE

Output of execv:

```
fast@HALAB-11:~/Desktop/Lab_7$ gcc main.c -o main
fast@HALAB-11:~/Desktop/Lab_7$ ./main
Parent Process
Child Process
```

Question1 What is the 1st argument to the execv() call? What is it's contents?

ANSWER:

The first argument to the execv() call is arg[0], which is "/usr/bin/ls"

This argument specifies the path to the executable that the child process will run, which in this case is the ls command (used to list directory contents).

Question2 What is the 2nd argument to it? What is it's contents?

ANSWER:

The second argument to execv() is arg, which is defined as:

```
char *arg[] = {"/usr/bin/ls", 0};
```

arg is an character pointers array

The contents of this array are:

- arg[0] points to the string "/usr/bin/ls".
- arg[1] is 0 (or NULL), which signifies the end of the argument list for the `execv()` call.

Question3 What is arg?

ANSWER:

arg is an array of character pointers that holds the command line arguments for the `execv()` call

In this case, it contains:

- The first element is the path to the executable ("/usr/bin/ls").
- The second element is NULL(0), indicating the end of the arguments for the command.

Question4 Look at the code of the child process (p==0). How many times does the statement Child Process appear? Why?

The line ``printf("Child Process\n");`` is found twice in the child process section. However, the second time it appears after the ``execv()`` call would not run if ``execv()`` works correctly.

Here's what happens:

1. The first ``printf("Child Process\n");`` runs before the ``execv()`` call, so it prints "Child Process".
2. The ``execv()`` function then takes over the child process and runs the ``ls`` command. If ``execv()`` works, the original process is replaced, and the second ``printf()`` won't run.
3. So, if ``execv()`` is successful, "Child Process" will be printed only once.

If ``execv()`` fails (like if the path is wrong), then the second ``printf()`` will run, and "Child Process" will be printed twice in that case.

Exercise

Write a C program that can display a count from 10 to 0 (reverse order) using a for or a while loop. Each number should be displayed after delay of 1 second.

CODE Written in VIM:

```
#include<unistd.h>
#include<stdio.h>
int main()
{
    for(int i = 10; i >= 0; i--)
    {
        printf("%d\n",i);
        sleep(1); //delay for 1 second
    }
    return 0;
}
```

Each output is printed after 1 second:

```
Fast@HALAB-11:~/Desktop/Lab_7$ gcc reverse.c -o reverse
Fast@HALAB-11:~/Desktop/Lab_7$ ./reverse
10
9
8
7
6
5
4
3
2
1
0
Fast@HALAB-11:~/Desktop/Lab_7$
```

EXIT() CALL:

The `exit()` system call causes a normal program to terminate and return status to the parent process. Study the behaviour of the following program. You will see that there may be a number of exit points from a program. Can you identify which `exit()` call is being used each time a program exits??

```
#include <stdlib.h> // Include for exit() function
void anotherExit(); // Function Prototype

int main()
{
    int num;
    printf("Enter a Number: ");
    scanf("%i", &num);

    if(num > 25)
    {
        printf("exit 1\n");
        exit(1); // Exit point 1
    }
    else
    {
        anotherExit(); // Call another exit function
    }

    return 0;
}

void anotherExit()
{
    printf("Exit 2\n");
    exit(2); //Exit point 2
}
```

Answering the Questions

1. First Exit Point:

- The first exit point is when `exit(1)` is called in the `main()` function if `num > 25`. This indicates that the program terminates with status 1.

2. Second Exit Point:

- The second exit point is in the `anotherExit()` function, where `exit(2)` is called. This indicates that the program terminates with status 2 when the input number is 25 or less.

In Short:

- `exit(1)`: Used when the input number is greater than 25.
- `exit(2)`: Used in the `anotherExit()` function when the input number is 25 or less.

Atexit() Call:

Q1 What is the difference between `exit()` and `atexit()`? What do they do? (Check `man atexit` and `man 3 exit`).

ANSWER:

- `exit()`: This function stops the program. The number you give it (like 0) tells whether the program finished successfully (0 means success, anything else usually means an error).
- `atexit()`: This function lets you set up other functions to run when the program ends normally. You can add multiple functions, and they will run in reverse order from how you added them.

```
#include<stdlib.h>

void f1(void);
void f2(void);
void f3(void);

int main(void)
{
    atexit(f1);
    atexit(f2);
    atexit(f3);
    printf("Getting ready to exit\n");
    exit(0);
}

void f1(void)
{
    printf("In f1\n");
}

void f2(void)
{
    printf("In f2\n");
}

void f3(void)
{
    printf("In f3\n");
}
```

OUTPUT:

```
fast@HALAB-11:~/Desktop/Lab_7$ vim atexit_call.c
fast@HALAB-11:~/Desktop/Lab_7$ gcc atexit_call.c -o atexit.out
fast@HALAB-11:~/Desktop/Lab_7$ ./atexit.out
Getting ready to exit
In f3
In f2
In f1
fast@HALAB-11:~/Desktop/Lab_7$
```

Q2 What does the 0 provided in the `exit()` call mean? What will happen if we change it to 1? (Check manual page for `exit`)

ANSWER:

The 0 in `exit(0)` means the program is ending successfully. If you change it to 1, it means the program ended because of an error. This number can be used by the program that started it to check how it finished.

```
#include<stdlib.h>

void f1(void);
void f2(void);
void f3(void);

int main(void)
{
    atexit(f1);
    atexit(f2);
    atexit(f3);
    printf("Getting ready to exit\n");
    exit(1);
}

void f1(void)
{
    printf("In f1\n");
}

void f2(void)
{
    printf("In f2\n");
}

void f3(void)
{
    printf("In f3\n");
}
```

OUTPUT:

```
fast@HALAB-11:~/Desktop/Lab_7$ vim atexit_call.c
fast@HALAB-11:~/Desktop/Lab_7$ gcc atexit_call.c -o atexit.out
fast@HALAB-11:~/Desktop/Lab_7$ ./atexit.out
Getting ready to exit
In f3
In f2
In f1
fast@HALAB-11:~/Desktop/Lab_7$
```

Q3 If we add an `exit` call to function `f1`, `f2`, or `f3`. What will happen to execution of our program?

ANSWER:

```
#include <stdlib.h>

void f1(void);
void f2(void);
void f3(void);

int main(void) {
    atexit(f1);
    atexit(f2);
    atexit(f3);
    printf("Getting ready to exit\n");
    exit(0);
}

void f1(void) {
    printf("In f1\n");
}

void f2(void) {
    printf("In f2\n");
}

void f3(void)
{
    exit(0);
    printf("In f3\n");
}

~
$
```

OUTPUT:

```
fast@HALAB-11:~/Desktop/Lab_7$ vim atexit_call_with_exit2.c
fast@HALAB-11:~/Desktop/Lab_7$ gcc atexit_call_with_exit2.c -o atexit2.out
fast@HALAB-11:~/Desktop/Lab_7$ ./atexit2.out
Getting ready to exit
In f2
In f1
fast@HALAB-11:~/Desktop/Lab_7$
```

If you add an `exit()` call inside `f1`, `f2`, or `f3`, the program will stop right away when that function runs. This means it won't finish running any other functions that were supposed to run afterward.

Q4 Why do you think we are getting reverse order of execution of `atexit` calls?

ANSWER:

The program runs the functions you registered with `atexit()` in reverse order. This is done to make sure that things created or opened are cleaned up in the opposite order, which helps avoid mistakes and keeps everything tidy.

Abort Call:

```
In f1
fast@HALAB-11:~/Desktop/Lab_7$ vim abort.c
fast@HALAB-11:~/Desktop/Lab_7$ gcc abort.c -o abort.out
fast@HALAB-11:~/Desktop/Lab_7$ ./abort
bash: ./abort: No such file or directory
fast@HALAB-11:~/Desktop/Lab_7$ ./abort.out
Aborted (core dumped)
fast@HALAB-11:~/Desktop/Lab_7$
```

Q1: Check the man pages for `abort`. How does the `abort` call terminate the program? What is the name of the particular signal?

Answer:

The abort function stops the program by sending a signal called SIGABRT. This signal means something went wrong, and when abort is called, the program ends right away. If there's no special handler for this signal, the program will finish and may create a core dump to help with debugging.

Q2: Execute your program. What is the output of our program?**Answer:**

When you run the program, it will call abort() and stop immediately

```
fast@HALAB-11:~/Desktop/Lab_7$ vim abort.c
fast@HALAB-11:~/Desktop/Lab_7$ gcc abort.c -o abort.out
fast@HALAB-11:~/Desktop/Lab_7$ ./abort
bash: ./abort: No such file or directory
fast@HALAB-11:~/Desktop/Lab_7$ ./abort.out
Aborted (core dumped)
fast@HALAB-11:~/Desktop/Lab_7$
```

Q3: Include the abort call in function f3 in our code provided for Atexit() call. How does our program terminate using this?**Answer:**

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

void f3()
{
    abort();
}

int main()
{
    atexit(f3); // register f3 to run when the program exits
    exit(0); // Exit
}
```

Output:

```
fast@HALAB-11:~/Desktop/Lab_7$ vim abortQ3.c
fast@HALAB-11:~/Desktop/Lab_7$ gcc abortQ3.c -o abortQ3.out
fast@HALAB-11:~/Desktop/Lab_7$ ./a
abort.out  abortQ3.out  atexit1.out  atexit2.out  atexit.out
fast@HALAB-11:~/Desktop/Lab_7$ ./abortQ3
bash: ./abortQ3: No such file or directory
fast@HALAB-11:~/Desktop/Lab_7$ ./abortQ3.out
Aborted (core dumped)
fast@HALAB-11:~/Desktop/Lab_7$
```

when I call exit(0) in main, the program will run any registered exit functions, including f3(). Since f3() calls abort(), the program will stop immediately.

KILL CALL:

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
printf("Hello");
kill(getpid(), 9);
printf("Goodbye");
}
```

You are already familiar with getpid() system call. To find out what 9 is, first look at the output of the command: kill -l

Find out the word mentioned next to 9. Search on the internet for this.

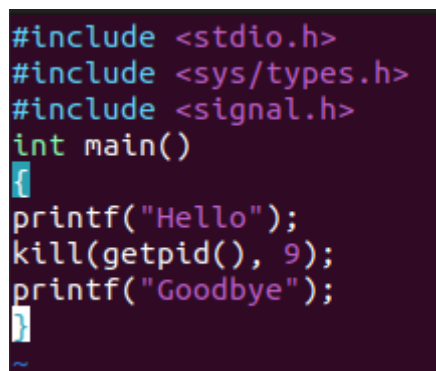
ANSWER:

In the context of the `kill` command and signal handling, the number `9` corresponds to the "SIGKILL" signal

This signal is used to forcefully terminate a process, and it cannot be caught or ignored by the process.

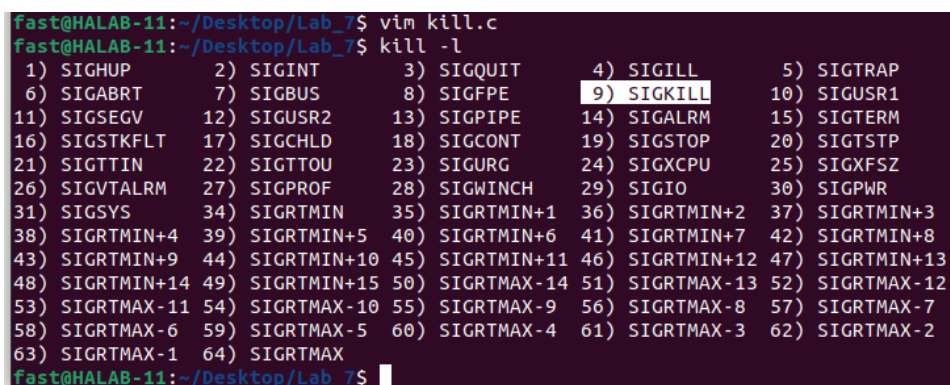
When I run the provided code, the output will be "Hello" but the program terminated before it can print "Goodbye" **because `kill(getpid(), 9);` sends the `SIGKILL` signal to itself, causing immediate termination.**

PROGRAM WRITTEN IN VIM:



```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
printf("Hello");
kill(getpid(), 9);
printf("Goodbye");
}
```

OUTPUT:

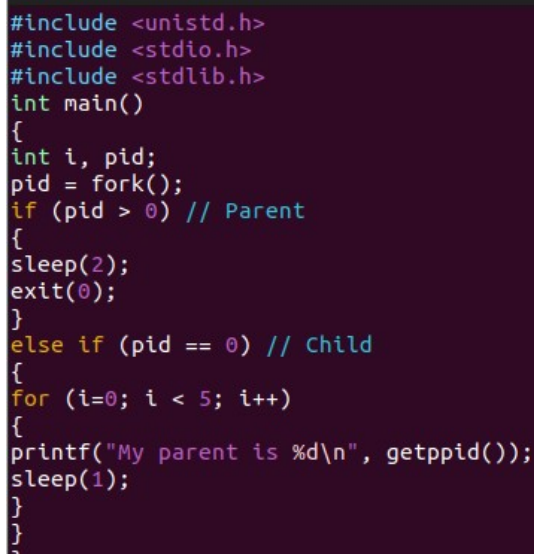


```
fast@HALAB-11:~/Desktop/Lab_7$ vim kill.c
fast@HALAB-11:~/Desktop/Lab_7$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH   29) SIGIO      30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
fast@HALAB-11:~/Desktop/Lab_7$
```

Parent Dies Before Child:

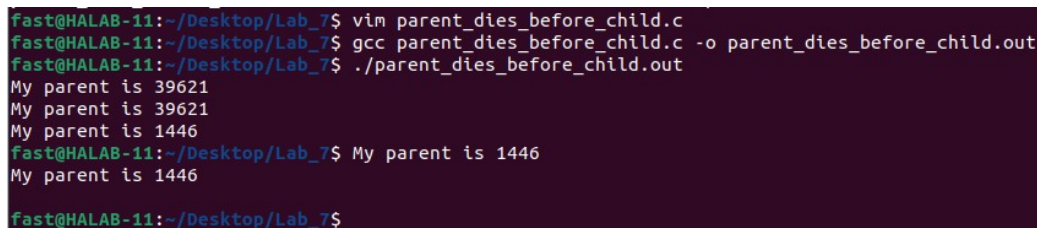
```
#include <unistd.h>
#include <stdio.h>
int main()
{
    int i, pid;
    pid = fork();
    if (pid > 0) // Parent
    {
        sleep(2);
        exit(0);
    }
    else if (pid == 0) // Child
    {
        for (i=0; i < 5; i++)
        {
            printf("My parent is %d\n", getppid());
            sleep(1);
        }
    }
}
```

Run the code.

A screenshot of a terminal window with a dark background and light-colored text. The code is the same as the one in the previous block, but with syntax highlighting: keywords like 'int', 'if', 'else', 'for', 'printf', 'sleep', and 'exit' are in blue, string literals are in red, and comments are in green. The code is displayed line by line, matching the original source code.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int i, pid;
    pid = fork();
    if (pid > 0) // Parent
    {
        sleep(2);
        exit(0);
    }
    else if (pid == 0) // Child
    {
        for (i=0; i < 5; i++)
        {
            printf("My parent is %d\n", getppid());
            sleep(1);
        }
    }
}
```

OUTPUT:

A screenshot of a terminal window showing the execution of the program. The prompt is 'fast@HALAB-11:~/Desktop/Lab_7\$'. The user enters 'vim parent_dies_before_child.c', then 'gcc parent_dies_before_child.c -o parent_dies_before_child.out', and finally './parent_dies_before_child.out'. The output shows five lines: 'My parent is 39621', 'My parent is 39621', 'My parent is 1446', 'fast@HALAB-11:~/Desktop/Lab_7\$ My parent is 1446', and 'My parent is 1446'. The prompt returns at the end.

```
fast@HALAB-11:~/Desktop/Lab_7$ vim parent_dies_before_child.c
fast@HALAB-11:~/Desktop/Lab_7$ gcc parent_dies_before_child.c -o parent_dies_before_child.out
fast@HALAB-11:~/Desktop/Lab_7$ ./parent_dies_before_child.out
My parent is 39621
My parent is 39621
My parent is 1446
fast@HALAB-11:~/Desktop/Lab_7$ My parent is 1446
My parent is 1446
fast@HALAB-11:~/Desktop/Lab_7$
```

1. What are the PPID values you are receiving from the for loop?

ANSWER:

The `getppid()` function retrieves the parent process ID (PPID)

In the beginning, when the child process runs, it will print the PPID of its parent (the original parent process)

After the parent process (`pid > 0`) exits after sleeping for 2 seconds, the child will continue to run but its PPID will change to that of the init process (usually PID 1)

Therefore, you will see the PPID change from the original parent's PID to 1 after the parent exits.

2. What has happened when the numbers of the PPID change?

ANSWER:

When the parent process exits (after 2 seconds), the child process's PPID changes because in linux, when a process's parent terminates, the child process is adopted by the init process. This is why the PPID will show as 1 for the remaining iterations of the loop. This mechanism ensures that orphaned processes are still managed by the system.

3. What is now PID of the init process?

ANSWER:

The PID of the init process is typically 1

This process is responsible for starting and managing system processes and is the ancestor of all other processes

IN SHORT:

- Initially, the PPID is the original parent's PID.
 - Once the parent exits, the PPID changes to 1 (the PID of the init process).
 - The PID of the init process is 1.
-