**NAME:: Rayan Shabbir**

**Roll No::BITF20M535**

***OPERATING SYSTEM***

LAB # 03

\*\*\***Compilation Process\*\*\***

Task :: 01

1. To create preprocessed file we use

gcc -E lab03.c -o lab03.i

1. To create assembly code we use

gcc -S lab03.i -o lab03.s

1. To create Object file we use

gcc -c lab03.s -o lab03.o

1. To create executebale file we use

gcc lab03.o

**\*\*\*Extracting Information from Binaries\*\*\***

Task :: 01

gcc cat.c -o cat\_dynamic

This command is use to compile and link binary file “dynamically”.

gcc –static cat.c -o cat\_static

This command is use to compile and link binary file “statically”.

Running By commands::

cat ./cat\_static /proc/self/maps

cat ./cat\_dynamic /proc/self/maps

Difference b/w these two::

* When we compile and link like static the size of file increases becauses it contain all the definations or predefined libraries code in the code itself stataically (One of the reason also maybe that all the code of routines called by our program became part of our program that’s why its size increases).
* But when we compile like dynamically the size of file decreases because it only got the definition or code of instruction which our program called (it needed) at run time by going maybe at usr/bin/… etc.
* By gcc –static cat, the file we get will open in format of statically version, while by gcc cat, the file we got will open in dynamically format.
* The main difference I got from these file when I checked : file cat\_static , I got that file is statically link, and when I checked : file cat\_dynamic, I got that file is dynamically link, Moreover, the static files has some resources included like LSB, but in the dynamic case those were not.

Task :: 02

1. objdump -Mintel -d ./a.out

This command is use to generate the disassembly of our executebale file in intel format.

1. readelf -S ./a.out

To see contents of our program on disk we use simply readelf and for displaying section headers information we use above command readelf with option -S. It will display the sections contents of our file. And in the first line it displays the count of section headers, which was 31 in our case.

(It will also displays address from where sections start etc)

1. readelf -l ./a.out

This command is used to display the program header of our executeable file. It gives the all related information about our program how it executes and run It has an array like data structure which includes different segments, each segment displaying program related information.

1. readelf -h ./a.out

This command will display ELF header Table. This executebale and linable Format Table is used to diplay contents about object file, name of file, size, type, version and path where it stored in memory.

Task :: 03

Firstly, gcc cat.c -o cat.out

* readelf 1 cat.out

This command will displays all the contents of our executed cat.c file.

* od cat.out

This command will display the octal format of our executeable file.

If we want to display hex format of our file we can use option -x

with it.

* size cat.out

This command will display the size of are executebale file. First it displays total text, then total initialized data, then amount of uninitialized data and then decimal and hex format of our file along with the file name.

gcc -g cat.c -o cat2.out

Basically -g option with gcc is used to generate the debugging information of our file.

But when we run these commands on it,

* readelf cat2.out
* od cat2.out
* size cat2.out

There will be no difference among these and previous files,

Similarly, if I do

file cat.out

or

file cat2.out

Both results are same and both are dynamically linked.

gcc –static cat.out -o stat.out

Here, we compile and link our file statically

But by gcc or gcc -g it compile and link dynamically

So, when we run these commands

* readelf stat.out

It will display the content on disk of our statically linked file.

* size stat.out

By this the size of file we seen, has been increased as compared to the previous ones, It shows this behavoiur because when we compile statically it included all the functions in out program which are called by out program.

* od stat.out

Similarly, here the octal we got of out file is also increased, means more sections are created here, and more no of contants with their octals are generated.

1. cat ./static.out

If we try to debug the statically linked file, then a huge result as a file we got, because it has all the systems called cuntions included in it.

cat ./cat.out

But by this debugging a small file display which has less content because it add code only at run time.

**\*\*\*Process Management\*\*\***

Task:: 01

1. We can use command :: jobs

To see which process are running or which processes are not running in our terminal.

We can get info about all those processes which are running in our system by using command:: ps

ps means “Process Status”.

It display the information of processes.

1. First we create a job that is running in the background

Sleep 1000 &

And we know (or we get by jobs), the pid of this process, which will be 6

Then, if we do

fg 6

Then this process i.e sleep 1000 will work / run in foreground.

Then we can use <ctrl+z> to suspend it.

1. We can use the command:: upime

It will display the amount of time from which system is running, no of users using it, and also the load average.

1. We can use the command:: free

It will display the amount of total, free, shared and used memory of the system

Task:: 02

sleep 50 🡪 This process will run in the foreground

<ctrl+z> is used to suspend the command, and by this we also gets its pid

bg [pid] 6 🡪 It will run the suspended process in the background.

* And we got a message when it terminates (execution completes)

sleep 50 & 🡪 This process will run in the background.

fg 6 🡪 It will bring the process that is running in the background to the foreground.

* On other terminal::

ps -u :: By this we see the process STATUS. As firstly process is running in the background so it does not show any sign.

But after that when process running in foreground process status is written by ‘+’ sign.

ps -l

**\*\*\*Zombie Processes\*\*\***

Task:: 01

A)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

int main()

{

    int pid = fork();

    if (pid != 0)

    {

        printf("Parent Running, PID=%ld\n", (long)getpid());

        while (1);

    }

    else

    {

        printf("Child Terminating, PID = %ld\n",(long)getpid());

        exit(0);

    }

    return 0;

}

B) We can simply verify that a process has became zombie by using “ps” command.

If we write

ps -u

This command give us 11 column information, one of which shows status of our processes. If our process is zombie then we got ‘Z’ in that field.

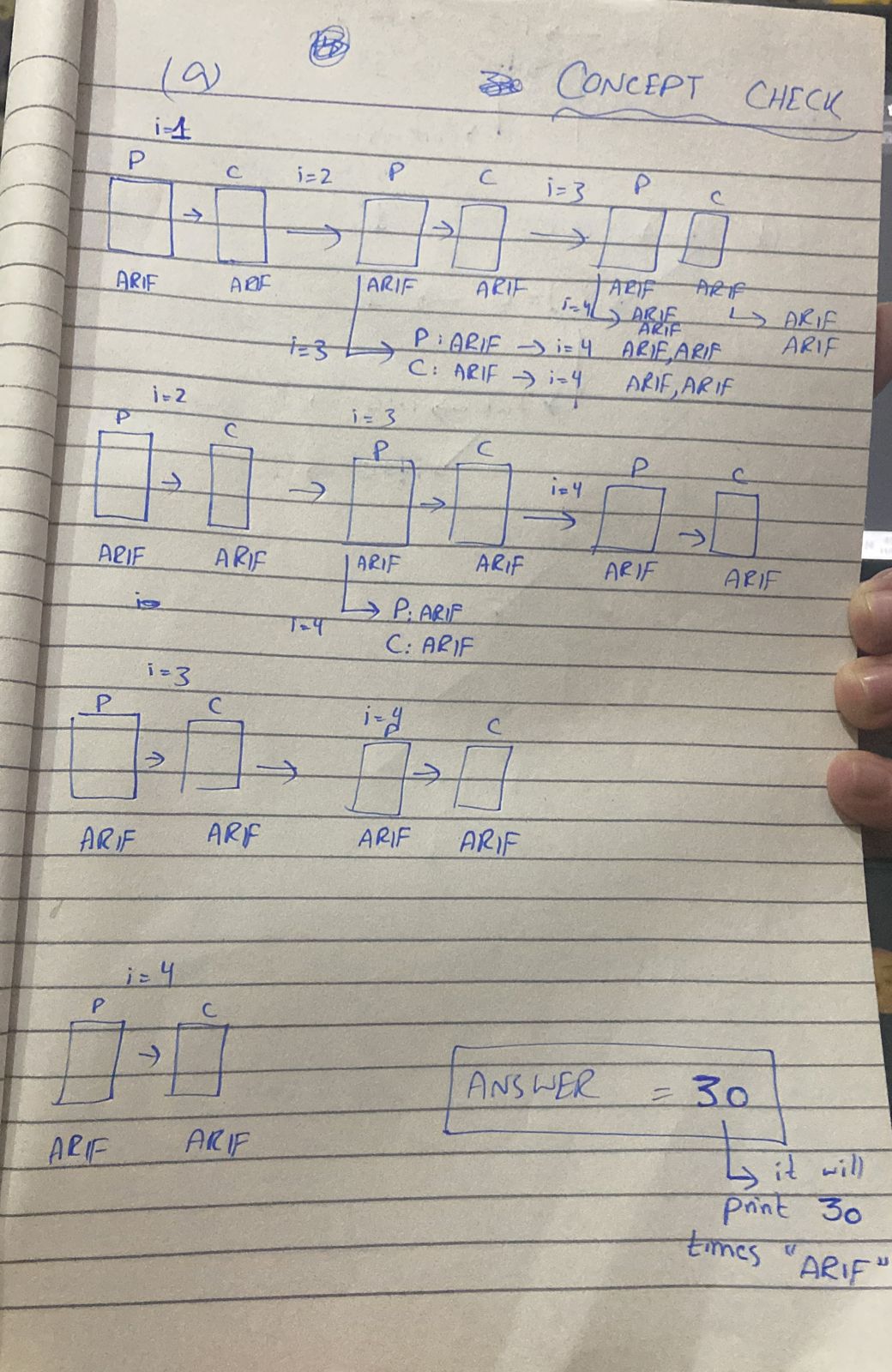
Moreover, ps command writes keyword ‘defunct’ for zombie.

**\*\*\*Concept Check\*\*\***

Task:: 01

It will print/displays 30 times ARIF.

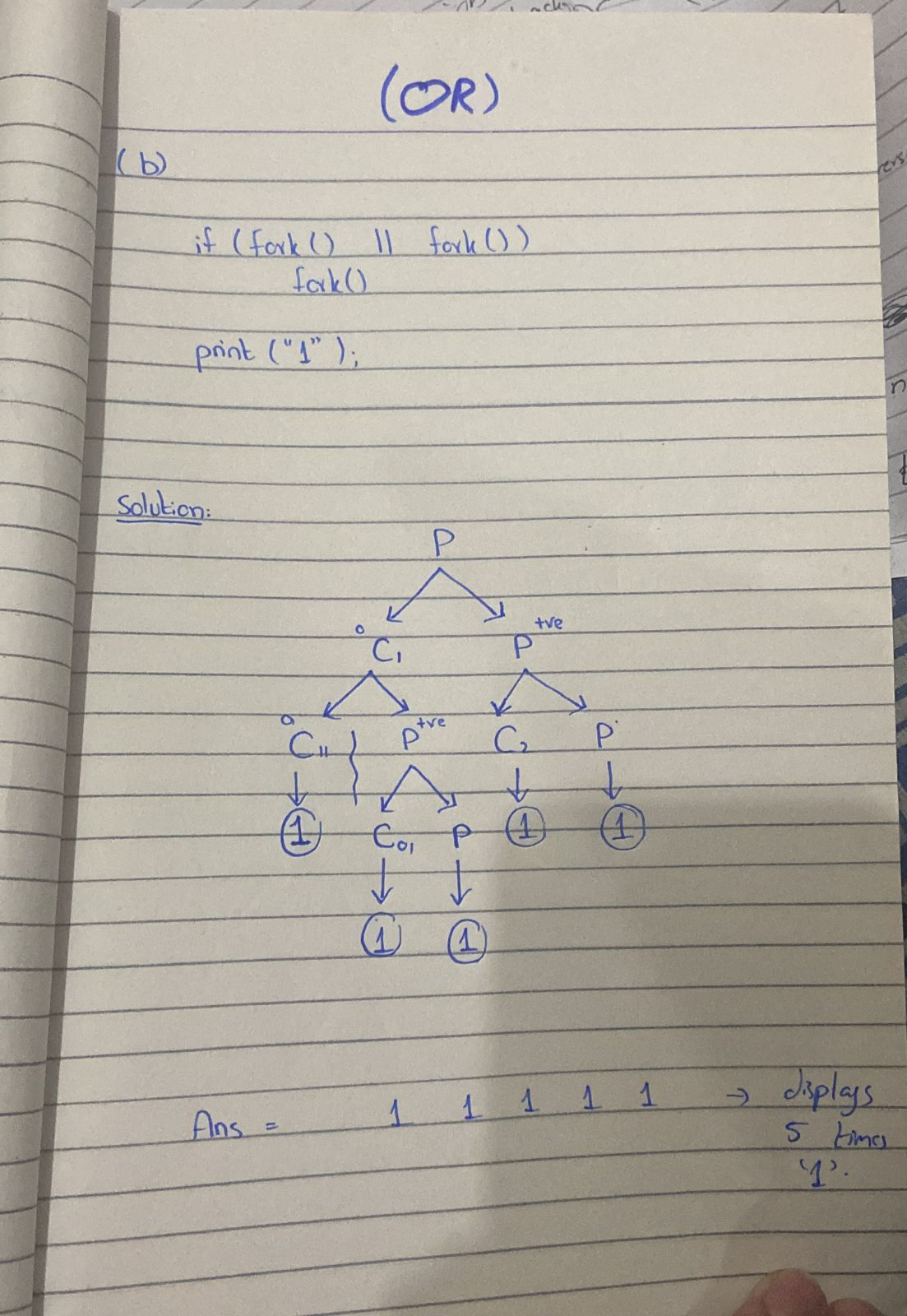
Solution:



Task:: 02

It will display 5 times “1”.

Solution:



Task:: 03

It will create a zombie process. Because parent remain executing whether the child terminates.

So, it don’t return us the prompt

Output::

I am not a zombie process

cont.

Proof:: So, if now we check by

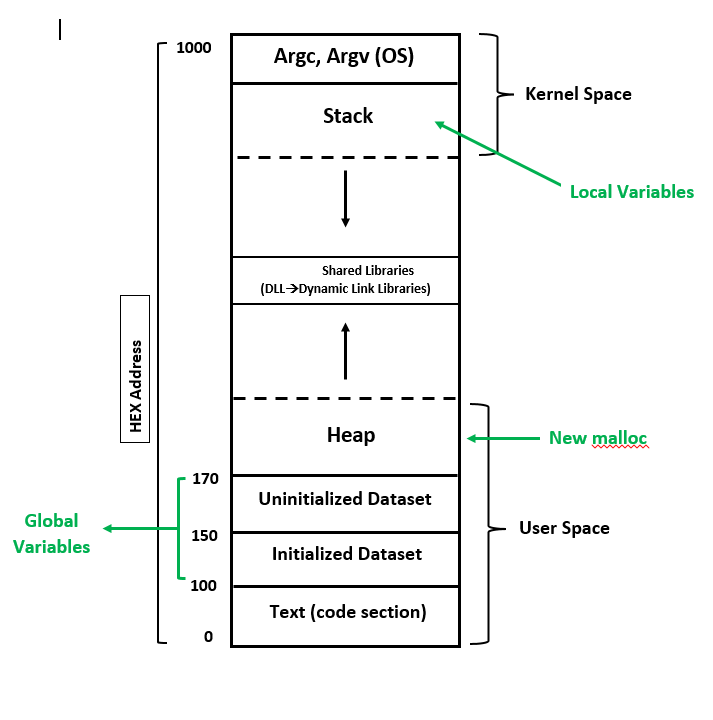
ps -u

Then, it will return us that that process is became zombie.

Child process became **ZOMBIE**.

The child process is not reaped by parent because parent is busy i.e executing infinite loop. And, the child has finished his work. So, init takeover it.

Task:: 04



1. Because “**cake”** is a global initialized variable, it is placed in the Initialized data section in process address space, which lies above the code section. While, it will exist in the format of “.data” in program hard disk.
2. As **“another”** array is globally uninitialized, it will be placed in “Uninitialized data” section in process address space, which lies above the initialized data section. While, it will exist in the format of “.bss” in program hard disk.
3. As **“calender[4]”** array is globally initialized, it will be placed in “Initialized data” section in process address space, which lies above the code section. While, it will exist in the format of “.data” in program hard disk.
4. **“Hello”** is an initialized local variable. It will be placed inside the stack section in process address space, which keeps growing in the downward direction.
5. Array **“X”** is an initialized local array. It will be placed inside the stack section in process address space, which keeps growing in the downward direction, below the kernel space.
6. **“writer”** is a dynamic array which is being created on heap. So, it will be placed in the heap section which is above the uninitialized dataset in process address space, which keeps growing in upward direction.

The parameters in main() function are **environmental variables** and **command line arguments (argc, argv).** These are placed in the segment below the kernel space where environment variables and command line arguments lie.

**BONUS TASKS**

TASK :: 01

1. To display hostname we have following commands in linux:

echo $HOSTNAME or hostename :: It will display hostname along with hypervisor name

whoami :: It will display the user name (host name) which is working on this machine.

1. We can change the HOSTNAME Environmental variable by following commands/steps:
2. set (Used to display all the environmental variables, also those which are not exported)
3. echo $HOSTNAME (This step is optional it is only used to display current hostename)
4. HOSTNAME = Miffy
5. echo $HOSTNAME
6. Printing all the Environmental Variables on stdout

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

{

    for(char \*\*environ = envp; \*environ != 0; environ++) {

        char \*newEnv = \*environ;

        printf("%s\n", newEnv);

    }

    return 0;

}

1. We can use the following commands to display all the environmental variables on terminal:

* env (It will display only exported environmental variables)
* set (It will display exported as well as not exported environmental variables, it will also display functions)

1. C program to take 2 numbers from command line and display their sum on stdout.

#include <stdio.h>

#include <stdlib.h>

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

    if(argc != 3) {

        printf("Enter 2 command line arguments...");

        exit(0);

    }

    printf("The value of argc is %d\n", argc);

    int sum = 0;

    for (int i = 1; i < argc; i++)

    {

*// sum = sum + (\*argv[i] - '0');*

        sum = sum + atoi(argv[i]);

*// printf("\nThe argument at index %d has value %s", i, argv[i]);*

    }

    printf("\nSum = %d", sum);

    printf("\n");

    return 0;

}

1. This program will display the name of the Program / file.

#include <stdio.h>

#include <stdlib.h>

int main(int agrc, char \*argv[], char \*evp) {

    int i = 0;

    printf("Program name:: %s", argv[i]);

    printf("\n");

    return 0;

}

1. **EXPORT COMMAND** ::

All the variables which we create in the parent shell, can only be accessed in the shell(parent shell) where we created them. Those variables are not accessible in the child shells below the parent shell. If we create a variable

name = "Rayan"

echo $name

It will print "Rayan" ⬆️

But if we change the shell:

/bin/bash

echo $name

It will not print anything.

If we want to access those variables in the child shells, which we created in parent shell, we will use "export command".

Export is a built-in command which export the variables from your (parent) shell to the child shells, and you can easily access them there(child shell).

If we export "name",

export name

/bin/bash

echo $name

Now, it will print "Rayan" ⬆️

This printing will be in the child shell.

Task :: 02

1. This program will only take 1 command line argument.

#include <stdio.h>

#include <stdlib.h>

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

    if(argc != 2) {

        printf("Command Line Arguments, limit exceeded...");

        exit(0);

    }

    int id = 55;

    printf("My name is Rayan with id: %d, And I am not a terrorist...", id);

    return 0;

}

1. Using fork() and running the execve in the child process using user arguments

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

{

    if (argc < 2)

    {

        printf("Command line Arguments limit exceeds....\n");

        exit(0);

    }

    char \*newargv[] = {"/bin/ls", NULL, NULL, NULL};

    char \*enivp[] = {NULL};

    newargv[0] = argv[1];

    if (argc > 2)

    {

        int j = 2;

        for (int i = 1; i < (argc - 1); i++)

        {

            newargv[i] = argv[j];

            j++;

*// printf(newargv[i]);*

*// printf("\n");*

        }

*// for (int i = 0; i < (argc - 1); i++)*

*// {*

*//     printf(newargv[i]);*

*//     printf("\n");*

*// }*

    }

*// printf("Start");*

    int cpid = fork();

    if(cpid == 0) {

        execve(argv[1], newargv, enivp);

*// execve("/bin/ls", newargv ,enivp);*

        printf("Hello, I am Child\n");

    } else {

        printf("Hello, I am Parent\n");

    }

    return 0;

}

1. Parent waiting and child executing execve by using user argument

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

{

    if (argc < 2)

    {

        printf("Command line Arguments limit exceeds....\n");

        exit(0);

    }

    char \*newargv[] = {"/bin/ls", NULL, NULL, NULL};

    char \*enivp[] = {NULL};

    newargv[0] = argv[1];

    if (argc > 2)

    {

        int j = 2;

        for (int i = 1; i < (argc - 1); i++)

        {

            newargv[i] = argv[j];

            j++;

*// printf(newargv[i]);*

*// printf("\n");*

        }

*// for (int i = 0; i < (argc - 1); i++)*

*// {*

*//     printf(newargv[i]);*

*//     printf("\n");*

*// }*

    }

*// printf("Start");*

    int cpid = fork();

    if(cpid == 0) {

        execve(argv[1], newargv, enivp);

*// execve("/bin/ls", newargv ,enivp);*

        printf("Hello, I am Child\n");

    } else {

        wait(NULL);

        printf("Hello, I am Parent\n");

    }

    return 0;

}



#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

{

    if (argc < 2)

    {

        printf("Command line Arguments limit exceeds....\n");

        exit(0);

    }

    char \*newargv[] = {"/bin/ls", NULL, NULL, NULL};

    char \*enivp[] = {NULL};

    newargv[0] = argv[1];

    if (argc > 2)

    {

        int j = 2;

        for (int i = 1; i < (argc - 1); i++)

        {

            newargv[i] = argv[j];

            j++;

*// printf(newargv[i]);*

*// printf("\n");*

        }

*// for (int i = 0; i < (argc - 1); i++)*

*// {*

*//     printf(newargv[i]);*

*//     printf("\n");*

*// }*

    }

*// printf("Start");*

    int cpid = fork();

    if(cpid == 0) {

        execve(argv[1], newargv, enivp);

*// execve("/bin/ls", newargv ,enivp);*

        printf("Hello, I am Child\n");

    } else {

        int pid = wait(NULL);

        printf("Hello, I am Parent\n");

        printf("Pid of child is:: %d", pid);

    }

    return 0;

}

Task:: 03

#include <stdio.h>

#include <math.h>

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

{

    for (int idx = 0; envp[idx] != NULL; idx++)

    {

        printf("%s\n", envp[idx]);

    }

    return 0;

}

This above program will print all the environmental variables one, by one on stdout.

1. I place the above executed program in bonus7.out and then I execute it through child process by using execl, by giving it’s a absolute path.

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <fcntl.h>

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

    int cpid = fork();

    if(cpid == 0) {

        execl("/home/rayan/myfolder/bonus7.out", "bonus7", NULL);

        printf("Child...");

    } else {

        wait(NULL);

        printf("Parent...");

    }

}

b)

Yes, we can do it. When we run the file, if the environmental variable which we are passing is NULL, instead of other environmental (original) variables. Then it will print NULL, which means that our program will print out nothing and only it shows NULL.

c)

Yes, we can print the output of our choice when we run our program. While executing, if we change the environmental variables which are being passed, if they are changed, then the output will be different.

For this purpose, we can use pipe operator as well.

For example:

./bonus7.out | grep PATH