

Question no 1:

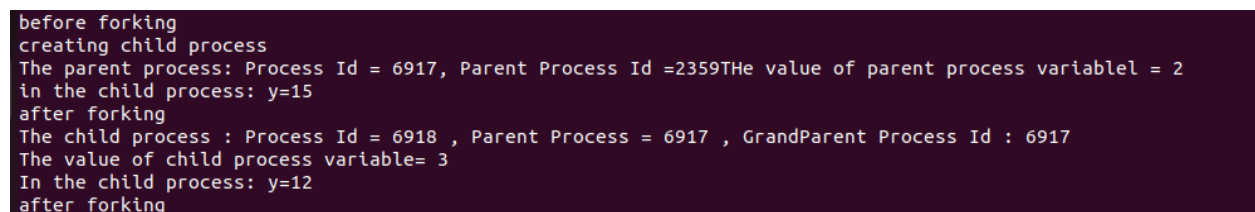
Code :



The screenshot shows a terminal window titled 'ayesha@ayesha: ~/Desktop' with a file named 'labTask.c' open in the nano 6.2 editor. The code is a C program that demonstrates process forking. It includes standard headers, defines parent and child functions, and a main function that forks a child process. The child process increments a variable 'y' and prints its own and its parent's process IDs. The parent process prints its own process ID and the child's process ID. The code is as follows:

```
GNU nano 6.2 labTask.c *
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void parent(int cvar);
void child(int pvar);
int y = 10;
int main(){
    int x = 0;
    printf("before forking \n");
    printf("creating child process \n");
    int i = fork();
    if (i==0){
        child(x);
    }
    else{
        parent(x);
    }
    printf("after forking \n");
    return 0;
}
void child(int a){
    y+=2;
    a=3;
    printf("The child process : Process Id = %d , Parent Process = %d , GrandParent Process Id : %d \n",getpid(),getppid(),getppid());
    printf("The value of child process variable= %d\n",a);
    printf("In the child process: y=%d\n",y);
}
void parent(int b){
    b=2;
```

Output:



The screenshot shows the output of the program execution in a terminal window. The output is as follows:

```
before forking
creating child process
The parent process: Process Id = 6917, Parent Process Id =2359The value of parent process variable= 2
in the child process: y=15
after forking
The child process : Process Id = 6918 , Parent Process = 6917 , GrandParent Process Id : 6917
The value of child process variable= 3
In the child process: y=12
after forking
```

Question no 2:

Code :

```

GNU nano 6.2                                labTask.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int i, pid[3];
    // create 3 children
    pid[0]=fork();
    if(pid[0]==0){
        printf("Child 1 with pid %d is created\n",getpid());
        sleep(2);
        printf("Child 1 with pid %d is terminated\n",getpid());
        exit(0);
    }

    pid[1]=fork();
    if(pid[1]==0){
        printf("Child 2 with pid %d is created\n",getpid());
        printf("Child 2 with pid %d is terminated\n",getpid());
        exit(0);
    }

    pid[2]=fork();
    if(pid[2]==0){
        printf("Child 3 with pid %d is created\n",getpid());
        printf("Child 3 with pid %d is terminated\n",getpid());
        exit(0);
    }

    // parent process
    sleep(1);

```

```

    // parent process
    sleep(1);
    printf("Parent process with pid %d is waiting for children to exit\n", getpid());
    for (i = 0; i < 3; i++) {
        waitpid(pid[i], NULL, 0); // wait for each child to exit
    }
    printf("Parent process with pid %d is terminating\n", getpid());
    exit(0);
}

```

Output :

```

ayesha@ayesha:~/Desktop$ ./lab
Child 1 with pid 9405 is created
Child 3 with pid 9407 is created
Child 3 with pid 9407 is terminated
Child 2 with pid 9406 is created
Child 2 with pid 9406 is terminated
Parent process with pid 9404 is waiting for children to exit
Child 1 with pid 9405 is terminated
Parent process with pid 9404 is terminating

```

Question no 3:

`fork()` creates a new process by duplicating the calling process, while `exec()` replaces the current process with a new process image. `fork()` creates a child process with the same memory state as the parent process, while `exec()` loads a new executable file into the current process's address space and runs it. Together, `fork()` and `exec()` are often used to create a new process and then replace it with a different program.

Question no 4:

```
ayesha@ayesha:~/Desktop$ ./lab | grep -c "Hello"
17976
```

Question no 5:

```
ayesha@ayesha:~/Desktop$ pkill -9 -f ./lab
ayesha@ayesha:~/Desktop$ ps -al
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   1000     1629     1621  0  80   0 - 58008 do_pol  tty2        00:00:00 gnome-session-b
4 R   1000    54850    9841  0  80   0 - 5419 -      pts/0        00:00:00 ps
ayesha@ayesha:~/Desktop$
```

Question no 6:

Exit System Call:

The `exit()` system call is used to terminate a program and return an exit status to the parent process or the operating system. It takes a single argument, which is the exit status of the program. The exit status is a number between 0 and 255, where 0 indicates successful termination and any other value indicates an error or abnormal termination.

Example:

Program 1: Normal Termination

This program simply prints a message and exits with a status of 0, indicating successful termination:

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

int main() {
    printf("Program has ended successfully.\n");
    exit(0);
}
```

Program 2: Abnormal Termination

This program attempts to open a non-existent file and exits with a status of 1, indicating an error:

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

int main() {
    FILE *fp = fopen("non_existent_file.txt", "r");
    if (fp == NULL) {
        printf("Error: Could not open file.\n");
        exit(1);
    }
    fclose(fp);
    return 0;
}
```