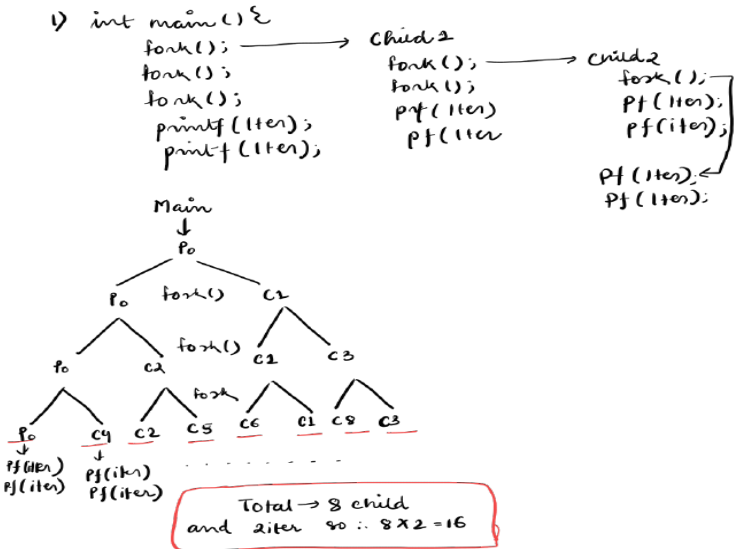
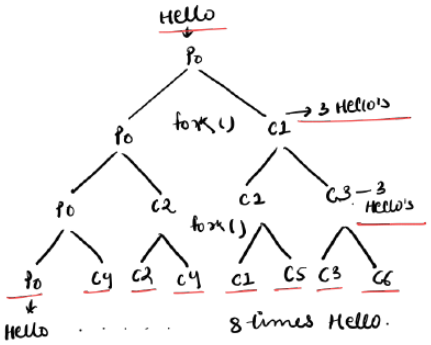
C MINOR ASSIGNMENT- 08 {fork()}

```
int main(void) {
   fork();
   fork();
   fork();
   printf("ITER\n");
   printf("ITER\n");
   return 0;
}
/* Any formula can be
    devised for number
     of processes
    created here?
If so, state.*/
```



2. Construct the process tree di following code snippet.

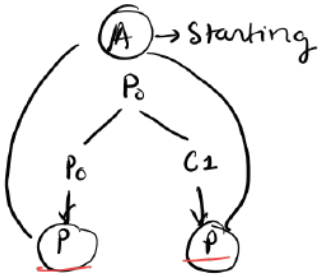
```
int main(void){
  printf("hello\n");
  fork();
  printf("hello\n");
  fork();
  printf("hello\n");
  fork();
  printf("hello\n");
  return 0;
}
/* Any formula for
    nuber of outputs?
If so, state.*/
```



total = 1 (Initially) +
3 (After 1st call) +
2 (After 2nd call) +
1 (After 3rd call) +
8 (Child Call)

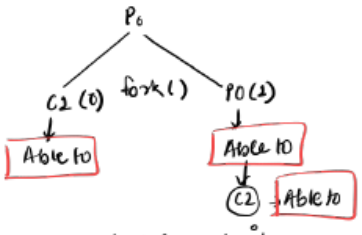
Total= 15

```
3) int main(){
     printf('A');
     fork();
     printf("P/n");
     return 0;
   }
```



O/t: AP (Parent)
AP C (Child)

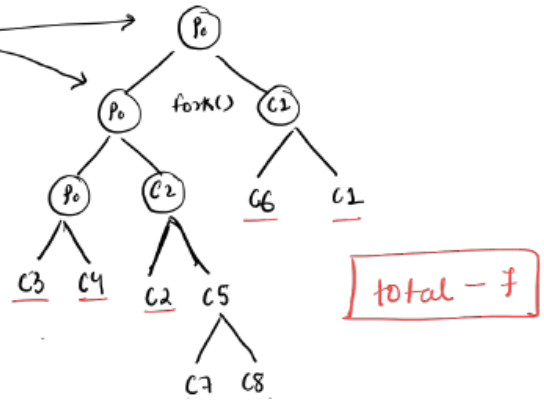
4) int main(){
fork() && fork()
Pf("Able to");
return 0;
}



3

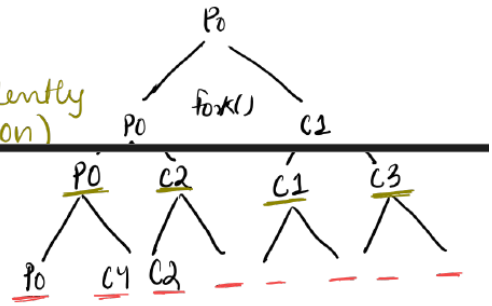
&& → will evaluate the second fork() only if the first fork() is true (non-zero).

5) int main(){
fork();
fork() && fork();
fork();
printf("Got");
return 0;
}



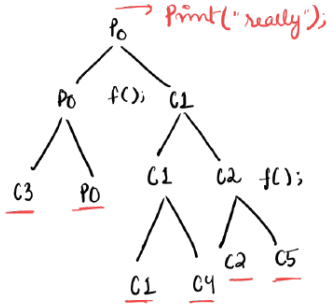
total – 7

6) int main(){
fork();
fork() + fork(); (Independently Execution)
return 0;
}



total = 9

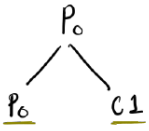
7) int main(){
fork();
fork() || fork();
fork();
printf("really");
return 0;
}



total – 7

|| means that second fork will execute only when child is 0.

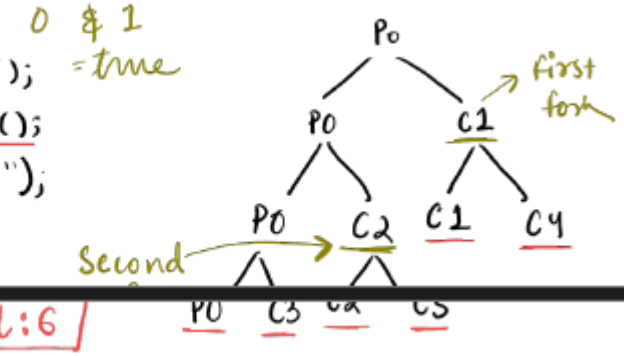
8) int main(){
fork();
fork() && fork() || fork();
fork();
Pf("Guess");
return 0;
}



It depends.

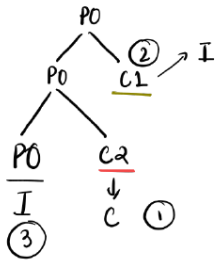
9) int main(){
fork() && fork();
fork() || fork();
printf("Hi/n");
return 0;
}

0 & 1 = true



10) pid = fork() → the value (it's not 0 so it'll go to else block)

output:
C
I
I

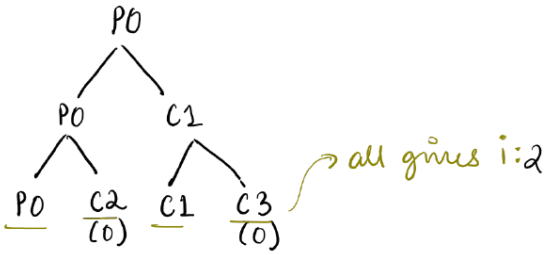


11. Construct the process tree following code snippet.

```
int
main(void){
 pid_t childpid;
 int i, n=3;
 for(i=1;i<n;i++){
  childpid=fork();
  if(childpid==-1)
     break;
 }
 printf("i:%d\n",i);
 return 0;
}
```

11)

o/t: i:2
i:2
i:2
i:2

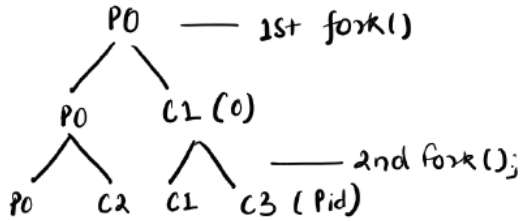


12. Draw the process tree because of as **x≠0**will be displayed.

```
pid_t
add(pid_t a, pid_t b){
   return a+b;
}
int main(void){
 pid_t x=10;
 printf("%d\n",x);
 x=add(fork(),fork());
 printf("%d\n",x);
 return 0;}
```

12)

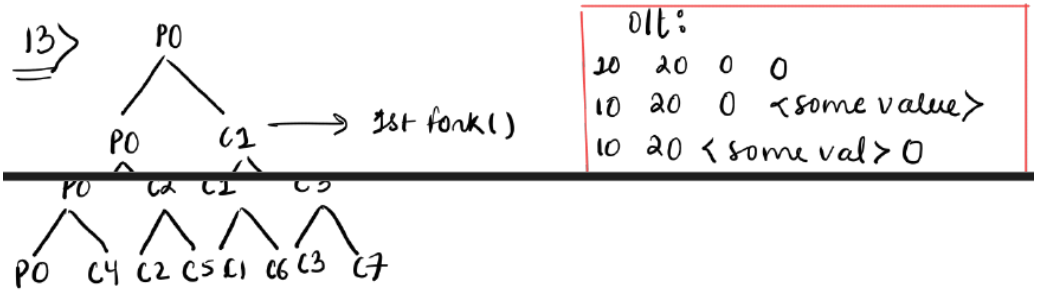
output:
10
<x in p>
10
<x in c1>
10
0
10
0



13. Determine the total number of displayed for the given

```
int main(void){
 int x[]={10,20,fork(),fork()+fork()};
 int len=sizeof(x)/sizeof(int);
 for(int i=0;i<len;i++)
   fprintf(stderr," %d ",x[i]);
 printf("\n");
 return 0;
}
```

13)



o/t:
10 20 0 0
10 20 0 <some value>
10 20 <some val> 0

14. Determine the number of process(s) also write the output.

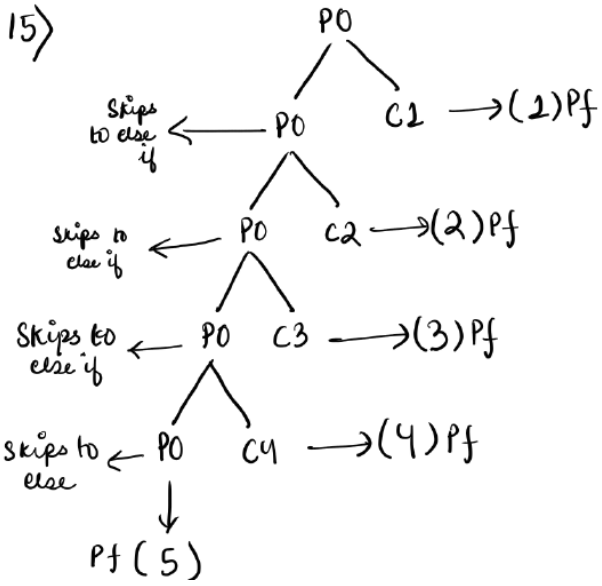
```
void show(){
  if(fork()==0)
   printf("1\n");
  if(fork()==0)
   printf("2\n");
 if(fork()==0)
   printf("3\n");
}
int main(void){
 show();
 return 0;
}
```

14) The output will varry no. of time. the number 1,2,3 are printed multiple times by different processes. The order depends on process scheduling.

15. Draw the process tree of the fo the following code. Can the cc

```
int main(void){
 if(fork()==0)
  printf("1\n");
else if(fork()==0)
  printf("2\n");
else if(fork()==0)
  printf("3\n");
else if(fork()==0)
  printf("4\n");
else
  printf("5\n");
return 0;
}
```

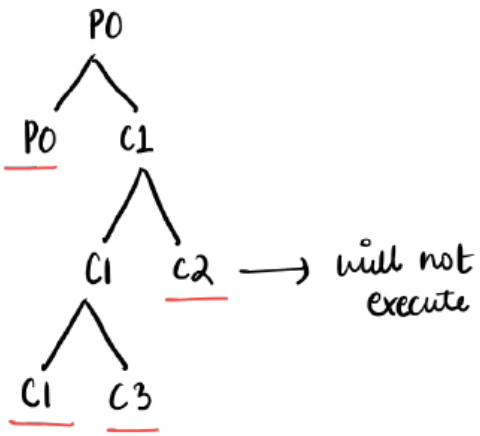
15)



Olt: 1 2 3 4 5

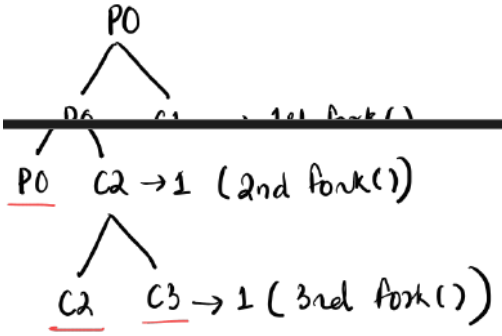
16. Find the output of the code

```
int main(){
 pid_t p1,p2;
 p2=0;
 p1=fork();
 if (p1 == 0)
    p2 = fork();
 if (p2 > 0)
    fork();
 printf("done\n");
 return 0;
 }
```



Olt: done done done done

```
int main(){pid_t c1=1,c2=1;
 c1=fork();
 if(c1!=0)
    c2=fork();
 if(c2==0){
    fork();printf("1\n");
  }
return 0;}
```

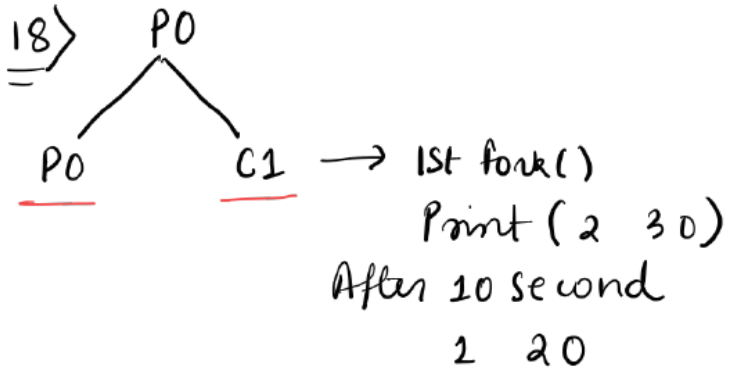


C2 & C3 will generate 1

print anything because they do not enter the if(c2==0)

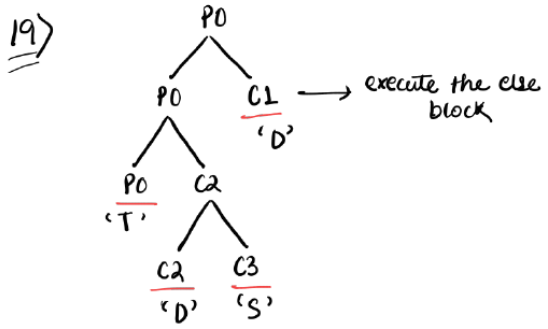
18. Find the output of the code segment.

```
int main(){
  struct stud s1={1,20};
  pid_t pid=fork();
  if(pid==0){
     struct stud s1={2,30};
     printf("%d %d\n",s1.r,s1.m);
     return 0;
  }
  else{
    sleep(10);
    printf("%d %d\n",s1.r,s1.m);
    return 0;
  }
}
```



19. Find the output of the code s

```
int main(){
  if(fork()){
    if(!fork()){
      fork();
      printf("S ");
    }
    else{
      printf("T ");
    }
  }
  else{
       printf("D ");
  }
  printf("A ");
  return 0;
}
```

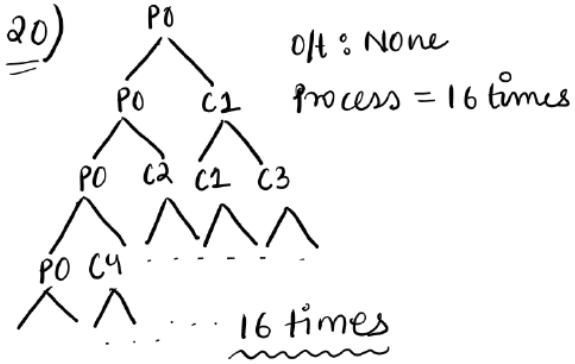


the output will depend on how the OS schedules the processes, but Generally it'll follow above tree.

'T S S A D A A A'

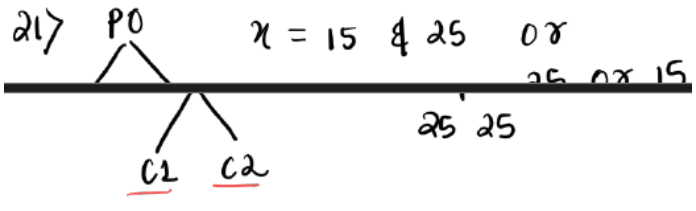
20. Calculate the number of proce

```
int main(){int i;
  for(i=0;i<12;i++){
     if(i%3==0){
          fork();
     }
  }
  return 0;
}
```



21. State the possible values of x for the given

```
int x;
int a[2]={10,20};
x=5+a[fork() || fork()];
printf("%d ",x);
```



22. Suppose four user-defined exit handlers X, Y, P, and Q are installed in the order X then Y then P then Q using atexit() function in a C program. Exit hadler X is designed to display 1, Y is designed to display 2, P is designed to display 3, and Q to display 4. State the order of their display, when the program is going to terminate after calling return 0/exit(0).

(A) 4, 3, 2 1 (C) 1, 2, 4, 3

(B) 1,2,3,4 (D) none

Choice

Order of Exit Handlers:

Explanation:

- Exit handlers (`atexit`) execute in reverse order of registration.
- Registered as: X, Y, P, Q.
- Execution order: Q (4), P (3), Y (2), X (1).

Output: (A) 4, 3, 2, 1.

23. You know that the **ps** utility in UNIX reports a snapshot of the current processes. Determine the state code of the given program, that became a process.

```
int main(void){
 fprintf(stderr,"PID=%ld\n",(long)getpid());
 while(1);
 return 0;
}
```

Explanation:

- Infinite loop without any blocking operations.
- Process state is `R` (running).

Output: (A) R.

24. Find the process state code of the given program, that became a process using the Unix utility **ps**. As you know **ps** displays information about a selection of the active processes.

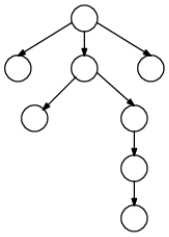
```
int main(void){
 fprintf(stderr,"PID=%ld\n",(long)getpid());
 while(1)
     sleep(1);
 return 0;
}
```

Explanation:

- `sleep(1)` puts the process in a `SLEEP` state (`S`) between executions.

Output: (B) S.

25. Develop a C code to create the following process tree. Display the process ID, parent ID and return value of `fork()` for each process.



OBSERVARIONS:

- Use ps utility to verify the is-a-parent relationship?
- Are ypu gettig any orphan process case?
- Are you getting any ZOMBIE case?

Figure 1: Process tree

```c
/*Output Description:
Displays the parent process ID.
Forks create two child processes.
Prints the PID and parent PID for each process.
Verifies process hierarchy using the ps utility.*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
    pid_t pid1, pid2;
    printf("Parent PID: %d\n", getpid()); // Display parent PID
    pid1 = fork(); // First fork
    if (pid1 == 0) {
        printf("Child 1 PID: %d, Parent PID: %d\n", getpid(), getppid());// Child process 1
    } else {
        pid2 = fork(); // Second fork parent process
        if (pid2 == 0) {
            // Child process 2
            printf("Child 2 PID: %d, Parent PID: %d\n", getpid(), getppid());
        } else {
            // Parent process waits for both children
            wait(NULL);
            wait(NULL);
        }
    }    return 0;   }
```

26. Create two different user-defined functions to generate the following process hierarchy shown in **Figure-(a)** and **Figure-(b)**. Finally all the processes display their process ID and parent ID.

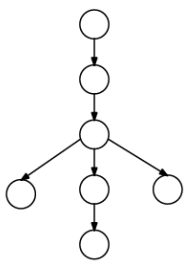


Figure-(a)

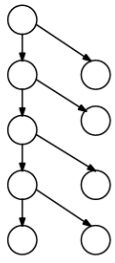


Figure-(b)

```c
// Q26 a)
#include <stdio.h>
#include <unistd.h>

void figureA() {
    if (fork() == 0) {
        printf("Child 1 PID: %d, Parent PID: %d\n", getpid(), getppid());
    } else {
        if (fork() == 0) {
            printf("Child 2 PID: %d, Parent PID: %d\n", getpid(), getppid());
        }
     }
 }
int main() {
    figureA();
    return 0;
}
```

```c
// Q26 b)
#include <stdio.h>
#include <unistd.h>

void figureB() {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("Grandchild PID: %d, Parent PID: %d\n", getpid(), getppid());
        } else {
            printf("Child PID: %d, Parent PID: %d\n", getpid(), getppid());
        }
    }
}
int main() {
    figureB();
    return 0;
}
```

27. What output will be at Line X and Line Y?

```c
#define SIZE 5
int nums[SIZE] = { 0,1,2,3,4 } ;
int main(){
 int i;
 pid t pid;
 pid = fork();
 if(pid == 0){
    for (i = 0; i < SIZE; i++) {
       nums[i] *= nums[i] *-i;
       printf("CHILD:%d ",nums[i]);  /* LINE X */
    }
 }
 else if (pid > 0) {
    wait(NULL);
    for (i = 0; i < SIZE; i++)
       printf("PARENT: %d ",nums[i]);  /* LINE Y */
    }
  return 0;
}
```

```
CHILD: 0
CHILD: -1
CHILD: -8
CHILD: -27
CHILD: -64
PARENT: 0
PARENT: 1
PARENT: 2
PARENT: 3
PARENT: 4
```

```c
/* 28. The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, Write a C program using the fork() system call that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child.*/
 #include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void fibonacci(int n) {
    int a = 0, b = 1, next;
    for (int i = 0; i < n; i++) {
        printf("%d ", a);
        next = a + b;
        a = b;
        b = next;
    }
    printf("\n");
}
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        return 1;
    }
    int n = atoi(argv[1]);
    if (fork() == 0) {
        fibonacci(n);  // Child process generates Fibonacci
    } else {
        wait(NULL);   // Parent process waits
    }
    return 0;
}
```

```c
/*29. Write a MulThree.cprogram to multiply three numbers and display the output. Now write another
 Cprogram PracticeExecl.c, which will fork a child process and the child process will execute the file MulThree.c and generate the output. The parent process will wait till the termination of the child and the parent process will print the process ID and exit status of the child.*/

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

int main(int argc, char *argv[]) {
    if (argc != 4) {
        printf("Usage: %s num1 num2 num3\n", argv[0]);
        return 1;
    }
    int a = atoi(argv[1]);//atoi stands for ASCII To Integer.
    int b = atoi(argv[2]);
    int c = atoi(argv[3]);
    printf("Multiplication: %d\n", a * b * c);
    return 0;
}
```

```c
/* 30. You know the usages of the command grep. Implement the working of grep-n pattern filename in a child process forked from the parent process using execlsystem call. The parent process will wait till the termination of the child and the parent process will print the process ID and exit status of the child.*/
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    if (fork() == 0) {
        execl("/bin/grep", "grep", "-n", "pattern", "filename", NULL);// Child executes grep
    } else {
```

```
        int status;
        wait(&status); //Parent waits for child
        printf("Child exited with status: %d\n", WEXITSTATUS(status));
    }
    return 0;
}
```

```
//31. Implement the question using execv system calls.
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    char *args[] = {"/bin/grep", "-n", "pattern", "filename", NULL};
    if (fork() == 0) {
        execv(args[0], args);
    } else {
        int status;
        wait(&status);
        printf("Child exited with status: %d\n", WEXITSTATUS(status));
    }
    return 0;
}
```

```
// Q30 b) Implement using execlp System call.
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    printf("Using execlp:\n");
    if (fork() == 0) {
        execlp("grep", "grep", "-n", "pattern", "filename", NULL); // Execute grep using execlp
        perror("execlp failed");
    } else {
        wait(NULL);
    }
    return 0;
}
```

```
// Q30 c) Implement using execvp system call.

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

int main() {
    printf("Using execvp:\n");
    char *args[] = {"grep", "-n", "pattern", "filename", NULL};
    if (fork() == 0) {
        execvp(args[0], args); // Execute grep using execvp
        perror("execvp failed");
    } else {
        wait(NULL);
    }
    return 0;
}
```

```c
// Q30 d) Implement using execle system call.

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

int main() {
    printf("Using execle:\n");
    char *envp[] = {NULL}; // No additional environment variables
    if (fork() == 0) {
        execle("/bin/grep", "grep", "-n", "pattern", "filename", NULL, envp); // Execute grep using
execle
        perror("execle failed");
    } else {
        wait(NULL);
    }
    return 0;
}
```

```c
// Q30 e) Implement using execve system call.

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

int main() {
    printf("Using execve:\n");
    char *args[] = {"/bin/grep", "-n", "pattern", "filename", NULL};
    char *envp[] = {NULL}; // No additional environment variables
    if (fork() == 0) {
        execve(args[0], args, envp); // Execute grep using execve
        perror("execve failed");
    } else {
        wait(NULL);
    }
    return 0;
}
```

Explanation:

1. `execv`: Takes the full path of the executable and an array of arguments (`args`).
2. `execlp`: Searches for the executable in `PATH` and takes a list of arguments.
3. `execvp`: Similar to `execlp` but takes arguments as an array (`args`).
4. `execle`: Allows specifying the environment (`envp`) explicitly.
5. `execve`: The most general, requiring both the path, arguments, and environment to be provided explicitly.