OPERATING SYSTEMS LAB - PRACTICAL 4

Name - Sakshi Soni
Roll no - 13

AIM - Demonstrate process control system calls:

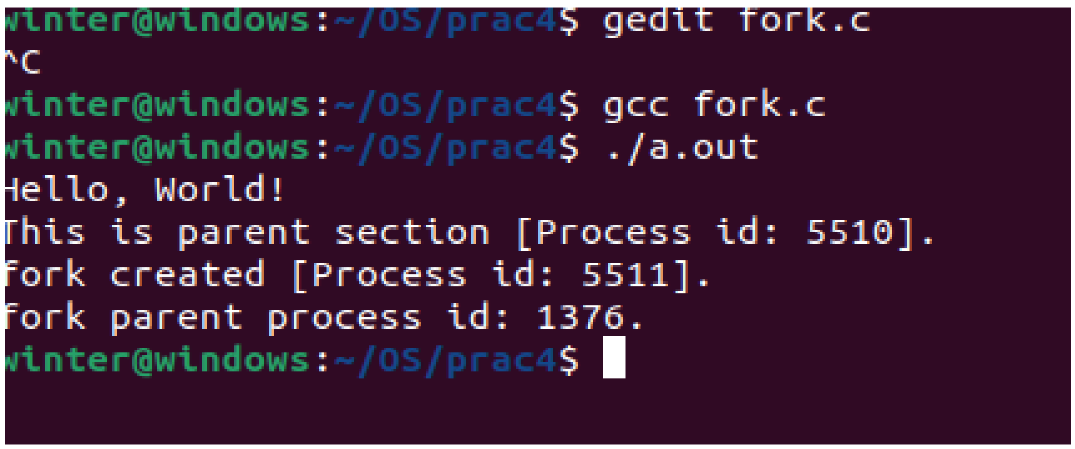
1. Fork
2. Vfork
3. exec
4. Wait and sleep
5. Getpid and getppid
6. Also understand the concept of fork bomb, zombie states, and orphan states.

PROGRAM AND OUTPUT -

Program-1: fork() Example - C program demonstrating use of fork() in Linux

```
#include <stdio.h>
#include <unistd.h>
 int main()
{
   int id;
   printf("Hello, World!\n");
   id=fork();
   if(id>0)
   {
      /*parent process*/
      printf("This is parent section [Process id: %d].\n",getpid());
   }
```

```
    elseif(id==0)
    {
        /*child process*/
        printf("fork created [Process id: %d].\n",getpid());
        printf("fork parent process id: %d.\n",getppid());
    }
    else
    {
        /*fork creation faile*/
        printf("fork creation failed!!!\n");
    }

    return0;
}
```

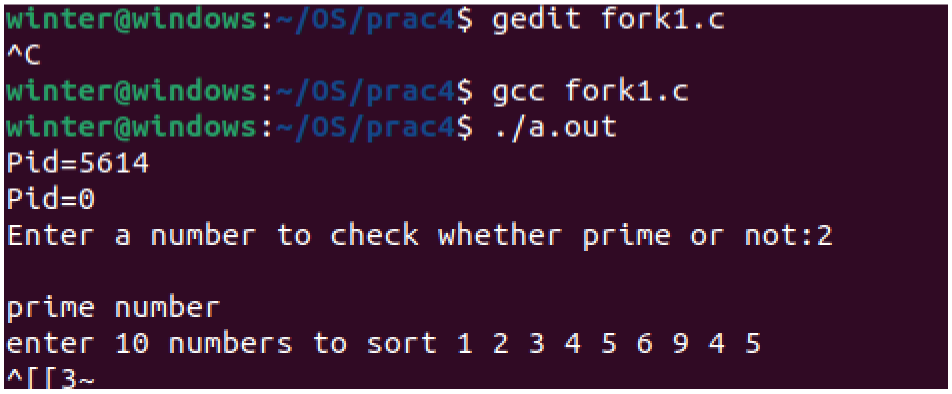


```
winter@windows:~/OS/prac4$ gedit fork.c
^C
winter@windows:~/OS/prac4$ gcc fork.c
winter@windows:~/OS/prac4$ ./a.out
Hello, World!
This is parent section [Process id: 5510].
fork created [Process id: 5511].
fork parent process id: 1376.
winter@windows:~/OS/prac4$
```

Program-2: Program on prime numbers and to print array of numbers using fork...(consider the statement as it can perform two separate tasks from each process)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
```

```
void main()
{
        int pid,n,a[10],i,t,j,flag=0;
        pid=fork();
        printf("Pid=%d\n",pid);
        if(pid==0)
        {
                printf("Enter a number to check whether prime or not:");
                scanf("%d",&n);

                        if(n==1)
                        {
                                printf("number is prime");
                        }

                        Else
                        for(j=2;j<=(n/2);j++)
                        {
                                if(n%j==0)
                                {
                                flag=1;
                                printf("not a prime number");
                                break;
                                }

                        }
                                if(flag==0)
                                printf("\nprime number");
        }
         else
        {
                sleep(5);
                printf("\nenter 10 numbers to sort");
                        for(j=0;j<10;j++)
                        scanf("%d",&a[j]);
                for(i=0;i<9;i++)
                {
                        for(j=0;j<9-i;j++)
                        {
                                if(a[j+1]<a[j])
                                {
                                t=a[j];
                                a[j]=a[j+1];

                                a[j+1]=t;
                                }
                        }
                }
        printf("sorted element:\n");
        for(i=0;i<10;i++)
        printf("%d ",a[i]);
        }
        }
```

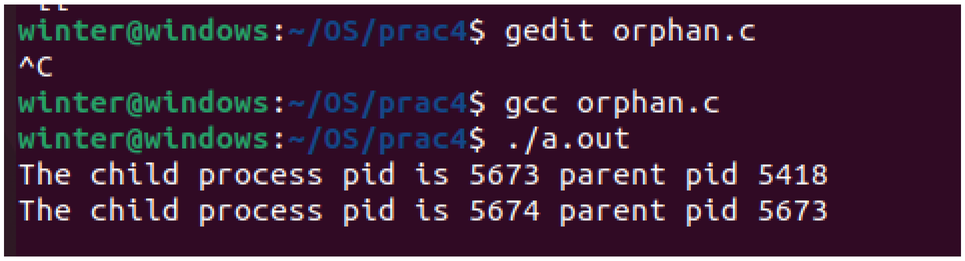


```
winter@windows:~/OS/prac4$ gedit fork1.c
^C
winter@windows:~/OS/prac4$ gcc fork1.c
winter@windows:~/OS/prac4$ ./a.out
Pid=5614
Pid=0
Enter a number to check whether prime or not:2

prime number
enter 10 numbers to sort 1 2 3 4 5 6 9 4 5
^[[3~
```

Program-3 : Program to create Orphan process

/*

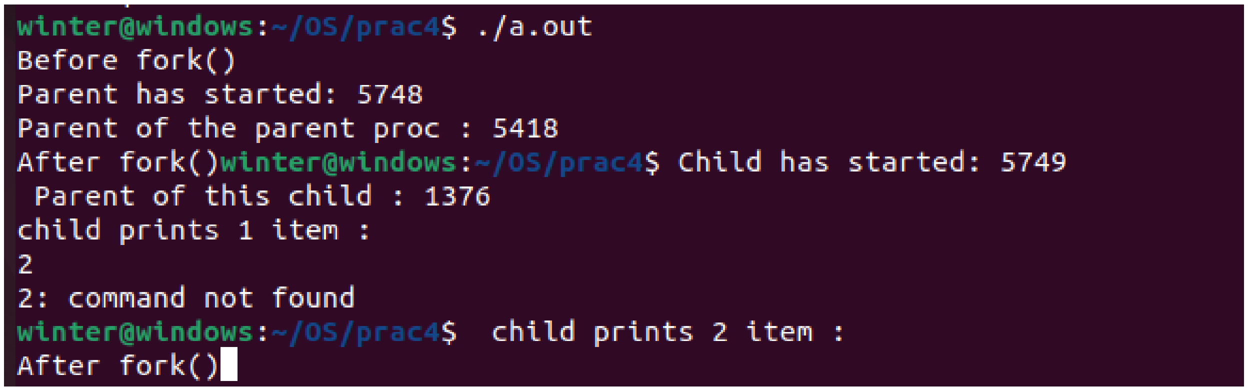
```
 * Program to create orphan process @ Linux
 * getpid() gives process PID and
 * getppid() gives process's parent ID
 * here main() process ID is parent id is current shells PID
 * once process becomes orphan it is adopted by init process(it's PID is 1)
 */

#include<stdio.h>
#include<unistd.h>
int main()
{

 pid_t p;

/* create child process */
 p=fork();

 if(p==0) {
    /* fork() returns Zero to child */
    sleep(10);
 }
 printf("The child process pid is %d parent pid %d\n", getpid(), getppid());
/*parent/child waits for 20 secs and exits*/
 sleep(20);
 printf("\nProcess %d is done its Parent pid %d...\n", getpid(), getppid());

 return 0;
}
```



```
winter@windows:~/OS/prac4$ gedit orphan.c
^C
winter@windows:~/OS/prac4$ gcc orphan.c
winter@windows:~/OS/prac4$ ./a.out
The child process pid is 5673 parent pid 5418
The child process pid is 5674 parent pid 5673
```

Program-4: Code for PROGRAM FOR ORPHAN PROCESS in C Programming

```
#include<stdio.h>

main()
{
   int id;

   printf("Before fork()\n");
   id=fork();

   if(id==0)
   {
      printf("Child has started: %d\n ",getpid());
      printf("Parent of this child : %d\n",getppid());
      printf("child prints 1 item :\n ");
      sleep(10);
      printf("child prints 2 item :\n");
   }
   else
   {
      printf("Parent has started: %d\n",getpid());
      printf("Parent of the parent proc : %d\n",getppid());
   }

   printf("After fork()");
}
```



```
winter@windows:~/OS/prac4$ ./a.out
Before fork()
Parent has started: 5748
Parent of the parent proc : 5418
After fork()winter@windows:~/OS/prac4$ Child has started: 5749
 Parent of this child : 1376
child prints 1 item :
2
2: command not found
winter@windows:~/OS/prac4$  child prints 2 item :
After fork()
```

PROGRAM FOR ZOMBIE PROCESS

Zombie Process:

A process which has finished the execution but still has entry in the process table to report to its parent process is known as a zombie process. A child process always first becomes a zombie before being removed from the process table. The parent process reads the exit status of the child process which reaps off the child process entry from the process table.
In the following code, the child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn't call wait() and the child process's entry still exists in the process table.

Program-5 :

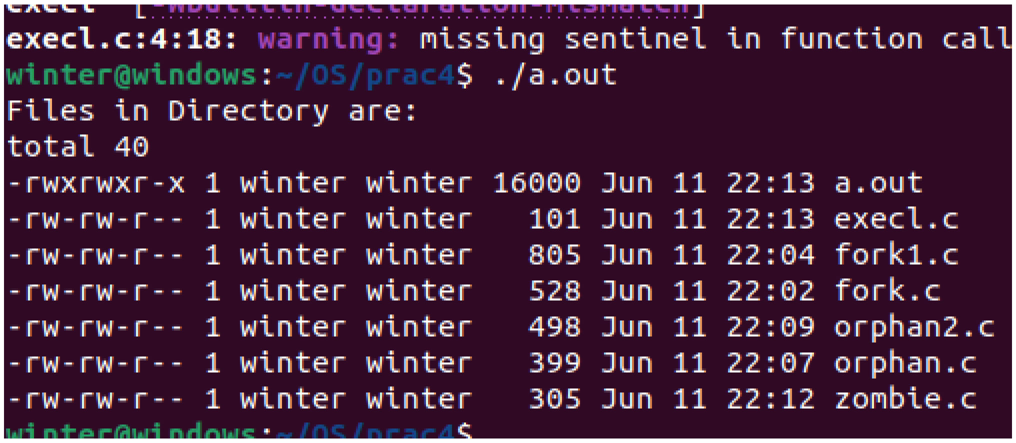
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.

```
#include <stdlib.h>
#include <sys/types.h>
```

```
#include <unistd.h>
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

Program-6 : execl:

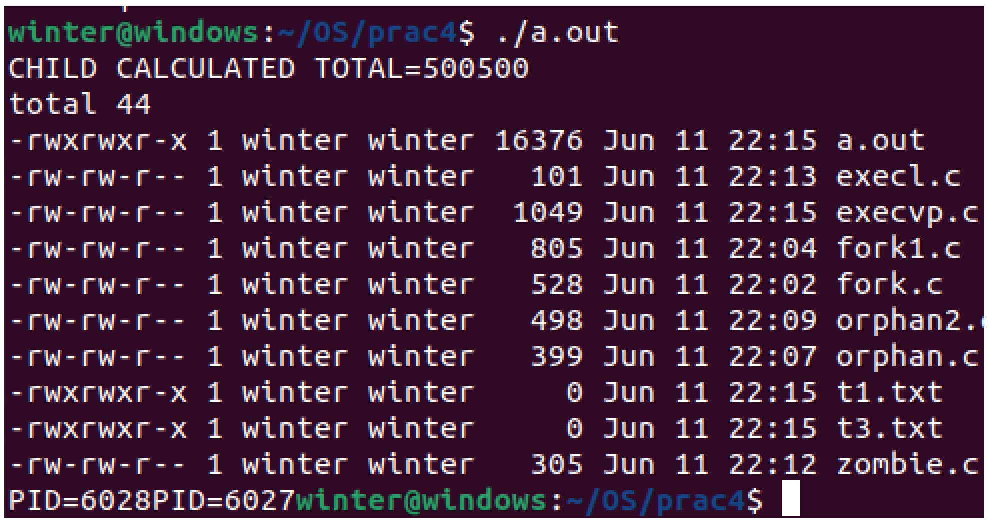
```
main()
{ printf("Files in Directory are:\n");
                    execl("/bin/ls","ls", "-l",0);
}
```



```
execl.c:4:18: warning: missing sentinel in function call
winter@windows:~/OS/prac4$ ./a.out
Files in Directory are:
total 40
-rwxrwxr-x 1 winter winter 16000 Jun 11 22:13 a.out
-rw-rw-r-- 1 winter winter   101 Jun 11 22:13 execl.c
-rw-rw-r-- 1 winter winter   805 Jun 11 22:04 fork1.c
-rw-rw-r-- 1 winter winter   528 Jun 11 22:02 fork.c
-rw-rw-r-- 1 winter winter   498 Jun 11 22:09 orphan2.c
-rw-rw-r-- 1 winter winter   399 Jun 11 22:07 orphan.c
-rw-rw-r-- 1 winter winter   305 Jun 11 22:12 zombie.c
```

Program-7 : execvp:
(Another approach for this code must be to perform CPU bound task from one process and I/O bound task from another process)

```
#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
#define BUF_SIZE 100
void ChildProcess();
void ParentProcess();
void Child1Process();
main(void)
{
        pid_t pid;
        pid=fork();
        if(pid==0)
                ChildProcess();
        else
                ParentProcess();
}

void ChildProcess()
{
        int i;
        char *args[]={"ls","-l",0};
        execvp("ls",args);
}

void ParentProcess()
{
        int i,t;
        char b[BUF_SIZE];
        char buffer[5];
        int f1,f2,ret;
        pid_t pid1;
        pid1=fork();
        if(pid1==0)
                Child1Process();
        else
        {
        f1=open("t1.txt",O_RDONLY|O_CREAT,0777);
        if(f1==-1)
                write(1,"Error while opening file!!!!!!",36);
        f2=open("t3.txt",O_WRONLY|O_CREAT,0777);
        if(f2==-1)
                write(1,"Error while opening file!!!!!!",36);
        else
        do{
                ret=read(f1,buffer,sizeof(buffer));
                if(ret)
                {
                        ret=write(f2,buffer,sizeof(buffer));
                }
        }       while(ret);
        }
        i=wait(NULL);
        if(i!=-1)
                printf("PID=%d",i);
        i=wait(NULL);
        if(i!=-1)
                printf("PID=%d",i);
}

void Child1Process()
{
        int i;
        long int sum=0;
        for(i=1;i<=1000;i++)
                sum+=i;
        printf("CHILD CALCULATED TOTAL=%ld\n",sum);

}
```



```
winter@windows:~/OS/prac4$ ./a.out
CHILD CALCULATED TOTAL=500500
total 44
-rwxrwxr-x 1 winter winter 16376 Jun 11 22:15 a.out
-rw-rw-r-- 1 winter winter   101 Jun 11 22:13 execl.c
-rw-rw-r-- 1 winter winter  1049 Jun 11 22:15 execvp.c
-rw-rw-r-- 1 winter winter   805 Jun 11 22:04 fork1.c
-rw-rw-r-- 1 winter winter   528 Jun 11 22:02 fork.c
-rw-rw-r-- 1 winter winter   498 Jun 11 22:09 orphan2.
-rw-rw-r-- 1 winter winter   399 Jun 11 22:07 orphan.c
-rwxrwxr-x 1 winter winter     0 Jun 11 22:15 t1.txt
-rwxrwxr-x 1 winter winter     0 Jun 11 22:15 t3.txt
-rw-rw-r-- 1 winter winter   305 Jun 11 22:12 zombie.c
PID=6028PID=6027winter@windows:~/OS/prac4$
```

Program-8 : Programs to estimate the working of fork() and vfork():

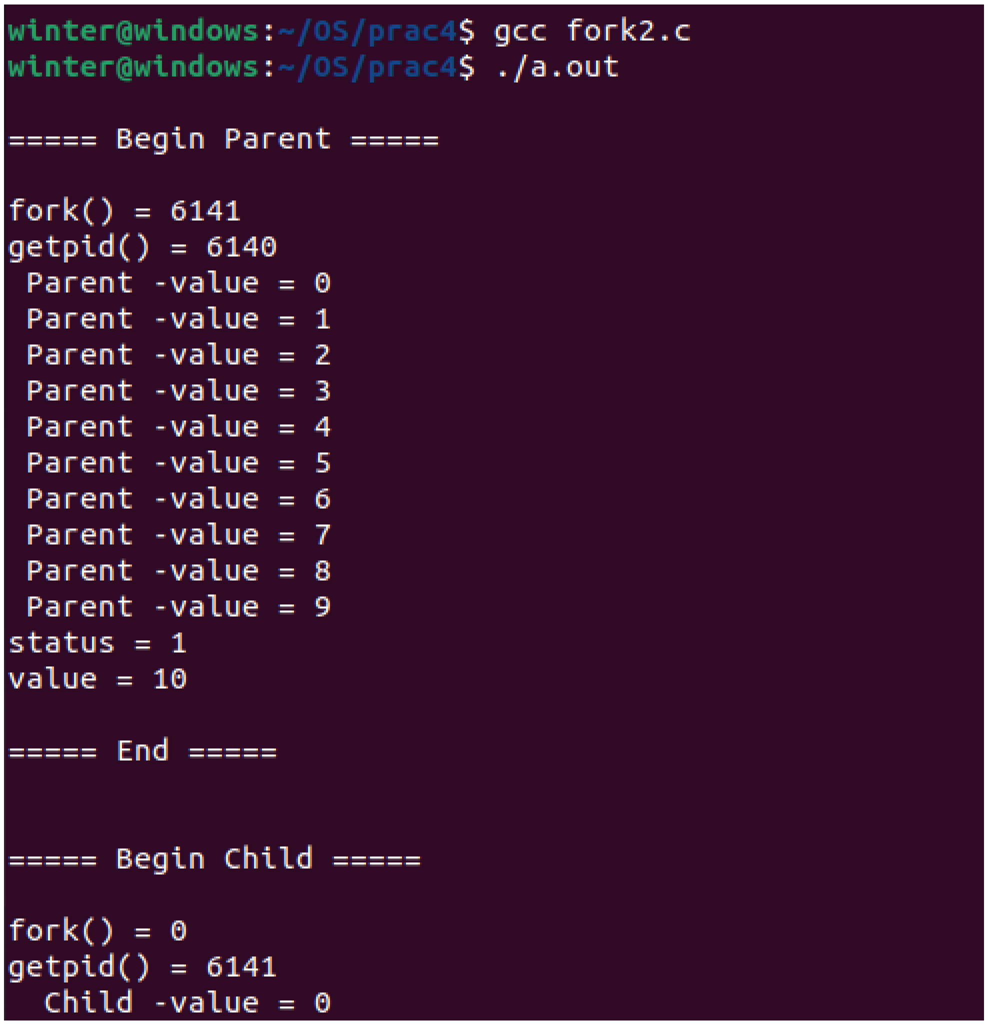
Using the fork() syscall

main.c

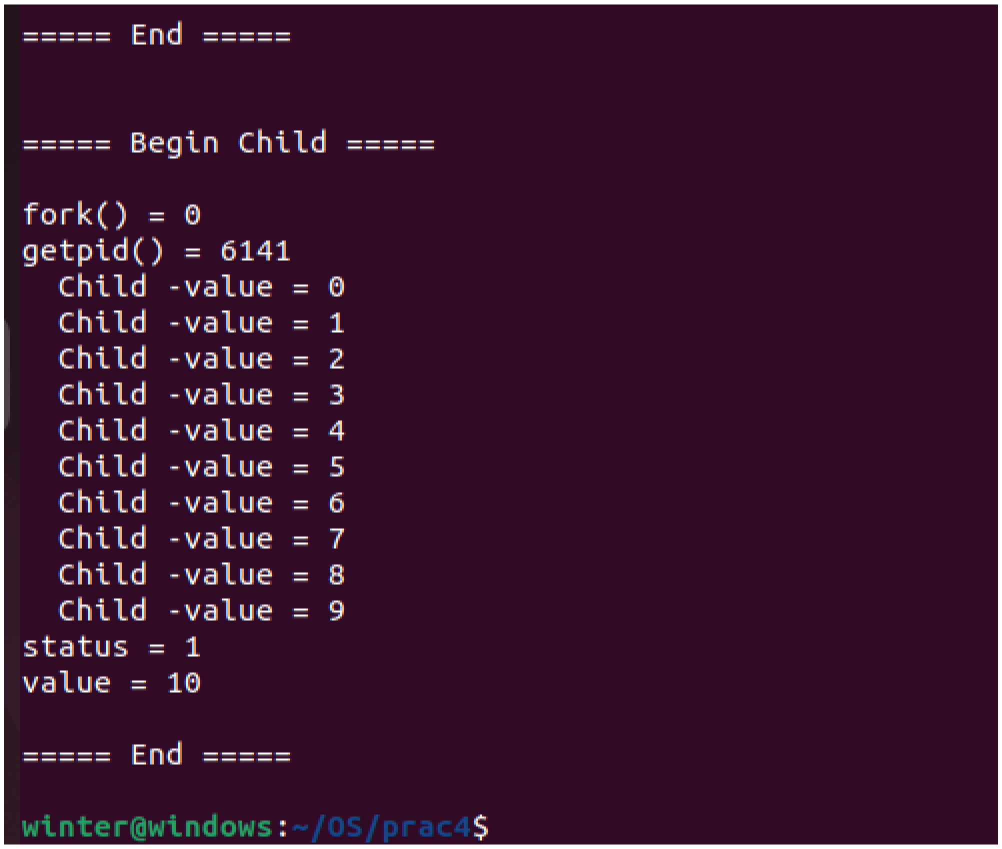
```
#include        <stdio.h>
```

```
#include       <stdlib.h>
#include       <sys/types.h>
#include       <sys/wait.h>
#include       <unistd.h>
#include       <string.h>
#include       <errno.h>
int     main()
{
  int i, value;
  int status;
  pid_t f;

  value = 0;
  i = 0;
  status = 1;
  f = fork();
  if (f < 0)
    {
      fprintf(stderr, "Error: %s - fork() < 0 (%d)\n", strerror(errno), f);
    }
  else if (f > 0)
    {
      printf("\n===== Begin Parent =====\n\n");
      printf("fork() = %d\n", f);
      printf("getpid() = %d\n", getpid());
      while (i < 10)
      {
          printf(" Parent -value = %d\n", value);
          ++value;
          ++i;
      }
    }
  else
```

```
    {
      printf("\n===== Begin Child =====\n\n");
      printf("fork() = %d\n", f);
      printf("getpid() = %d\n", getpid());
      while (i < 10)
      {
          printf("  Child -value = %d\n", value);
          ++value;
          ++i;
      }
    }
  printf("status = %d\n", status);
  printf("value = %d\n\n", value);
  printf("===== End =====\n\n");
  return 0;
}
```



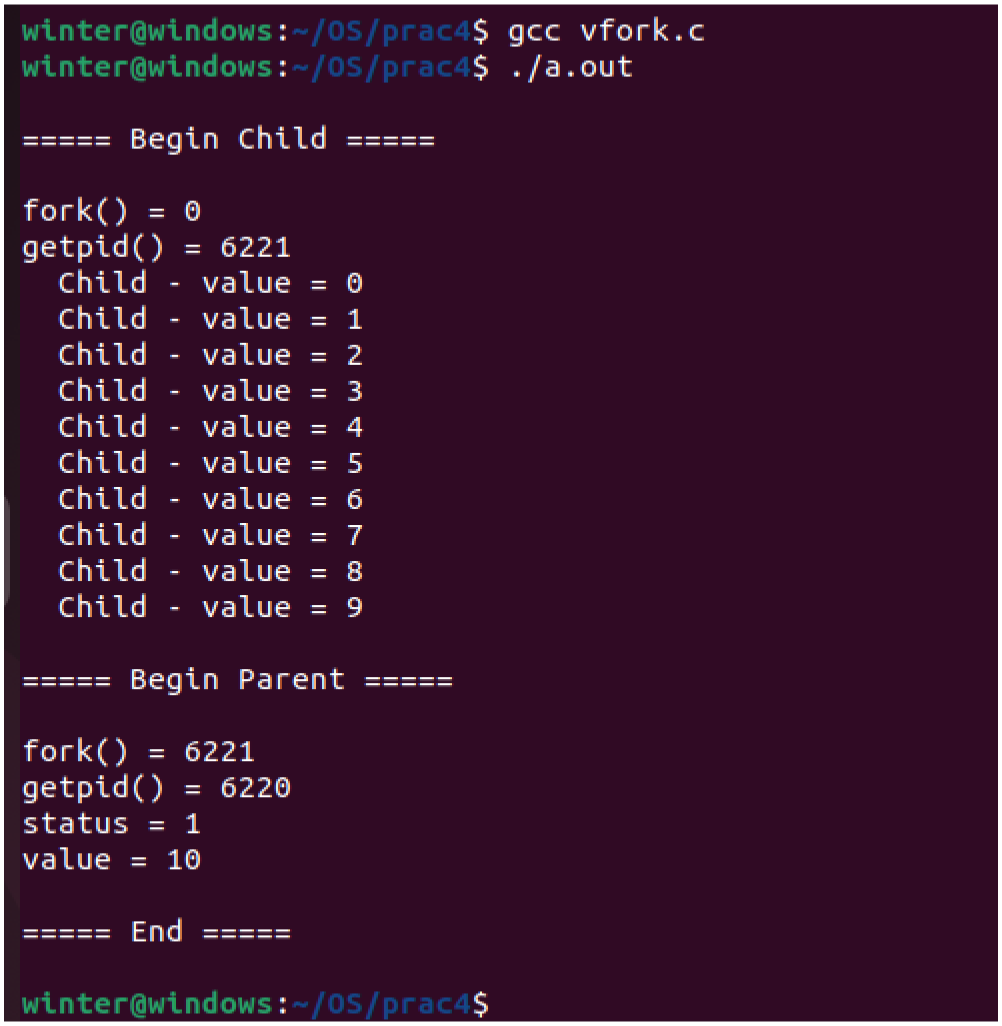
```
winter@windows:~/OS/prac4$ gcc fork2.c
winter@windows:~/OS/prac4$ ./a.out

===== Begin Parent =====

fork() = 6141
getpid() = 6140
 Parent -value = 0
 Parent -value = 1
 Parent -value = 2
 Parent -value = 3
 Parent -value = 4
 Parent -value = 5
 Parent -value = 6
 Parent -value = 7
 Parent -value = 8
 Parent -value = 9
status = 1
value = 10

===== End =====


===== Begin Child =====

fork() = 0
getpid() = 6141
  Child -value = 0
```


```
===== End =====


===== Begin Child =====

fork() = 0
getpid() = 6141
  Child -value = 0
  Child -value = 1
  Child -value = 2
  Child -value = 3
  Child -value = 4
  Child -value = 5
  Child -value = 6
  Child -value = 7
  Child -value = 8
  Child -value = 9
status = 1
value = 10

===== End =====

winter@windows:~/OS/prac4$
```

Program-9: Using the vfork() syscall

```
#include      <stdio.h>
#include      <stdlib.h>
#include      <sys/types.h>
#include      <sys/wait.h>
#include      <unistd.h>
#include      <string.h>
#include      <errno.h>

int    main()
{
int i,value;
int status;
pid_t f;
```

```
value = 0;
  i = 0;
  status = 1;
  f = vfork();
  if (f < 0)
    {
fprintf(stderr, "Error: %s - fork() < 0 (%d)\n", strerror(errno), f);
    }
  else if (f > 0)
    {
printf("\n===== Begin Parent =====\n\n");
printf("fork() = %d\n", f);
printf("getpid() = %d\n", getpid());
      while (i < 10)
      {
printf(" Parent - value = %d\n", value);
          ++value;
          ++i;
      }
    }
  else
    {
printf("\n===== Begin Child =====\n\n");
printf("fork() = %d\n", f);
printf("getpid() = %d\n", getpid());
      while (i < 10)
      {
printf("  Child - value = %d\n", value);
          ++value;
          ++i;
      }
      _exit(status);
  }
printf("status = %d\n", status);
printf("value = %d\n\n", value);
printf("===== End =====\n\n");
 return 0;
}
```



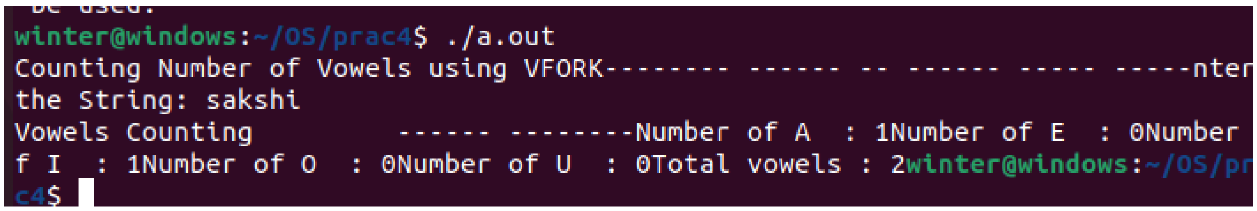
```
winter@windows:~/OS/prac4$ gcc vfork.c
winter@windows:~/OS/prac4$ ./a.out

===== Begin Child =====

fork() = 0
getpid() = 6221
  Child - value = 0
  Child - value = 1
  Child - value = 2
  Child - value = 3
  Child - value = 4
  Child - value = 5
  Child - value = 6
  Child - value = 7
  Child - value = 8
  Child - value = 9

===== Begin Parent =====

fork() = 6221
getpid() = 6220
status = 1
value = 10

===== End =====

winter@windows:~/OS/prac4$
```

Program-10 :   C Program for vowels counting using vfork

```
#include<stdio.h>
#include<sys/types.h>
int main()
```

```
{
intj,n,a,i,e,o,u;
        char str[50];
        a=e=i=o=u=0;
pid_tpid;
        if((pid=vfork())<0)
         {
perror("FORK ERROR");
                  exit(1);
         }
         if(pid==0)
         {
printf("
                        Counting Number of Vowels using VFORK");
printf("
                        -------- ------ -- ------ ----- -----");
printf("
Enter the String:");
                  gets(str);
                  _exit(1);
         }
         else
         {
                  n=strlen(str);
                  for(j=0;j<n;j++)
                  {
                           if(str[j]=='a' || str[j]=='A')
                                    a++;
                           else if(str[j]=='e' || str[j]=='E')
                                    e++;
                           else if(str[j]=='i' || str[j]=='I')
                                    i++;
                           else if(str[j]=='o' || str[j]=='O')
```

```
                                o++;
                    else if(str[j]=='u' || str[j]=='U')
                        u++;
                }
printf("
                Vowels Counting");
printf("
                ------ --------");
printf("
Number of A  : %d",a);
printf("
Number of E  : %d",e);
printf("
Number of I  : %d",i);
printf("
Number of O  : %d",o);
printf("
Number of U  : %d",u);
printf("
Total vowels : %d
",a+e+i+o+u);
            exit(1);
        }
}
```



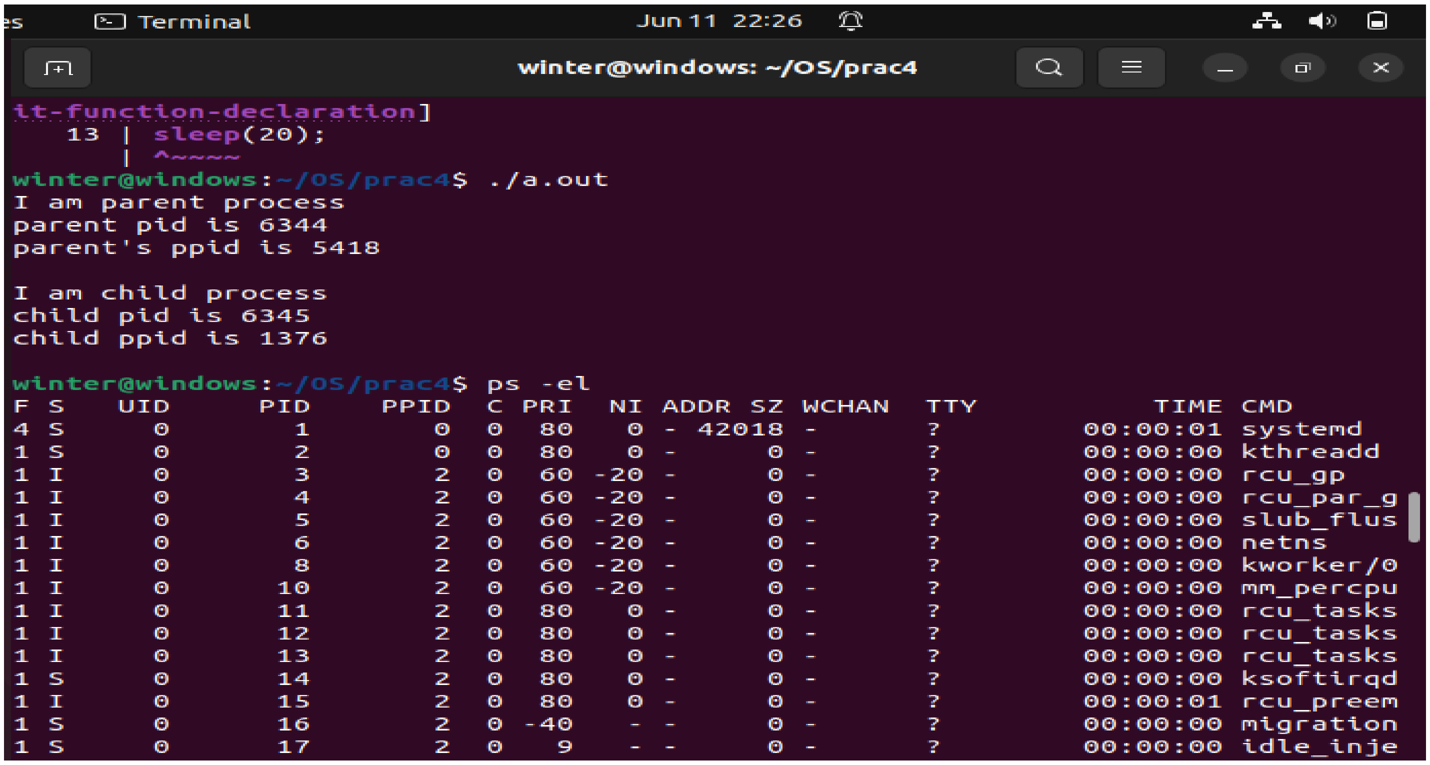
```
 be used.
winter@windows:~/OS/prac4$ ./a.out
Counting Number of Vowels using VFORK-------- ------ -- ------ ----- -----nter
the String: sakshi
Vowels Counting          ------ --------Number of A  : 1Number of E  : 0Number
f I  : 1Number of O  : 0Number of U  : 0Total vowels : 2winter@windows:~/OS/pr
c4$ 
```

Program-11 : C program for Fork Bomb

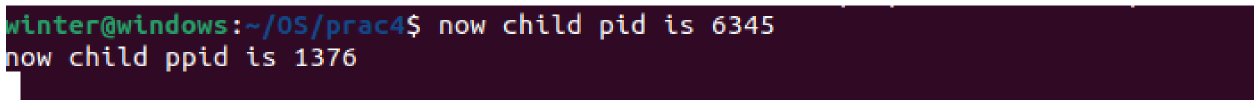
```
// C program Sample  for FORK BOMB
// It is not recommended to run the program as
// it may make a system non-responsive.
#include <stdio.h>
#include <sys/types.h>

int main()
{
   while(1)
      fork();
   return 0;
}
```

PROGRAM - 11 :
Part 1: demonstrate orphan child process
//In this program fork pid, ppid will be demonstrated
// Initially child will have one parent,after the death of its father it will be assigned
another father
// named init process. This can be checked by noting down the new ppid and running ps-
el. run the program in background by concanating & at the end.

```
#include <stdio.h>
void main()
{
int pid1,pid,ppid;
pid1 =fork();
if(pid1==0)
{
printf("I am child process \n");
printf("child pid is %d\n",getpid());
printf("child ppid is %d\n",getppid());
printf("\n");
//system("ps -el");
sleep(20);
printf("now child pid is %d\n",getpid());
printf("now child ppid is %d\n",getppid());
//system("ps -el");
}

if(pid1>0)
{
printf("I am parent process \n");
printf("parent pid is %d\n",getpid());
printf("parent's ppid is %d\n",getppid());
printf("\n");
}
}
```



```
it-function-declaration]
   13 |  sleep(20);
      |  ^~~~~
winter@windows:~/OS/prac4$ ./a.out
I am parent process
parent pid is 6344
parent's ppid is 5418

I am child process
child pid is 6345
child ppid is 1376

winter@windows:~/OS/prac4$ ps -el
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0       1       0  0  80   0 - 42018 -      ?        00:00:01 systemd
1 S     0       2       0  0  80   0 -     0 -      ?        00:00:00 kthreadd
1 I     0       3       2  0  60 -20 -     0 -      ?        00:00:00 rcu_gp
1 I     0       4       2  0  60 -20 -     0 -      ?        00:00:00 rcu_par_g
1 I     0       5       2  0  60 -20 -     0 -      ?        00:00:00 slub_flus
1 I     0       6       2  0  60 -20 -     0 -      ?        00:00:00 netns
1 I     0       8       2  0  60 -20 -     0 -      ?        00:00:00 kworker/0
1 I     0      10       2  0  60 -20 -     0 -      ?        00:00:00 mm_percpu
1 I     0      11       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks
1 I     0      12       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks
1 I     0      13       2  0  80   0 -     0 -      ?        00:00:00 rcu_tasks
1 S     0      14       2  0  80   0 -     0 -      ?        00:00:00 ksoftirqd
1 I     0      15       2  0  80   0 -     0 -      ?        00:00:01 rcu_preem
1 S     0      16       2  0 -40   - -     0 -      ?        00:00:00 migration
1 S     0      17       2  0   9   - -     0 -      ?        00:00:00 idle_inje
```



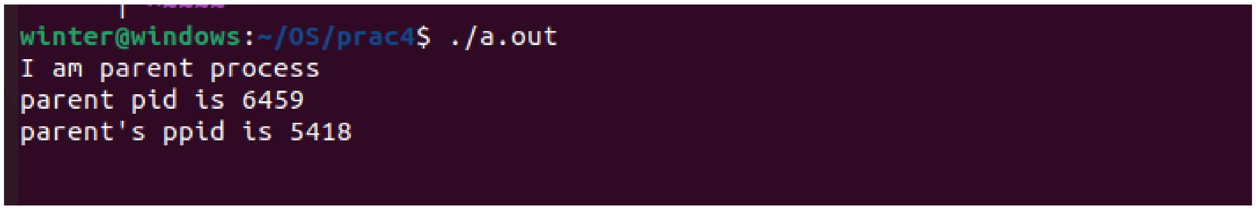
```
winter@windows:~/OS/prac4$ now child pid is 6345
now child ppid is 1376
```

Part 2: Demonstrate Zombie process:

//In this program Zombi Process will be demonstrated
// Run this program in background and run ps-el on the command prompt and see that
it is zombi Z process

```
#include <stdio.h>
void main()
{
int pid1,pid,ppid;
pid1 =fork();

if(pid1>0)
{
```

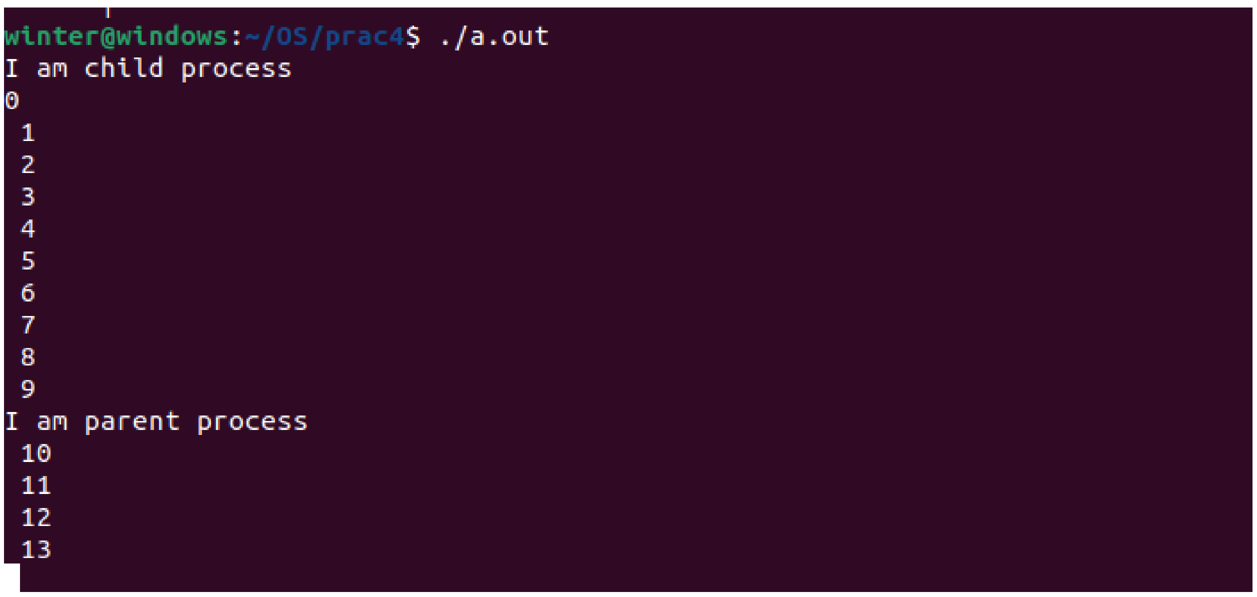
```
printf("I am parent process \n");
printf("parent pid is %d\n",getpid());
printf("parent's ppid is %d\n",getppid());
printf("\n");
sleep(10);
}
}
```



```
winter@windows:~/OS/prac4$ ./a.out
I am parent process
parent pid is 6459
parent's ppid is 5418
```

// We will demonstrate sleep and wait system calls in this programe

```
#include <stdio.h>
void main()
{
int pid1,i;
pid1 =fork();
if(pid1==0)
{
printf("I am child process \n");
  for(i=0;i<30;i++)
     {
      printf("%d\n ", i);
      sleep(1);
//system("ps -el");
     }
//exit(0);
}

else
{
//wait(0);
sleep(10);
printf("I am parent process\n ");
wait(0);
//sleep(10);
printf("I am parent process\n ");
//wait(0);
}
}
```

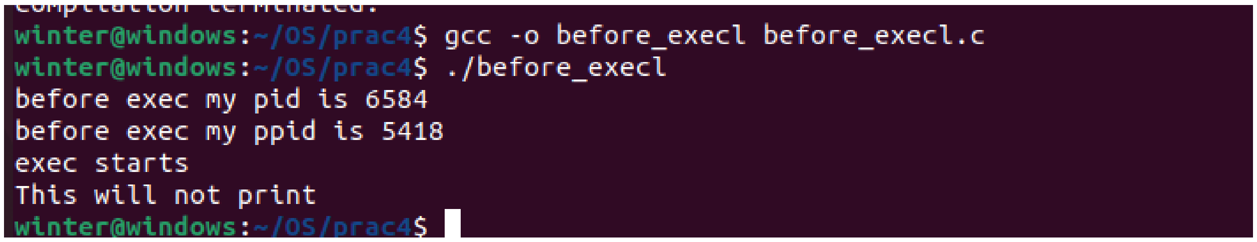


```
winter@windows:~/OS/prac4$ ./a.out
I am child process
0
 1
 2
 3
 4
 5
 6
 7
 8
 9
I am parent process
 10
 11
 12
 13
```

// We will demonstrate execl system calls in this programe
// Compile this program by gcc -o first_d first_d.c
// Compile next program by gcc -o first_e first_e.c

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

void main()
{
```

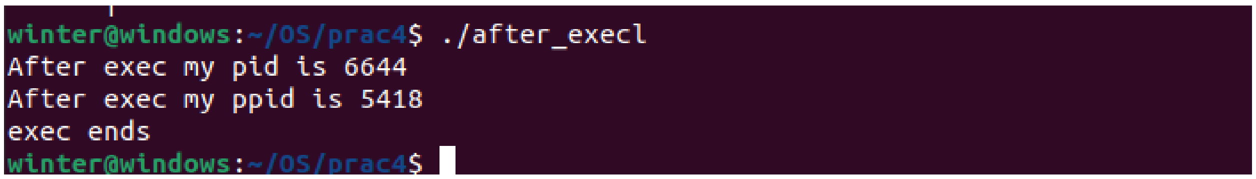
```
printf("before exec my pid is %d\n",getpid());
printf("before exec my ppid is %d\n",getppid());
printf("exec starts\n");
execl("first_e","first_e",(char*)0);
printf("This will not print\n");
}
```



```
winter@windows:~/OS/prac4$ gcc -o before_execl before_execl.c
winter@windows:~/OS/prac4$ ./before_execl
before exec my pid is 6584
before exec my ppid is 5418
exec starts
This will not print
winter@windows:~/OS/prac4$
```

// We will demonstrate execl system calls in this programe
// Compile this program by gcc -o first_d first_d.c
// Compile next program by gcc -o first_e first_e.c

```
#include <stdio.h>
void main()
{
printf("After exec my pid is %d\n",getpid());
printf("After exec my ppid is %d\n",getppid());
printf("exec ends\n");
}
```



```
winter@windows:~/OS/prac4$ ./after_execl
After exec my pid is 6644
After exec my ppid is 5418
exec ends
winter@windows:~/OS/prac4$
```

In this, you work with the fork(), wait() and the exec*() family of functions in order to find the maximum in an array of integers.
Part 1
Write a C program parmax.c that creates a tree of processes in order to recursively compute the maximum in an array of integers. The process at

the root of the tree reads the count n of integers in the array. An array A of size n is then populated with randomly generated integers of small values (in the range 0–127). The initially unsorted array is printed by the root process. Any process in the tree handles a chunk of the array A. The chunk is delimited by two indices L and R. For the root process, L = 0 and R = n – 1. Any process P in the tree (including the root) first counts the number of integers in the chunk it has got. If that count is less than 10, the process P itself computes the maximum element in its chunk, prints it, and exits. If the chunk size of P is 10 or more, then P creates two child processes PL and PR which handle the chunks [L, M] and [M + 1, R] in A respectively, where M = (L + R) / 2. P waits until the two child processes PL and PR exit. It then computes the maximum of the two maximum values computed by PL and PR, prints this maximum, and exits.
Every non-root process returns to its parent (via the exit status) the maximum value for its chunk. During the printing of the maximum computed by a process P, the PID and the parent PID of P are also printed. For n = 50, the ranges of the chunks handled by different processes in the tree are shown below.

It is expected that your code will handle values of n in the range 50 – 100. Compile your code, and generate an executable file with the name parmax.

Part 2
Write a separate C code wrapper.c to run the executable parmax created in Part 1. When parmax exits, your wrapper function should also exit.
Submit the two C source files parmax.c and wrapper.c.

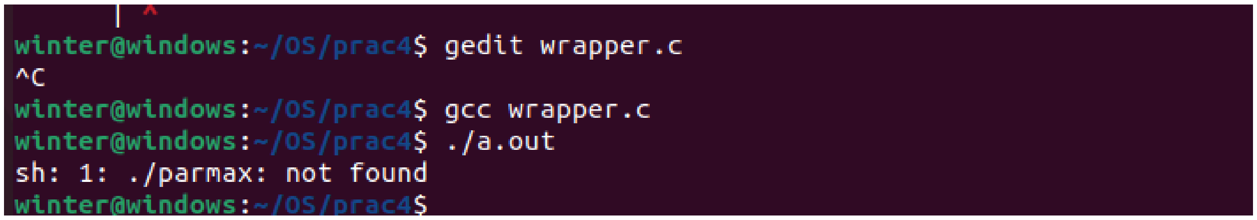
Parmax.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h> #include <sys/wait.h> int findmax(int *A, int L, int R)
```

```
{
int max = A[L];
for (int i = L + 1; i <= R; i++)
{
if (A[i] > max)
{
max = A[i];
}
}
return max;
}
void process(int *A, int L, int R)
{
if (R - L + 1 < 10)
{
int max = findmax(A, L, R); printf("Process Id : %d (Parent Process Id : %d) -
Max: %d\n", getpid(), getppid(), max); exit(max);
}
else
{
int M = (L + R) / 2;
int left_child, right_child; int left_status, right_status; left_child = fork(); if
(left_child == 0)
{
process(A, L, M);
}
else
{
right_child = fork(); if (right_child == 0)
{
process(A, M + 1, R);
}
else
{
waitpid(left_child, &left_status, 0); waitpid(right_child, &right_status, 0); int
max_l = WEXITSTATUS(left_status); int max_r =
WEXITSTATUS(right_status); int max; if (max_l > max_r) max = max_l;
else printf("Process Id : %d (Parent Process Id : %d) - Max: %d\n", getpid(),
getppid(), max); exit(max);
}
}
}
}
void main()
{
int n = 50;
int A[n]; printf("Array is : \n"); for (int i = 0; i < n; i++)
{
A[i] = rand() % 128;
printf("%d ", A[i]);
} printf("\n"); process(A, 0, n - 1);
}
```

Wrapper.c:

```
#include <stdlib.h> #include <stdio.h> int main()
{
system("./parmax"); return 0;
}
```

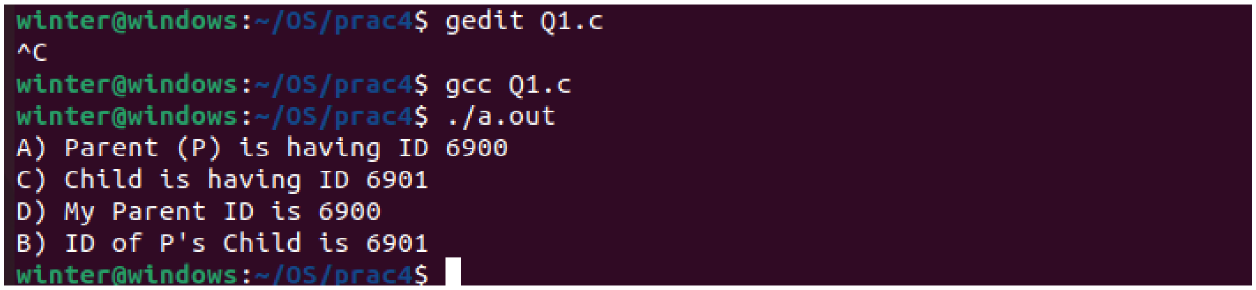


```
winter@windows:~/OS/prac4$ gedit wrapper.c
^C
winter@windows:~/OS/prac4$ gcc wrapper.c
winter@windows:~/OS/prac4$ ./a.out
sh: 1: ./parmax: not found
winter@windows:~/OS/prac4$
```

Demonstrate the following questions using programs:

Q1. Create a parent-child relationship between two processes. The parent should print two statements:
A) Parent (P) is having ID <PID>
B) ID of P's Child is <PID_of_Child>
The child should print two statements:
C) Child is having ID <PID>
D) My Parent ID is <PID_of_Parent>
Make use of wait() in such a manner that the order of the four statements A, B, C and D is:
A
C
D
B
You are free to use any other relevant statement/printf as you desire and their order of execution does not matter.

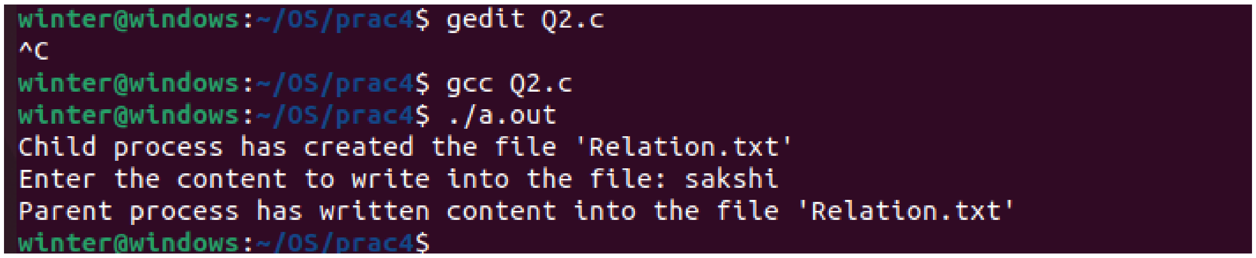
```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t child_pid;

    printf("A) Parent (P) is having ID %d\n", getpid());

    child_pid = fork();

    if (child_pid == 0) {
        printf("C) Child is having ID %d\n", getpid());
        printf("D) My Parent ID is %d\n", getppid());
    } else {
        wait(NULL);
        printf("B) ID of P's Child is %d\n", child_pid);
    }

    return 0;
}
```



```
winter@windows:~/OS/prac4$ gedit Q1.c
^C
winter@windows:~/OS/prac4$ gcc Q1.c
winter@windows:~/OS/prac4$ ./a.out
A) Parent (P) is having ID 6900
C) Child is having ID 6901
D) My Parent ID is 6900
B) ID of P's Child is 6901
winter@windows:~/OS/prac4$
```

Q2. Create a parent-child relationship between two processes such that the Child process creates a file named Relation.txt and the Parent process write some content into it by taking the input from the user

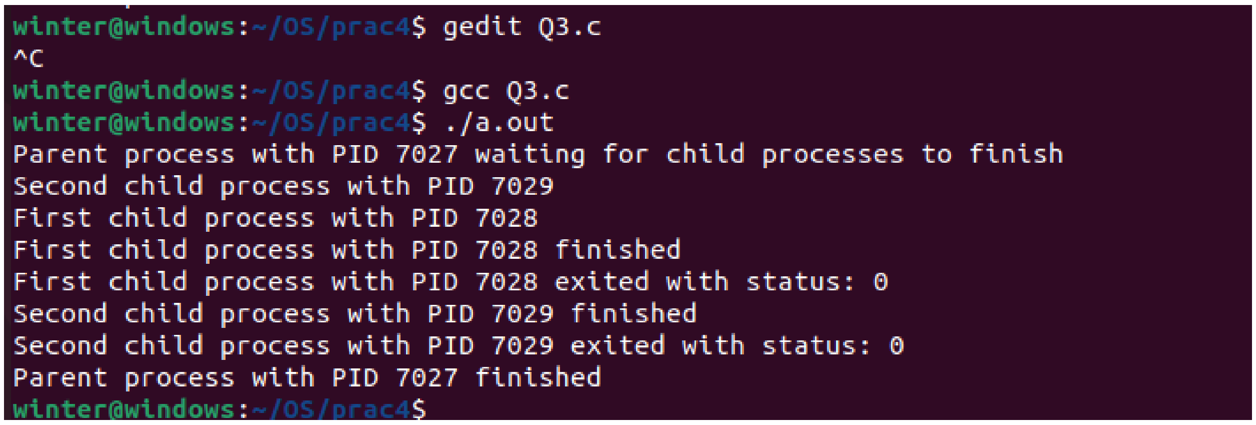
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t child_pid;

    child_pid = fork();

    if (child_pid == 0) {
        // Child process
        FILE *file = fopen("Relation.txt", "w");
        if (file == NULL) {
            perror("Error opening file");
            exit(1);
        }
        fprintf(file, "This is the content written by the child process.\n");
        fclose(file);
        printf("Child process has created the file 'Relation.txt'\n");
    } else {
        // Parent process
        wait(NULL);

        char content[100];
        printf("Enter the content to write into the file: ");
        fgets(content, sizeof(content), stdin);

        FILE *file = fopen("Relation.txt", "a");
        if (file == NULL) {
            perror("Error opening file");
            exit(1);
        }
        fprintf(file, "Content written by the parent process: %s", content);
        fclose(file);
        printf("Parent process has written content into the file 'Relation.txt'\n");
    }

    return 0;
}
```



```
winter@windows:~/OS/prac4$ gedit Q2.c
^C
winter@windows:~/OS/prac4$ gcc Q2.c
winter@windows:~/OS/prac4$ ./a.out
Child process has created the file 'Relation.txt'
Enter the content to write into the file: sakshi
Parent process has written content into the file 'Relation.txt'
winter@windows:~/OS/prac4$
```

Q3. Write a program to create two child process. The parent process should wait for both the child to finish.

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

int main() {
    pid_t child_pid1, child_pid2;
    int status1, status2;

    child_pid1 = fork();

    if (child_pid1 == 0) {
        // First child process
        printf("First child process with PID %d\n", getpid());
        sleep(2);
        printf("First child process with PID %d finished\n", getpid());
        exit(0);
    }

    child_pid2 = fork();

    if (child_pid2 == 0) {
        // Second child process
        printf("Second child process with PID %d\n", getpid());
        sleep(4);
        printf("Second child process with PID %d finished\n", getpid());
        exit(0);
    }

    // Parent process
    printf("Parent process with PID %d waiting for child processes to
finish\n", getpid());

    waitpid(child_pid1, &status1, 0);
    printf("First child process with PID %d exited with status: %d\n",
child_pid1, WEXITSTATUS(status1));

    waitpid(child_pid2, &status2, 0);
    printf("Second child process with PID %d exited with status: %d\n",
child_pid2, WEXITSTATUS(status2));

    printf("Parent process with PID %d finished\n", getpid());

    return 0;
}
```

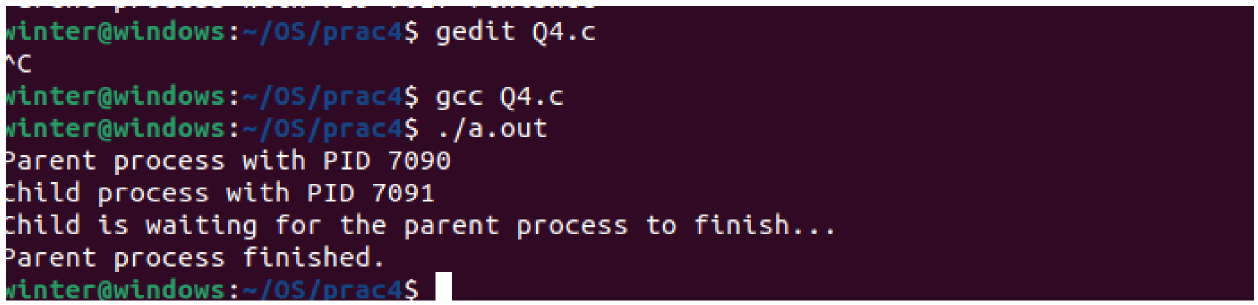


```
winter@windows:~/OS/prac4$ gedit Q3.c
^C
winter@windows:~/OS/prac4$ gcc Q3.c
winter@windows:~/OS/prac4$ ./a.out
Parent process with PID 7027 waiting for child processes to finish
Second child process with PID 7029
First child process with PID 7028
First child process with PID 7028 finished
First child process with PID 7028 exited with status: 0
Second child process with PID 7029 finished
Second child process with PID 7029 exited with status: 0
Parent process with PID 7027 finished
winter@windows:~/OS/prac4$
```

Q4. Can we use wait() to make the child process wait for the parent process to finish?

No, the wait() function is used by the parent process to wait for the child processes to finish. The purpose of wait() is to allow the parent process to synchronize its execution with the termination of its child processes.
In the case of a child process waiting for the parent process to finish, you can achieve this by using the getppid() function to obtain the parent's process ID and then calling waitpid() in a loop until the parent process is no longer running.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t child_pid;
    child_pid = fork();
    if (child_pid == 0) {
        // Child process
        printf("Child process with PID %d\n", getpid());
        printf("Child is waiting for the parent process to finish...\n");
        while (getppid() != 1) {
            sleep(1);
        }
        printf("Parent process has finished. Child process exiting.\n");
        exit(0);
    } else {
        // Parent process
        printf("Parent process with PID %d\n", getpid());
        sleep(5);
        printf("Parent process finished.\n");
    }
    return 0;
}
```



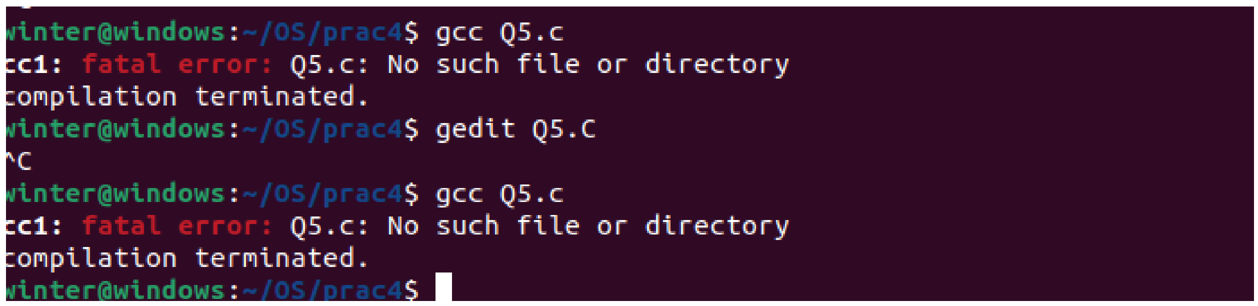
```
winter@windows:~/OS/prac4$ gedit Q4.c
^C
winter@windows:~/OS/prac4$ gcc Q4.c
winter@windows:~/OS/prac4$ ./a.out
Parent process with PID 7090
Child process with PID 7091
Child is waiting for the parent process to finish...
Parent process finished.
winter@windows:~/OS/prac4$ 
```

Q5. What does the wait() system call return on success?

The wait() system call returns the process ID (PID) of the terminated child process on success. It allows the parent process to wait for the termination of its child processes and retrieve information about the terminated child process. The wait() system call takes a pointer to an integer variable status as an argument. This variable is used to store the termination status of the child process. If the wait() system call is successful, it returns the PID of the terminated child process.

```
#include <sys/types.h>
```

```
#include <sys/wait.h>
pid_t wait(int *status);
```



```
winter@windows:~/OS/prac4$ gcc Q5.c
cc1: fatal error: Q5.c: No such file or directory
compilation terminated.
winter@windows:~/OS/prac4$ gedit Q5.C
^C
winter@windows:~/OS/prac4$ gcc Q5.c
cc1: fatal error: Q5.c: No such file or directory
compilation terminated.
winter@windows:~/OS/prac4$ 
```

RESULT - Process control system calls have been studied and Linux C programs on them have been implemented.