Operating Systems

Digital Assignment - I

Aadhitya Swarnesh I
[redacted]

Q1)

Code :

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

int main()
{
   pid_t pid1, pid2, pid3, pid4;

   pid1 = fork();
   pid2 = fork();

   if(pid2 > 0)
   {
      pid3 = fork();
      if(pid3 > 0)
      {
         pid4 = fork();
      }
      else if(pid3 == 0)
      {
         fork();
         fork();
      }
   }
   else if(pid2 == 0)
   {
      fork();
      fork();
   }
```

```
    sleep(5);
    printf("Hi\n");
}
```

Explanation :

The fork() function is used to create a new child thread. The child thread would have a fork value of zero and the parent would have a positive fork value. We use this principle to design as per the specification stated using conditional statements to check for parent and child thread. The order in which the statements are executed would vary depending on the environment used but the end output would be the same.

Q2)

Code :

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

int main()
{
    int arr[] = {5, 2, 1, 3, 4};

    pid_t pid;
    pid = fork();

    if(pid>0)
    {
        printf("Parent Process :\nSorted numbers are :\n");
        for(int i=0;i<5;i++)
        {
            for(int j=i;j<5;j++)
            {
                if(arr[i]>arr[j])
                {
                    int a = arr[i];
                    arr[i]=arr[j];
                    arr[j]=a;
```

```
            }
        }
    }
    for(int i=0;i<5;i++)
    {
        printf("%d\t", arr[i]);
    }
    printf("\n\n");
  }
  else if(pid == 0)
  {
    printf("Child Process :\nNumbers to sort :\n");
    for(int i=0;i<5;i++)
    {
        printf("%d\t", arr[i]);
    }
    printf("\n\n");
  }
}
```

Output :

```
(base) Aadhityas-MacBook-Air:DA1 aadhitya$ gcc q2.c
(base) Aadhityas-MacBook-Air:DA1 aadhitya$ ./a.out
Parent Process :
Sorted numbers are :
1       2       3       4       5

Child Process :
Numbers to sort :
5       2       1       3       4

(base) Aadhityas-MacBook-Air:DA1 aadhitya$
```

Q3)

Zombie process :

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process. If there are a lot of zombie processes, then all the available process ID's are monopolised by them. This prevents other processes from running as there are no process ID's available.

Code :

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t pid1 = fork();
   if (pid1 > 0)
      sleep(500);
   else
      exit(0);
}
```

Here the child exits the execution but still there would be an entry in the process table as the parent was sleeping for a particular duration of time. Thus this child process would be a zombie process.

Let the initial process table be as follows :

pid	PCB
pid1(Parent process)	addr1
pid2(Child process)	addr2
pid3(Init process)	addr3

The process table after executing the above code would be as follows :

pid	PCB
pid1(Parent process)	addr1
pid2(Child process)	addr2
pid3(Init process)	addr3

Orphan Process :

The process whose parent process has completed execution or has terminated and do not exists in the process table are called orphan process. Usually, a parent process waits for its child to terminate or finish their job and report to it after execution but when it fails to do so it results in the Orphan process. In most cases, the Orphan process is immediately adopted by the init process.

Code :

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        pid_t pid = fork();
        if (pid > 0)
        {
                printf("Parent process\n");
                printf("ID : %d\n\n”,getpid());
        }
        else if (pid == 0)
        {
                printf("Child process\n");
                printf("ID: %d\n",getpid());
                printf("Parent -ID: %d\n\n",getppid());
                sleep(10);
                printf("\nChild process \n");
                printf("ID: %d\n",getpid());
                printf("Parent -ID: %d\n",getppid());
        }
        else
        {
```

```
                printf("Failed to create child process");
        }
}
```

Output :

This code states a scenario when a parent process ends when a child process is in sleep state. This makes the parent process to be unavailable for the child.

Let the initial process table be as follows :

pid	PCB
pid1(Parent process)	addr1
pid2(Child process)	addr2
pid3(Init process)	addr3

The process table after executing the above code would be as follows :

pid	PCB
pid2(Child process)	addr2
pid3(Init process)	addr3

Q4)

Let initially the process table is as follows-

pid	PCB
pid1(Parent process)	addr1
pid2(Child process)	addr2
pid3(Init process)	addr3

Methods to prevent the process from becoming a Zombie-

A. Using wait() system call:

By using a wait() system call in the parent process indicates that the parent process will wait for the child to terminate and until then all the execution in parent process is terminated. When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
A process that calls wait or waitpid() can :
– Block, if all of its children are still running.
– Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched.
– Return immediately with an error, if it doesn't have any child processes.

Program for using wait as a call-

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

   pid_t pid = fork();
   if (pid > 0)
      {
          wait(null);
          while(1);
          }
   else
      exit(0);

   return 0;
}
```

Process table after wait System call-

pid	PCB
pid1(Parent process)	addr1
pid3(Init process)	addr3

2. By ignoring the SIGCHLD signal :

When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. If we call the 'signal(SIGCHLD,SIG_IGN)', then the SIGCHLD signal is ignored by the system, and the child process entry is deleted from the process table. Thus, no zombie is created. But the parent process does not get the exit status of the child process.

Program -

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>
int main()
{
   int i;
   pid_t pid = fork();
   if (pid == 0)
      exit(0);
   else
   {
      signal(SIGCHLD,SIG_IGN);
      while(1);
   }
}
```

Process table after execution-

pid	PCB
pid1(Parent process)	addr1
pid3(Init process)	addr3

3. By using a signal handler :

The parent process installs a signal handler for the SIGCHLD signal. The signal handler calls wait() system call within it. In this scenario, when the child terminated, the SIGCHLD is delivered to the parent. On receipt of SIGCHLD, the corresponding handler is activated, which in turn calls the wait() system call. Hence, the parent collects the exit status almost immediately and the child entry in the process table is cleared. Thus no zombie is created.

Program:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>
void func(int signum)
{
   wait(NULL);
}
int main()
{
   int i;
   pid_t pid = fork();
   if (pid == 0)
      exit(0);
   else
   {
      signal(SIGCHLD, func);
      while(1);
   }
}
```

Process table after execution:

pid	PCB
pid1(Parent process)	addr1
pid3(Init process)	addr3

To View Process Table :

The process table can be viewed in the terminal and it lists the current executing process. The command to display the table in a linux terminal is **"ps".** A sample for such a table is shown as follows :