ASSIGNMENT No: 2

Title : Process Control System Calls

AIM:- Process control system calls: The demonstration of FORK, EXECVE and WAIT system calls along with zombie and orphan states.

a. Implement the C program in which main program accepts the integers to be sorted. Main program uses the FORK system call to create a new process called a child process. Parent process sorts the integers using sorting algorithm and waits for child process using WAIT system call to sort the integers using any sorting algorithm. Also demonstrate zombie and orphan states.

b. Implement the C program in which main program accepts an array. Main program uses the FORK system call to create a new process called a child process. Parent process sorts an array and passes the sorted array to child process through the command line arguments of EXECVE system call. The child process uses EXECVE system call to load new program which display array in reverse order.

THEORY:

Process in UNIX:

A process is the basic active entity in most operating-system models.

Process IDs

- Each process in a Linux system is identified by its unique process ID, sometimes referred to as pid.
- Process IDs are 16-bit numbers that are assigned sequentially by Linux as new processes are created.
- When referring to process IDs in a C or C++ program, always use the pid_t typedef, which is defined in <sys/types.h>
- A program can obtain the process ID of the process it's running in with the getpid() system call, and it can obtain the process ID of its parent process with the getpid() system call.

Creating ProcessesUsing fork()

- A process can create a new process by calling fork().
- The calling process becomes the parent, and the created process is called the child.
- The fork function copies the parent's memory image so that the new process receives a copy of the address space of the parent.
- Both processes continue at the instruction after the fork statement (executing in their respective memory images).
- **SYNOPSIS**
  #include <unistd.h> pid_t fork(void);
  - The fork function returns 0 to the child and returns the child's process ID to the parent. When fork fails, it returns –1.

The wait Function

- When a process creates a child, both parent and child proceed with execution from the point of the fork.
- The parent can execute wait to block until the child finishes.
- The wait function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.
- **SYNOPSIS**
  #include <sys/wait.h>
  pid_t wait(int *status);
  - If wait returns because the status of a child is reported, these functions return the process ID of that child. If an error occurs, these functions return –1.
- Example:

```
pid_t childpid;
childpid      =wait(NULL);
if(childpid != -1)
printf("Waited for child with pid %ld\n", childpid);
```

Status values

- The status argument of wait is a pointer to an integer variable.
- If it is not NULL, this function stores the return status of the child in this location.
- The child returns its status by calling exit, _exit or return from main.
- A zero return value indicates EXIT_SUCCESS; any other value indicates EXIT_FAILURE.
- POSIX specifies six macros for testing the child's return status.
- Each takes the status value returned by a child to wait as a parameter.
- Following are the two such macros:
- **SYNOPSIS**

```
#include <sys/wait.h>
WIFEXITED(int stat_val)
WEXITSTATUS(int stat_val)
```

- New program execution within the existing process (The exec Function)
- The fork function creates a copy of the calling process, but many applications require the child process to execute code that is different from that of the parent.
- The exec family of functions provides a facility for overlaying the process image of the calling process with a new image.
- The traditional way to use the fork–exec combination is for the child to execute (with an exec function) the new program while the parent continues to execute the original code.

exec() system call:

- The exec() system call is used after a fork() system call by one of the two processes to replace the memory space with a new program.
- The exec() system call loads a binary file into memory (destroying image of the program containing the exec() system call) and go their separate ways.
- Within the exec family there are functions that vary slightly in their capabilities.
- **exec family:**

execl() and execlp()

1. execl(): It permits us to pass a list of command line arguments to the program to be executed. The list of arguments is terminated by NULL.

e.g. execl("/bin/ls", "ls", "-l", NULL);

2. execlp(): It does same job except that it will use environment variable PATH to determine whichexecutable to process. Thus a fully qualified path name would not have to be used. The function execlp() can also take the fully qualified name as it also resolves explicitly.

e.g. execlp("ls", "ls", "-l", NULL);

Process Termination

- Normally, a process terminates in one of two ways.
- **exit() function-**Either the executing program calls the exit() function, or the program's main function returns.
- Each process has an exit code: a number that the process returns to its parent.
- The exit code is the argument passed to the exit function, or the value returned from main.
- **Zombie Processes-** If a child process terminates while its parent is calling a wait function, the child process vanishes and its termination status is passed to its parent via the wait call. But what happens when a child process terminates and the parent is not calling wait? Does it

simply vanish? No, because then information about its termination—such as whether it exited normally and, if so, what its exit status is—would be lost. Instead, when a child process terminates, is becomes a zombie process.

- A zombie process is a process that has terminated but has not been cleaned up yet. It is the responsibility of the parent process to clean up its zombie children. The wait functions do this, too, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose, for instance, that a program forks a child process, performs some other computations, and then callswait.
    - If the child process has not terminated at that point, the parent process will block in the wait call until the child process finishes.
    - If the child process finishes before the parent process calls wait, the child process becomes a zombie.
    - When the parent process calls wait, the zombie child's termination status is extracted, the child process is deleted, and the wait call returns immediately.
- **Orphan Process-** An Orphan Process is nearly the same thing which we see in real world. Orphan means someone whose parents are dead. The same way this is a process, whose parents are dead, that means parents are either terminated, killed or exited but the child process is still alive.

- In Linux/Unix like operating systems, as soon as parents of any process are dead, re-parenting occurs, automatically.
- Re-parenting means processes whose parents are dead, means Orphaned processes, are immediately adopted by special process.
- Thing to notice here is that even after re-parenting, the process still remains Orphan as the parent which created the process is dead, Reasons for Orphan Processes:
    - A process can be orphaned either intentionally or unintentionally.
    - Sometime a parent process exits/terminates or crashes leaving the child process still running, and then they become orphans.
    - Also, a process can be intentionally orphaned just to keep it running.
    - For example when you need to run a job in the background which need any manual intervention and going to take long time, then you detach it from user session and leave it there. Same way, when you need to run a process in the background for infinite time, you need to do the same thing. Processes running in the background like this are known as daemon process.

Example 1

Printing the Process ID

```
#include <stdio.h> #include <unistd.h>
int main()
{
printf("The process ID is %d\n", (int) getpid());
printf("The parent process ID is %d\n", (int) getppid());
return 0;
}
```

Example 2

Using the system call

```
#include <stdlib.h> int main()
{
int return_value;
return_value=system("ls –l /");
return return_value;
}
```