Assignment No 02 (A)

Process Control System Calls

AIM:

The demonstration of fork, execve and wait system calls along with zombie and orphan states.

1. 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 merge sort and waits for child process using wait system call to sort the integers using quick sort. Also demonstrate zombie and orphan states.
2. Implement the C program in which main program accepts an integer array. Main program uses the fork system call to create a new process called a child process. Parent process sorts an integer 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 that uses this sorted array for performing the binary search to search the particular item in the array.

OBJECTIVE:

This assignment covers the UNIX process control commonly called for process creation, program execution and process termination. Also covers process model, including process creation, process destruction, zombie and orphan processes.

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 getppid() system call.

Creating Processes

Two common techniques are used for creating a new process.

- using system() function.
- using fork() system calls.

1. **Using system**

The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell. In fact, system

creates a subprocess running the standard Bourne shell (/bin/sh)and hands the command to that shell for execution.

The system function returns the exit status of the shell command. If the shell itself cannot be run, system returns 127; if another error occurs, system returns –1.

2. **Using 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.

Assignment No 02 (B)
Process Control System Calls

SYNTAX

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg0, ... /*, char *(0) */);
int execle (const char *path, const char *arg0, ... /*, char *(0), char *const envp[] */);
```

- int execlp (const char *file, const char *arg0, ... /*, char *(0) */);
- int execv(const char *path, char *const argv[]);

- int execve (const char *path, char *const argv[], char *const envp[]);
- int execvp (const char *file, char *const argv[]);

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:

1. execl() and execlp():
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);

execlp(): It does same job except that it will use environment variable PATH to determine which executable 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);

2. execv() and execvp():
execv(): It does same job as execl() except that command line arguments can be passed to it in the form of an array of pointers to string.
e.g. char *argv[] = ("ls", "-l", NULL);
execv("/bin/ls", argv);

execvp(): It does same job expect that it will use environment variable PATH to determine which executable to process. Thus a fully qualified path name would not have to be used.
e.g. execvp("ls", argv);

3. execve( ):
int execve(const char *filename, char *const argv[ ], char *const envp[ ]);
It executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form: argv is an array of argument strings passed to

the new program. By convention, the first of these strings should contain the filename associated with the file being executed. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both argv and envp must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

int main(int argc, char *argv[ ] , char *envp[ ])]

execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

All exec functions return –1 if unsuccessful. In case of success these functions never return to the calling function.

Process Termination

Normally, a process terminates in one of two ways. 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 calls wait. 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.

Finding a Orphan Process:

It is very easy to spot a Orphan process. Orphan process is a user process, which is having init (process id â€“ 1) as parent. You can use this command in linux to find the Orphan processes.

```
# ps -elf | head -1; ps -elf | awk '{if ($5 == 1 && $3 != "root") {print $0}}' | head
```

This will show you all the orphan processes running in your system. The output from this command confirms that they are Orphan processes but does not mean that they are all useless, so confirm from some other source also before killing them.

Killing a Orphan Process:

As orphaned processes waste server resources, so it is not advised to have lots of orphan processes running into the system. To kill a orphan process is same as killing a normal process.

```
# kill -15 <PID>
```

If that does not work then simply use

```
# kill -9 <PID>
```

Daemon Process:

It is a process that runs in the background, rather than under the direct control of a user; they are usually initiated as background processes.

vfork: alternative of fork

create a new process when exec a new program.

Compare with fork:

- Creates new process without fully copying the address space of the parent.
- vfork guarantees that the child runs first, until the child calls exec or exit.
- When child calls either of these two functions(exit, exec), the parent resumes.

INPUT:

- An integer array with specified size.
- An integer array with specified size and number to search.

OUTPUT:

- Sorted array.
- Status of number to be searched.

FAQS:

bie process ?
n ?
ur system ?

rent and child process?
Block?

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

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;
}
```

Example 3

Using fork to duplicate a program's process

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

int main()
{
pid_t child_pid;
printf("The main program process ID is %d\n", (int) getpid());
```

```
child_pid=fork();
if(child_pid!=0)  {
printf(“This is the parent process ID, with id %d\n”, (int) getpid());
printf(“The child process ID is %d\n”, (int) chi;d_pid);
}
else
printf(“This is the child process ID, with id %d\n”, (int) getpid());
return 0;
}
```

Example 4

Determining the exit status of a child.

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

void show_return_status(void)
{

    pid_t childpid; int
    status;

    childpid = wait(&status); if
    (childpid == -1)

        perror("Failed to wait for child");

    else if (WIFEXITED(status))

        printf("Child %ld terminated with return status %d\n", (long)childpid,
                WEXITSTATUS(status));

}
```

Example 5

A program that creates a child process to run ls -l.

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

int main(void)
{
    pid_t childpid;

    childpid = fork();
```

```
    if (childpid == -1) { perror("Failed to
        fork"); return 1;

    }
    if (childpid == 0) {

                                    /* child code */
        execl("/bin/ls", "ls", "-l", NULL);
        perror("Child failed to exec ls"); return 1;

    }

    if (childpid != wait(NULL)) {

                                /* parent code */

        perror("Parent failed to wait due to signal or error"); return 1;

    }
    return 0;
}
```

Example 6
Making a zombie process

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

int main()
{
pid_t child_pid;
//create a child process
child_pid=fork();
if(child_pid>0)  {
//This is a parent process. Sleep for a minute
sleep(60)
}
else
{
//This is a child process. Exit immediately.
exit(0);
}
return 0;
}
```

Example 7
Demonstration of fork system call

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

int main()
{
pid_t pid;
char *msg;
int n;
printf("Program starts\n");
pid=fork();
switch(pid)
{
case -1:
printf("Fork error\n");
exit(-1);
case 0:
msg="This is the child process";
n=5;
break;
default:
msg="This is the parent process";
n=3;
break;
}
while(n>0)
{
puts(msg);
sleep(1);
n--;
}
return 0;
}
```

Example 8
Demo of multiprocess application using fork()system call

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

#define SIZE 1024
void do_child_proc(int pfd[2]);
void do_parent_proc(int pfd[2]);

int main()
{
int pfd[2];
```

```
int ret_val,nread;
pid_t pid;
ret_val=pipe(pfd);
if(ret_val==-1)
{
perror("pipe error\n");
exit(ret_val);
}
pid=fork();
switch(pid)
{
case -1:
printf("Fork error\n");
exit(pid);
case 0:
do_child_proc(pfd);
exit(0);
default:
do_parent_proc(pfd);
exit(pid);
}
wait(NULL);

return 0;
}

void do_child_proc(int pfd[2])
{
int nread;
char *buf=NULL;
printf("5\n");
close(pfd[1]);
while(nread=(read(pfd[0],buf,size))!=0)
printf("Child Read=%s\n",buf);
close(pfd[0]);
exit(0);
}
void do_parent_proc(int pfd[2])
 {
char ch;
char *buf=NULL;
close(pfd[0]);
while(ch=getchar()!='\n')  {
printf("7\n");
*buf=ch;
buff+;
}
*buf='\0';
write(pfd[1],buf,strlen(buf)+1);
close(pfd[1]);
}
```