Roll No :- 33328

NAME :-Vidhita Amit Jain

ASSIGNMENT No: 2 B

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.

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 which display array in reverse order.

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:

exec() system call:

The exec system call replaces the running process with some other executable process. The address space of the running process is replaced with the address space of the new process. It is important to note that the new program is loaded into the same address space. The process id remains the same. The new program will run independently; that is, the starting point will be the entry point of the new program. The exec system call has many variants.

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.

3. execve( ):

execvp("ls", argv);

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.