Experiment Number: 2

TITLE: UNIX Process Control- Program-II.

OBJECTIVE:

1. Study how to use wait(), exec() system calls,
2. Study of zombie, daemon and orphan states.

PROBLEM STATEMENT: 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.

THEORY:

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.

The wait() system call:

It blocks the calling process until one of its child processes exits or a signal is received. wait() takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of wait() is to wait for completion of child processes.

The execution of wait() could have two possible situations.

1. If there are at least one child processes running when the call to wait() is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

2. If there is no child process running when the call to wait() is made, then this wait() has no effect at all. That is, it is as if no wait() is there.

Zombie Process:

When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately. Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a zombie process.

Orphan Process:

An orphan process is a computer process whose parent process has finished or terminated, though itself remains running. A process may also be intentionally orphaned so that it becomes detached from the user's session and left running in the background; usually to allow a long-running job to complete without further user attention, or to start an indefinitely running service. Under UNIX, the latter kinds of processes are typically called daemon processes. The UNIX *nohup* command is one means to accomplish this.

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.

ALGORITHM:

PROGRAM (With output and Comments):

CONCLUSION (At least three points):

ASSIGNMENTS:

1. Write and implement any five options of **kill** command from command manual.
2. What is **nice** value? Write and implement nice command.
3. Explain with example, how to show processes created by system?

REFERENCES:

1. Beginning Linux Programming by Neil Mathew and Richard Stones, Wrox Publications.
2. Unix Concepts and Applications By Sumitabha Das, Tata McGraw Hill

Tushar B Kute
http://tusharkute.com