LINUX SYSTEM CALLS

Objectives:

In this lab, student will be able to:

1. Understand the working of the different system calls.
2. Understand the creation of new processes with fork and altering the code space of process using exec.

getpid()
This function returns the process identifiers of the calling process.

#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void); // this function returns the process identifier (PID)
pid_t getppid(void); // this function returns the parent process identifier (PPID)

fork()
A new process is created by calling fork. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors. Combined with the **exec** functions, **fork** is all we need to create new processes.

#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);

The return value of fork() is pid_t (defined in the header file sys/types.h). As seen in the Fig. 4.1, the call to fork in the parent process returns the PID of the new child process. The new process continues to execute just like the parent process, with the exception that in the child process, the PID returned is 0. The parent and child process can be determined by using the PID returned from fork() function. To the parent the fork()

returns the PID of the child, whereas to the child the PID returned is zero. This is shown in the following Fig. 4.1.

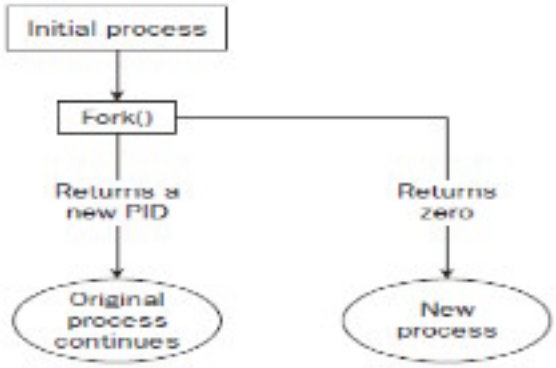

Figure 4.1 : Fork system call

In Linux in case of any error observed in calling the system functions, then a special variable called errno will contain the error number. To use errno a header file named errno.h has to be included in the program. If fork fails, it returns -1. This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN. If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM.

A typical code fragment using fork is

```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
break;
default : /* We are parent */
break;
}
```

Sample Program on fork1.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
        pid_t pid;
        char *message;
        int n;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        message = "This is the child";
                        n = 5;
                        break;
                default:
                        message = "This is the parent";
                        n = 3;
                        break;
        }
        for(; n > 0; n--) {
                puts(message);
                sleep(1);
        }
        exit(0);
}
```

This program runs as two processes. A child process is created and prints a message five times. The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.

```
$ ./fork1
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
```

When fork is called, this program divides into two separate processes. The parent process is identified by a nonzero return from fork and is used to set a number of messages to print, each separated by one second.

The wait() System Call

A parent process usually needs to synchronize its actions by waiting until the child process has either stopped or terminated its actions. The wait() system call allows the parent process to suspend its activities until one of these actions has occurred. The wait() system call accepts a single argument, which is a pointer to an integer and returns a value defined as type pid_t. If the calling process does not have any child associated with it, wait will return immediately with a value of -1. If any child processes are still active, the calling process will suspend its activity until a child process terminates.

Example of wait():

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

void main()
{
        int status;
        pid_t  pid;
        pid = fork();
        if(pid = -1)
          printf("\nERROR child not created ");
        else if (pid == 0) /* child process */
        {
                printf("\n I'm the child!");
                 exit(0);
        }
        else /* parent process */
        {
                wait(&status);
                printf("\n I'm the parent!")
                printf("\n Child returned: %d\n", status)
        }
}
```

A few notes on this program:

wait(&status) causes the parent to sleep until the child process has finished execution. The exit status of the child is returned to the parent.

The exit() System Call

This system call is used to terminate the current running process. A value of zero is passed to indicate that the execution of process was successful. A non-zero value is passed if the execution of process was unsuccessful. All shell commands are written in C including grep. grep will return 0 through exit if the command is successfully runs (grep could find pattern in file). If grep fails to find pattern in file, then it will call exit() with a non-zero value. This is applicable to all commands.

The exec() System Call

The exec function will execute a specified program passed as argument to it, in the same process (Fig. 4.2). The exec() will not create a new process. As new process is not created, the process ID (PID) does not change across an execute, but the data and code of the calling process are replaced by those of the new process.

fork() is the name of the system call that the parent process uses to "divide" itself ("fork") into two identical processes. After calling fork(), the created child process is actually an exact copy of the parent - which would probably be of limited use - so it replaces itself with another process using the system call exec().

The versions of exec are:

- execl
- execv
- execle
- execve
- execlp
- execvp

The naming convention: exec*

- 'l' indicates a list arrangement (a series of null terminated arguments)
- 'v' indicate the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their own environment variable list
- 'p' indicates the current PATH string should be used when the system searches for executable files.

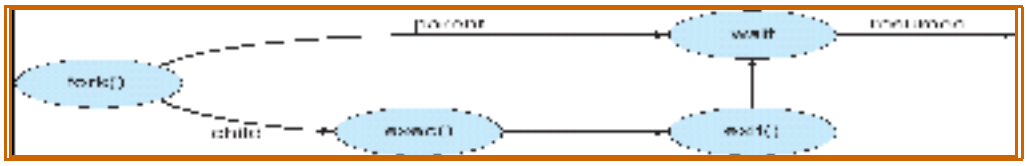

Figure 4.2: exec() system call

The parent process can either continue execution or wait for the child process to complete. If the parent chooses to wait for the child to die, then the parent will receive the exit code of the program that the child executed. If a parent does not wait for the child, and the child terminates before the parent, then the child is called **zombie** process. If a parent terminates before the child process then the child is attached to a process called init (whose PID is 1). In this case, whenever the child does not have a parent then child is called **orphan** process.

Sample Program:

C program forking a separate process.

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

{
        pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

execl: is used with a list comprising the command name and its arguments:

int execl(const char *path, const char *arg0, …../*, (char *) 0 */);

This is used when the number of arguments are known in advance. The first argument is the pathname which could be absolute or a relative pathname, The arguments to the command to run are represented as separate arguments beginning with the name of the command (*arg0). The ellipsis representation in the syntax (…/*) points to the varying number of arguments.

Example: How to use execl to run the wc –l command with the filename foo as argument:

execl (“/bin/wc”, “wc”, “-l”, “foo”, (char *) 0);
execl doesn’t use PATH to locate wc so pathname is specified as the first argument.

execv: needs an array to work with.
int execv(const char *path, char *const argv[ ]);

Here path represents the pathname of the command to run. The second argument represents an array of pointers to char. The array is populated by addresses that point to strings representing the command name and its arguments, in the form they are passes to the main function of the program to be executed. In this case also the last element of the argv[ ] must be a null pointer.

Here the following program uses execv program to run grep command with two options to look up the author’s name in /etc/passwd. The array *cmdargs[ ] are populated with the strings comprising the command line to be executed by execv. The first argument is the pathname of the command:

```
#include<stdio.h>
int main(int argc, char **argv){
char *cmdargs[ ] = {“grep”, “-I”, “-n”, “SUMIT”, “/etc/passed”, “NULL};
execv(“/bin/grep”, cmdargs);
printf (“execv error\n”);
}
```

Drawbacks:
Need to know the location of the command file since neither execl nor execv will use PATH to locate it. The command name is specified twice- as the first two arguments. These calls can't be used to run a shell script but only binary executable. The program has to be invoked every time there is a need to run a command.

execlp and execvp: requires pathname of the command to be located. They behave exactly like their other counterparts but overcomes two of the four limitations discussed above. First the first argument need not be a pathname it can be a command name. Second these functions can also run a shell script.

int execlp(const char *file, const char *arg0, …./*, (char *) 0 */);
int execvp(const char *file, char *const argv[ ]);

execlp ("wc", "wc", "-l", "foo", (char *) 0);

execle and execve: All of the previous four exec calls silently pass the environment of the current process to the executed process by making available the environ[ ] variable to the overlaid process. Sometime there may be a need to provide a different environment to the new program- a restricted shell for instance. In that case these functions are used.

int execle(const char *path, const char *arg0, … /*, (char *) 0, char * const envp[ ] */);
int execve(const char *path, char * const argv[ ], char *const envp[ ]);

These functions unlike the others use an additional argument to pass a pointer to an array of environment strings of the form variable = value to the program. It's only this environment that is available in the executed process, not the one stored in envp[ ].

The following program (assume fork2.c) is the same as fork1.c, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code:

```
switch(pid)
{
        case -1:
```

```
                perror("fork failed");
                exit(1);
        case 0:
                message = "This is the child";
                n = 3;
                break;
        default:
                message = "This is the parent";
                n = 5;
                break;
}
```

When the preceding program is run with ./fork2 & and then call the ps program after the child has finished but before the parent has finished, a line such as this. (Some systems may say <zombie> rather than <defunct>) is seen.

I/O SYSTEM CALLS

I/O through system calls is simpler and operates at a lower level than making calls to the C file-I/O library.

There are seven fundamental file-I/O system calls:

creat() Create a file for reading or writing.
open() Open a file for reading or writing.
close() Close a file after reading or writing.
unlink() Delete a file.
write() Write bytes to file.
read() Read bytes from file.

The creat() System Call

The "creat()" system call creates a file. It has the syntax: int fp; /* fp is the file descriptor variable */

fp = creat(<filename>, <protection bits>);

Ex: fp=creat(“students.dat”,RD_WR);

This system call returns an integer, called a "file descriptor", which is a number that identifies the file generated by "creat()". This number is used by other system calls in the program to access the file. Should the "creat()" call encounter an error, it will return a file descriptor value of -1.

The "filename" parameter gives the desired filename for the new file.
The "permission bits" give the "access rights" to the file. A file has three "permissions" associated with it:

Write permission - Allows data to be written to the file.
Read permission - Allows data to be read from the file.
Execute permission -Designates that the file is a program that can be run.

These permissions can be set for three different levels:

User level: Permissions apply to individual user.
Group level: Permissions apply to members of user's defined "group".
System level: Permissions apply to everyone on the system

The open() System Call
The "open()" system call opens an existing file for reading or writing. It has the syntax:

<file descriptor variable> = open(<filename>, <access mode>);

The "open()" call is similar to the "creat()" call in that it returns a file descriptor for the given file, and returns a file descriptor of -1 if it encounters an error. However, the second parameter is an "access mode", not a permission code. There are three modes (defined in the "fcntl.h" header file):

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.

O_RDWR Open for reading and writing

For example, to open "data" for writing, assuming that the file had been created by another program, the following statements would be used:

```
int fd;
fd = open( "students.dat", O_WRONLY );
```

A few additional comments before proceeding:
A "creat()" call implies an "open()". There is no need to "creat()" a file and then "open()" it.

The close() System Call

The "close()" system call is very simple. All it does is "close()" an open file when there is no further need to access it. The "close()" system call has the syntax:

```
close( <file descriptor> );
```

The "close()" call returns a value of 0 if it succeeds, and returns -1 if it encounters an error.

The write() System Call

The "write()" system call writes data to an open file. It has the syntax:
write(<file descriptor>, <buffer>, <buffer length>);

The file descriptor is returned by a "creat()" or "open()" system call. The "buffer" is a pointer to a variable or an array that contains the data; and the "buffer length" gives the number of bytes to be written into the file.

While different data types may have different byte lengths on different systems, the "sizeof()" statement can be used to provide the proper buffer length in bytes. A "write()" call could be specified as follows:

```
float array[10];
write( fd, array, sizeof( array ) );
```

The "write()" function returns the number of bytes it actually writes. It will return -1 on an error.

The read() Sytem Call

The "read()" system call reads data from a open file. Its syntax is exactly the same as that of the "write()" call:

read(<file descriptor>, <buffer>, <buffer length>);

The "read()" function returns the number of bytes it actually returns. At the end of file it returns 0, or returns -1 on error.

lseek: The lseek system call sets the read/write pointer of a file descriptor, fildes; that is, we can use it to set where in the file the next read or write will occur. We can set the pointer to an absolute location in the file or to a position relative to the current position or the end of file.

```
#include <unistd.h>
#include <sys/types.h>
off_t lseek(int fildes, off_t offset, int whence);
```

The offset parameter is used to specify the position, and the whence parameter specifies how the offset is used. whence can be one of the following:

SEEK_SET: offset is an absolute position
SEEK_CUR: offset is relative to the current position
SEEK_END: offset is relative to the end of the file

lseek returns the offset measured in bytes from the beginning of the file that the file pointer is set to, or -1 on failure. The type off_t, used for the offset in seek operations, is an implementation-dependent type defined in sys/types.h.

Errors*:*
EACCES Permission denied.
EMFILE Too many file descriptors in use by process.
ENFILE Too many files are currently open in the system.
ENOENT Directory does not exist, or name is an empty string.
ENOMEM Insufficient memory to complete the operation.