EXPERIMENT - 3

System calls of Linux operating system

(fork, exec, getpid, exit, wait, close, stat, opendir, readdir)

AIM:

To write C Programs using the following system calls of UNIX operating system fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

System Calls

System call is a request to the operating system to perform some activity.

A system call is a procedure that provides the interface between a process and the operating system. It is the way by which a computer program requests a service from the kernel of the operating system.

System calls are divided into 5 categories mainly:

- Process Control
- File Management
- Device Management
- Information Maintenance
- Communication

Process Control :

Process Controlsystem calls perform the task of process creation, execution, process termination, etc. The Linux System calls under this category are fork() , exit() , exec().

File Management:

File management system calls handle file manipulation jobs such as creating a file, reading, and writing, etc. The Linux System calls under this category are open(), read(), write(), close().

Linux System Calls

In Linux, making a system call includes transferring control from unprivileged user mode to privileged kernel mode. When a system call is used, it is communicating to the operating system and in return the OS communicates to the user through the parameters that are returned to system call functions (return values).

Opendir()

The opendir() function is used to open a directory stream corresponding to the specified directory name. It also returns a pointer to the directory stream. The stream is placed at the first entry in the directory.

Readdir()

The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp. It returns NULL on reaching the end of the directory stream or if an error occurred.

Closedir()

The closedir() function is used to close the directory stream specified with the directory pointer, dirp.

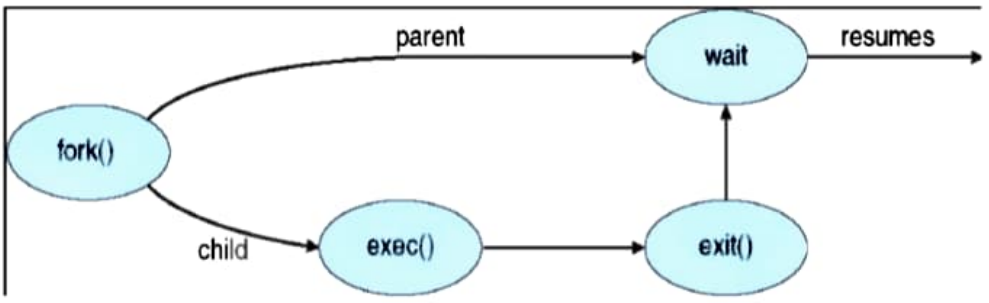
A successful call to closedir() also closes the underlying file descriptor associated with dirp. The directory stream descriptor dirpwill not be available after this call.

fork()

The fork system call creates a new child process. It creates a *copy* of the current process as a new child process, and then both processes resume execution from the fork() call. Since it creates two processes, fork also returns two values; one to teach process. To the parent process, fork returns the *process id of the newly created child process*. To the child process, fork returns 0. The reason it returns 0 is precisely because this is an invalid process id. You would have no way of differentiating between the parent and child processes if fork returned an arbitrary positive integer to each.

A call to fork looks like this:

```
int pid;
if ( (pid = fork()) == 0 ) {
/* child process executes inside here */
}
else {
/* parent process executes inside here */
}
```



execvp&execlp

The exec functions are a family of functions that *execute* some program *within* the current process space. So, if I write a program that calls one of the exec functions, as soon as the function call succeeds the original process gets *replaced* with whateverprogram I asked the exec to execute. This is usually used along with a fork call. You would typically fork a child process, and then call exec from within the child process, to execute some other program in the new process entry created by fork.

getpid

Each process is identified by a unique *process id* (called a "pid"). The init process (which is the supreme parent to all processes) possesses id 1. All other processes have some other (possibly arbitrary) process id. The getpid system call returns the current process' id as an integer.

```
// …
int pid = getpid();
printf("This process' id is %d\n",pid);// …
```

wait & exit

In order to wait for a child process to terminate, a parent process will just execute a **wait()** system call. This call will suspend the parent process until any of its child processes terminates, at which time the **wait()** call returns and the parent process can continue.
The prototype for the **wait()** call is:

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

pid_twait(int *status);
```

The return value from wait is the **PID** of the child process which terminated. The parameter to **wait()** is a pointer to a location which will receive the child's exit status value when it terminates.

When a process terminates it executes an **exit()** system call, either directly in its own code, or indirectly via library code. The prototype for the **exit()** call is:

```
#include <stdlib.h>

void exit(int status);
```

The **exit()** call has no return value as the process that calls it terminates and so couldn't receive a value anyway. Notice, however, that **exit()** does take a parameter value - status. As well as causing a waiting parent process to resume execution, **exit()** also returns the status parameter value to the parent process via the location pointed to by the **wait()** parameter.

3.a) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (OPENDIR, READDIR, CLOSEDIR)

ALGORITHM:

1) Start the program.
2) Create structure variable, struct dirent.
3) Declare a variable buffer and pointer dptr.
4) Read the name of the directory.
5) Open the directory.
6) Read the contents in directory and display the content details.
7) Close the directory.
8) Stop the program

PROGRAM:

```
/* geditdircommands.c */
#include<stdio.h>
#include<stdlib.h>
#include<dirent.h>
struct dirent *dptr;
int main(int argc, char *argv[])
{
char buffer[100];
DIR *dirp;
```

```
printf("\n ENTER DIRECTORY NAME");
scanf("%s", buffer);
if((dirp=opendir(buffer))==NULL)
{
printf("The given directory does not exist");
exit(1);
}
while(dptr=readdir(dirp))
{
printf("%s\n",dptr->d_name);
}
closedir(dirp);
}
```

OUTPUT:

```
ubuntu@ubuntu:~/skj$ geditdircommands.c
ubuntu@ubuntu:~/skj$ gccdircommands.c
ubuntu@ubuntu:~/skj$ ./a.out
ubuntu@ubuntu:~/skj$ mkdir folder1
ubuntu@ubuntu:~/skj$ cd folder1
ubuntu@ubuntu:~/skj/folder1$ cat > f1
Hai
ubuntu@ubuntu:~/skj/folder1$ cat > f2
Hello
ubuntu@ubuntu:~/skj$ ./a.out
ENTER DIRECTORY NAME folder1
.
..
f2
f1
```

3.b) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEM (fork, getpid, exit)

ALGORITHM:

1) Start the program.
2) Declare three variables pid, pid1, pid2.
3) Create process using fork() system call.
4) If pid==-1, exit.
5) If pid!=-1 , get the process id using getpid() system call.
6) Display the process id of parent and child process.
7) Stop the program

PROGRAM:

```
/* geditfork.c */
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
int pid,pid1,pid2;
pid=fork();
if(pid==-1)
{
printf("ERROR IN PROCESS CREATION \n");
exit(1);
}
if(pid!=0)
{
pid1=getpid();
printf("\n the parent process ID is %d\n", pid1);
}
else
{
pid2=getpid();
printf("\n the child process ID is %d\n", pid2);
}
return 0;
}
```

OUTPUT:

```
ubuntu@ubuntu:~/skj$ gedit dircommands2.c
ubuntu@ubuntu:~/skj$ gcc dircommands2.c
ubuntu@ubuntu:~/skj$ ./a.out
the parent process ID is 11306
the child process ID is 11307
```

3.c) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (wait, exit, close)

AIM :

To Execute a Unix Command in a 'C' program using wait() system call.

ALGORITHM :

1) Start the program
2) Initialize variables such as pid, status,
3) Use wait() to return the parent id of the child else return -1 for an error
4) Stop the program.

PROGRAM:

```
/* wait.c */
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{
int pid,status,exitch;
if((pid=fork())==-1)
{
perror("error");
exit (0);
}
if(pid==0)
{
sleep(1);
printf("child process");
exit (0);
}
else
{
printf("parent process\n");
if((exitch=wait(&status))==-1)
{
perror("during wait()");
exit (0);
}
printf("\nparent existing\n");
exit (0);
}
return 0;
}
```

OUTPUT:

```
buntu@ubuntu:~/OSLab$ gccwait.c
ubuntu@ubuntu:~/OSLab$ ./a.out
parent process
child process
parent existing
ubuntu@ubuntu:~/OSLab$
```

3.d) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (PROGRAM to get information about the file using **stat** system call.)

ALGORITHM:

1) Start the program
2) Declare structure variable struct stat
3) Allocate the size for the file by using malloc function
4) Read the filename whose status has to be displayed
5) Stop the program

PROGRAM:

```
/* stat.c */
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdlib.h>
int main()
{
char *path,path1[10];
struct stat *nfile;
nfile=(struct stat *) malloc (sizeof(struct stat));
printf("Enter name of the file forgetting its statistics: ");
scanf("%s",path1);
stat(path1,nfile);
printf("user id %d\n",nfile->st_uid);
printf("block size : %ld\n",nfile->st_blksize);
printf("last access time: %ld\n",nfile->st_atime);
printf("time of last modification: %ld\n",nfile->st_atime);
printf("production mode: %o \n",nfile->st_mode);
printf("size of file: %ld\n",nfile->st_size);
printf("number of links: %ld\n",nfile->st_nlink);
return 0;
}
```

OUTPUT:

```
ubuntu@ubuntu:~/OSLab$ gccstat.c
ubuntu@ubuntu:~/OSLab$ ./a.out
Enter name of the file for getting  itsstatistics: stat.c
user id 999
block size : 4096
last access time: 1660724493
time of last modification: 1660724493
production mode: 100664
size of file: 686
number of links: 1
```

3.e) PROGRAM FOR SYSTEM CALLS OF UNIX OPERATING SYSTEMS (PROGRAM to get information about the file using **exec**system call.)

AIM:

To Execute a Unix Command in a 'C' program using exec() system call.

ALGORITHM:

1) Start the program.
2) Declare the variables, pid.
3) Call the functionexecv (filename,argv) to transform an executable binary file into process.
4) Repeat this process until all the executed files are displayed.
5) Stop the program.

PROGRAM:

```
// exec() System call
#include<stdio.h>
int main()
{
int pid;

// A null terminated array of character pointers

char *args[]={"/bin/ls","-l",0};
printf("\nParent process");
pid=fork();
if(pid==0)
{
execv("/bin/ls",args);
}
else
{
printf("\nChild process");
}
return 0;
}
```

OUTPUT:

```
total 440
-rwxrwxr-x 1 skec25 skec25 5210 Apr 16 06:25 a.out
-rw-rw-r-- 1 skec25 skec25 775 Apr 9 08:36 bestfit.c
-rw-rw-r-- 1 skec25 skec25 1669 Apr 10 09:19 correctpipe.c
-rw-rw-r-- 1 skec25 skec25 977 Apr 16 06:15 correctprio.c
-rw------- 1 skec25 skec25 13 Apr 10 08:14 datafile.dat
```

-rw------- 1 skec25 skec25 13 Apr 10 08:15 example.dat

-rw-rw-r-- 1 skec25 skec25 166 Apr 16 06:25 exec.c

-rw-rw-r-- 1 skec25 skec25 490 Apr 10 09:43 exit.c
Parent Process

RESULT:
The programs are compiled, executed and the output is verified.