Experiment No. 5

System Calls

A system call provides an interface to the services made available by operating system. It is programmatic way by which a computer program requests a service from the kernel of the operating system. This document is a handout about some of the frequently used system calls.

1. `fork()` system call

Synopsis

```
#include  <unistd.h>
pid_t  fork(void);
```

`fork()` is used to create a new process which becomes child process of the caller. After a new child process is created, both processes will execute the next instruction following the `fork()` system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of `fork()`.

- If `fork()` returns a negative value, the creation of a child process was unsuccessful.
- `fork()` returns a zero to the newly created child process.
- `fork()` returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type `pid_t` defined in `sys/types.h`. Normally, the process ID is an integer.

2. `getpid()` and `getppid()` system call

Synopsis

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

```
pid_t  getpid(void);
pid_t  getppid(void);
```

getpid() returns the process ID of the calling process. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer.

getppid() returns the process ID of the parent of the calling process. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer.

3. open() system call

Synopsis

```
#include  <sys/types.h>
#include  <sys/stat.h>
#include  <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

The open() system call opens the file specified by pathname. If the specified file does not exist, it may optionally (if O_CREAT is specified in flags) be created by open() .

The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively. In addition more file creation flags and file status flags can be bitwise-or'd in flags.

The return value of open() is a file descriptor, a small, nonnegative integer that is used to refer to the open file.

4. `close()` system call

Synopsis

```
#include <unistd.h>

int close(int fd);
```

`close()` closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks held on the file it was associated with, and owned by the process, are removed.

If file descriptor closed via `close()` is the last file descriptor referring to the underlying open file description, the resources associated with the open file description are freed;

`close()` returns zero on success. On error, -1 is returned.

5. `read()` system call

Synopsis

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

`read()` attempts to read up to count bytes from file descriptor `fd` into the buffer starting at `buf`.

If count is zero, `read()` may detect the errors. In the absence of any errors, or if `read()` does not check for errors, a read() with a count of 0 returns zero and has no other effects.

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

On error, -1 is returned.

6. **`write()`** system call

Synopsis

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

`write()` writes up to count bytes from the buffer starting at `buf` to the file referred to by the file descriptor `fd`.

The number of bytes written may be less than count if, for example, there is insufficient space on the underlying physical medium.

On success, the number of bytes written is returned (zero indicates nothing was written).

On error, -1 is returned

7. **`wait()`** system call

Synopsis

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

#include <sys/wait.h>

pid_t wait(int *wstatus);
```

`wait()` system call suspends execution of the calling process until one of its children terminates.

`wait()` used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state

If the argument `wstatus` is not `NULL`, `wait()` store status information in the `int` to which it points.

wait() on success, returns the process ID of the terminated child on error, -1 is returned.

8. **`opendir()`** system call

Synopsis

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
```

The `opendir()` system call opens a directory stream corresponding to the directory name

The `opendir()` system call return a pointer to the directory stream. On error, NULL is returned.

9. **`readdir()`** system call

Synopsis

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

The `readdir()` system call 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.

10. **`dup()`** and **`dup2()`** system calls

Synopsis

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

`dup()` system call creates a copy of the file descriptor `oldfd`, using the lowest-numbered unused file descriptor for the new descriptor. After a successful return, the old and new file descriptors may be used interchangeably. They refer to the same open file description.

`dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. If the file descriptor `newfd` was previously open, it is silently closed before being reused.

On success, these system calls return the new file descriptor. On error, -1 is returned

11. `pipe()` system call

Synopsis

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

`pipe()` creates a pipe, a unidirectional data channel that can be used for inter process communication. The array `pipefd` is used to return two file descriptors referring to the ends of the pipe.

`pipefd[0]` refers to the read end of the pipe.

`pipefd[1]` refers to the write end of the pipe.

Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

On success, zero is returned. On error, -1 is returned

12. `exec()` System Call

Synopsis

```
#include <unistd.h>

extern char **environ;
```

```
int execl(const char *path, const char *arg0, ..., const char
*argn, (char *)0);
```

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

```
int execlp(const char *file, const char *arg0, ..., const char
*argn, (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[]);
```

Shown above is a list of several different flavors of the `exec()` call, but they all perform the same task. Use of the `exec()` system call is the only way to get a program executed under Linux. The way that it does this is to replace the text and user data segments of the process that executes the `exec()` call with the text and user data contained in the program file whose name is passed as a parameter to `exec()`

The `exec()` family of functions shown above replaces a current process image with a new process image. The `exec()` family of functions creates a new process image from a regular, executable file. This file is either an executable object file, or an interpreter script.

There is no return from a successful call to an `exec()` function, because the calling process is functionally replaced by the new process.

Parameters or Arguments passed to various `exec()` calls.

`path`

Specifies the path name of the new process image file.

`file`

Is used to construct a path name that identifies the new process image file. If it contains a slash character, the argument is used as the path name for this file. Otherwise, the path prefix for this file is obtained by a search of the directories in the environment variable PATH. If PATH is not set, the current directory is searched.

`arg0, ..., argn`

Point to null-terminated character strings. These strings constitute the argument list for the new process image. The list is terminated by a NULL pointer. The argument `arg0` should point to a file name that is associated with the process being started by the `exec()` function.

`argv`

Is the argument list for the new process image. This should contain an array of pointers to character strings, and the array should be terminated by a NULL pointer. The value in `argv[0]` should point to a file name that is associated with the process being started by the `exec()` function.

`envp`

Specifies the environment for the new process image. This should contain an array of pointers to character strings, and the array should be terminated by a NULL pointer.

Example 1:

C program to show the usage of `fork(),getpid(),getppid()` syscall

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main() {
  pid_t  pid;
  pid_t  ppid;
  // make two process which run same
  // program after this instruction
  fork();
  fork();
  fork();

  pid = getpid();
  ppid = getppid();

  printf("\nHi this is Process with pid:%d and ppid:
  %d\n",pid,ppid);
  return 0;
```

Example 2:

C program to show the usage of `fork()`,`getpid()`,`getppid()` syscall with delay timer

```
#include<stdio.h>

#include<sys/types.h>

#include<unistd.h>

int main() {

      pid_t  pid;

 pid_t  ppid;

 // make two process which run same

 // program after this instruction

 fork();

 fork();

 fork();


 pid = getpid();

 ppid = getppid();

 printf("\nHi this is Process with pid:%d and ppid:
%d",pid,ppid);

 sleep(10);

 //sleep(5) //use this if it takes too long to finish

 return 0;

}
```

Example 3:

C program to show the usage of `fork(),getpid(),getppid()` syscall

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

int main(){
    int id ;
    id = fork() ;
    printf("id value : %d\n",id);
    if ( id == 0 )
    {
        printf ( "Child : Hello I am the child process\n");
        printf ( "Child : Child’s PID: %d\n", getpid());
        printf ( "Child : Parent’s PID: %d\n", getppid());
    }
    else
    {
        printf ( "Parent : Hello I am the parent process\n" ) ;
        printf ( "Parent : Parent’s PID: %d\n", getpid());
        printf ( "Parent : Child’s PID: %d\n", id);
    }

return 0;
}
```

Exercise on System calls

Q1. Write a program in C to create a child process using `fork()` system call.

(a) The child process displays its own process identifier and its parent process identifier. Subsequently, one new program is loaded into the address space of the child process for creating two files, named f1.txt and f2.txt, inside the present working directory.

(b) The parent process is suspended until the child process terminates. The parent process prints the value returned by the `fork()` system call and its own process identifier. Subsequently, the parent process executes a program to display the contents of present working directory in long-listing format.

Q2. Write a program in C to count the number of contents in a given directory using `opendir()` and `readdir()` system calls.

Q3. Write a program in C to create a child process using `fork()` system call.

(a) One new program is loaded into the address space of the child process for displaying all the processes running on a system. The output is stored in a file and the child process sends the file name to the parent process using `pipe()` system call.

(b) The parent process reads the file name and executes a program to search the PPID of the bash shell and stores the result into another file.