Operating System Lab Work

**Difference between Windows and Linux OS:**

**Windows**

**Linux**

**Cost**

**Free of cost**

**Expensive**

**Open Source**

**Yes**

**No**

**Customizable**

**Yes**

**No**

**Security**

More secure

Vulnerable to viruses and malware attack

**Booting**

Either primary or logical partition

Only primary partition

**Separation of directories using**

Back Slash

Forward slash

**File System**

EXT2, EXT3, EXT4, Reisers FS, XFS and JFS

FAT, FAT32, NTFS and ReFS

**Type of Kernel**

Monolithic kernel

Microkernel

**Efficiency**

Effective running efficiency

Lower than Linux

**System Calls:**

In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating **system** it is executed on. A **system call** is a way for programs to interact with the operating **system**.

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

In general, system calls are required in the following situations:

• If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.

• Creation and management of new processes.

• Network connections also require system calls. This includes sending and receiving packets.

• Access to a hardware device such as a printer, scanner etc. requires a system call.

**There are different system calls in Linux for:-**

**• Process Management**

**• File Management**

**• I/O System calls**

**• Process Management**

System calls for this type are:

• fork()

• exec()

• wait()

• exit()

**fork() system call**

The fork() system call is used to create processes. When a process (a program in execution) makes a fork() call, an exact copy of the process is created. Now there are two processes, one being the parent process and the other being the child process.

The process which called the fork() call is the parent process and the process which is created newly is called the child process. The child process will be exactly the same as the parent. Note that the process state of the parent i.e., the address space, variables, open files etc. is copied into the child process. This means that the parent and child processes have identical but physically different address spaces. The change of values in parent process doesn't affect the child and vice versa is true too.

**Example program 1:**

#include <sys/types.h>

#include <stdio.h>

int main(){

pid\_tpid;

pid = fork();

printf("Fork call 1\n");

pid = fork();

printf("Fork call 2\n");

pid = fork();

printf("Fork call 3\n");

pid = fork();

printf("Fork call 4\n");

return 0;

}

**Output:**

“Fork call 1” printed 2 times.

“Fork call 2” printed 4 times.

“Fork call 3” printed 8 times.

“Fork call 4” printed 16 times.

**exec() system call:**

The exec() system call is also used to create processes. The fork() call creates a new process while preserving the parent process. But, an exec() call replaces the address space, text segment, data segment etc. of the current process with the new process.

It means, after an exec() call, only the new process exists. The process which made the system call, wouldn't exist.The exec() family of functions **replaces** the current process image with a new process image. It loads the program into the current process space and runs it from the entry point.

There are many flavors of exec() in UNIX.

**Example Program:**

**execCalled.c**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main(){

printf("Inside execCalled, PID = ");

printf("%d\n", getpid());

return 0;

}

**execDemo.c**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(){

printf("Inside execDemo, PID = ");

printf("%d\n", getpid());

char \*args[] = {"./execCalled", NULL};

execvp(args[0], args);

printf("Now inside execDemo\n");

return 0;

}

**Output (execDemo.c):**

Inside execDemo, PID = 4770

Inside execCalled, PID = 4770

**wait() system call:**

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent continues its execution after wait system call instruction.

Child process may terminate due to any of these:

• It calls exit();

• It returns (an int) from main

• It receives a signal (from the OS or another process) whose default action is to terminate.

If any process has more than one child processes, then after calling wait(), parent process has to be in wait state if no child terminates.If only one child process is terminated, then return a wait() returns process ID of the terminated child process.If more than one child processes are terminated than wait() reap any arbitrarily child and return a process ID of that child process.

**Example Program:**

#include <stdio.h>

#include <stdio.h>

#include <unistd.h>

#include <sys/wait.h>

#include <stdlib.h>

int main(){

printf("PID of waitProg.c = %d\n", getpid());

pid\_t p;

p = fork();

if(p==0){

printf("\nInside the child process, PID = %d, PPID=%d\n", getpid(), getppid());

exit(0);

} else {

printf("Inside the parent process, PID = %d\n", getpid());

int waitVal = wait(NULL);

printf("Inside parent, after child execution\n");

}

return 0;

}

**Output:**

PID of waitProg.c = 5501

Inside the parent process, PID = 5501

Inside the child process, PID = 5502, PPID=5501

Inside parent, after child execution

**exit() system call:**

A call to exit(status) terminates the calling process immediately. Any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal. The status parameter is the status value returned to the parent process.

**Example program:**

#include <stdio.h>

#include <stdlib.h>

int main(){

printf("Start of the program\n");

printf("Before the exit call\n");

exit(0);

printf("After the exit call\n");

return 0;

}

**Output:**

Start of the program

Before the exit call

**File Management**

The various file management system calls are:

**•Open**

Used to Open the file for reading, writing or both.

Syntax:

#include<sys/types.h>

#includ<sys/stat.h>

#include <fcntl.h>

int open (const char\* Path, int flags [, int mode ]);

**How it works in OS**

• Find existing file on disk

• Create file table entry

• Set first unused file descriptor to point to file table entry

• Return file descriptor used, -1 upon failure

**Program**

#include<stdio.h>

#include<fcntl.h>

#include<errno.h>

extern int errno;

int main()

{

// if file does not have in directory

// then file foo.txt is created.

int fd = open("foo.txt", O\_RDONLY | O\_CREAT);

printf("fd = %d/n", fd);

if (fd ==-1)

{

// print which type of error have in a code

printf("Error Number % d\n", errno);

// print program detail "Success or failure"

perror("Program");

}

return 0;

}

Output:

fd = 3

**•Close:**

Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

**Syntax in C language**

#include <fcntl.h>

int close(int fd);

**Parameter**

**• fd :**file descriptor

**Return**

**• 0** on success.

**• -1** on error.

**How it works in the OS**

• Destroy file table entry referenced by element fd of file descriptor table

– As long as no other process is pointing to it!

• Set element fd of file descriptor table to **NULL**

**Program:**

#include<stdio.h>

#include <fcntl.h>

int main()

{

int fd1 = open("foo.txt", O\_RDONLY);

if (fd1 < 0)

{

perror("c1");

exit(1);

}

printf("opened the fd = % d\n", fd1);

// Using close system Call

if (close(fd1) < 0)

{

perror("c1");

exit(1);

}

printf("closed the fd.\n");

}

**Output:**

opened the fd = 3

closed the fd.

**•read**

From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

**Syntax in C language**

size\_t read (int fd, void\* buf, size\_t cnt);

**Parameters**

**• fd:** file descripter

**• buf:** buffer to read data from

**• cnt:** length of buffer

**Returns: How many bytes were actually read**

• return Number of bytes read on success

• return 0 on reaching end of file

• return -1 on error

• return -1 on signal interrupt

Program:

#include<stdio.h>

#include <fcntl.h>

int main()

{

int fd, sz;

char \*c = (char \*) calloc(100, sizeof(char));

fd = open("foo.txt", O\_RDONLY);

if (fd < 0) { perror("r1"); exit(1); }

sz = read(fd, c, 10);

printf("called read(% d, c, 10). returned that"

" %d bytes were read.\n", fd, sz);

c[sz] = '\0';

printf("Those bytes are as follows: % s\n", c);

}

Output:

called read(3, c, 10). returned that 10 bytes were read.

Those bytes are as follows: 0 0 0 foo.

**•Write**

Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT\_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

#include <fcntl.h>

size\_t write (int fd, void\* buf, size\_t cnt);

**Parameters**

**• fd:** file descripter

**• buf:** buffer to write data to

**• cnt:** length of buffer

**Returns: How many bytes were actually written**

• return Number of bytes written on success

• return 0 on reaching end of file

• return -1 on error

• return -1 on signal interrupt

**Program**

#include<stdio.h>

#include <fcntl.h>

main()

{

int sz;

int fd = open("foo.txt", O\_WRONLY | O\_CREAT | O\_TRUNC, 0644);

if (fd < 0)

{

perror("r1");

exit(1);

}

sz = write(fd, "hello geeks\n", strlen("hello\n"));

printf("called write(% d, \"hello\\n\", %d)."

" It returned %d\n", fd, strlen("hello\n"), sz);

close(fd);

}

**Output:**

called write(3, "hello\n", 12). it returned 11