East West University
Department of Computer Science and Engineering
Lab Sheet of CSE325

Lab Task : Introduction to System Call

A. Overview:

- To understand about Linux system calls and their use in processes.
- To write C Programs using the following system calls of the UNIX operating system: fork, exec, getpid, exit, wait, close, stat, opendir, readdir.

B. Learning Outcome:

After this lecture, the students will be able to learn about the Unix System Calls: fork, exec, getpid, exit, wait, close, stat, opendir, readdir

C. Possible Challenges and Their Solutions

Students might face problems in understanding how a system call works. The teacher will give a brief introduction of system call.

D. Acceptance and Evaluation

Students are expected to complete all tasks during the class. This lab will be graded. No home assignment is allowed for the lab.

BACKGROUND:
What are system calls?
System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. A system call is invoked in a variety of ways, dependingon the functionality provided by the underlying processor. In all forms, it is the method used by a process to request action by the operating system.

Some Examples of Windows and UNIX system calls:

	Windows	UNIX

Process Control	CreateProcess()
	ExitProcess()
	WaitForSingleObject()
		fork()
		exit()
		wait()

File Manipulation	CreateFile()
	ReadFile()
	WriteFile()
	CloseHendle()
		open()
		read()
		write()
		close()

Device Manipulation	SetConsoleMode()
	ReadConsole()
	WriteConsole()
		ioctl()
		read()
		write()

Process Control System Calls of UNIX System:

I. fork()	• Has a return value • Parent process => invokes fork() system call • Continue execution from the next line after fork() • Has its own copy of any data • Return value is > 0 //it's the process id of the child process. This value is different from the Parents own process id. • Child process => process created by fork() system call • Duplicate/Copy of the parent process //LINUX • Separate address space • Same code segments as parent process • Execute independently of parent process • Continue execution from the next line right after fork() • Has its own copy of any data • Return value is 0
II. wait ()	• Used by the parent process • Parent's execution is suspended • Child remains its execution • On termination of child, returns an exit status to the OS • Exit status is then returned to the waiting parent process //retrieved by wait () • Parent process resumes execution • #include <sys/wait.h> • #include <sys/types.h>
III. exit()	• Process terminates its execution by calling the exit() system call • It returns exit status, which is retrieved by the parent process using wait() command • EXIT_SUCCESS // integer value = 0 • EXIT_FAILURE // integer value = 1 • OS reclaims resources allocated by the terminated process (dead process) Typically • performs clean-up operations within the process space before returning control back • to the OS • Terminates the current process without any extra program clean-up • Usually used by the child process to prevent from erroneous release of resources • belonging to the parent process

IV. sleep()	• Takes a time value as a parameter (in seconds on Unix-like OS and in milliseconds on Windows OS)
	sleep(2) // sleep for 2 seconds in Unix
	Sleep(2*1000) // sleep for 2 seconds in Window

V. getpid()	• returns the PID of the current process

VI. getppid()	• returns the PID of the parent of the current process

Activity List:

- ➔ All programs are shown in the Green boxes.
- ➔ All the programs should be written in C language.
- ➔ In the blue box write the appropriate output for each program.

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
fork();
printf("hello world \n");
return 0;
}
```

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
int main()
{
int id;
id=fork();
printf("ID: %d hello world \n", id);
return 0;
}
```

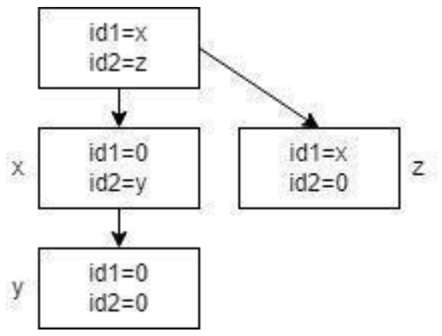
```
int main()
{
int id;
id=fork();
if(id==0)
{   printf("hello from child process \n");   }
else if(id>0)
{   printf("hello from main process \n");    }
 return 0;
}
```

```
int main()
{
int id, i;
id=fork();
printf("CurrentID: %d, parent ID:  %d \n", getpid(), getppid());
return 0;
}
```

```
int main()
{
int id, i;
id=fork();
if(id==0)
{
   for(i=0;i<=5;i++)
   {        printf(" %d",i);       }
 }
else if(id>0)
{
   for(i=6;i<=10;i++)
   {         printf("  %d",i);     }
 }
return 0;
}
```

```
int main()
{
int id, i;
id=fork();
if(id==0)
{sleep(2);}
printf("CurrentID: %d, parent ID:  %d \n", getpid(), getppid());
int response=wait(NULL);
if(response==-1)
{
   printf("No child process to finish\n");
}
else
{
   printf("%d finished its execution", response);
}
return 0;
}
```

Process Tree:



In Lab Exercise:
1. Create a program that creates a child process. The child process prints “I am a child process” 100 times in a loop. Whereas the parent process prints “I am a parent process” 100 times in a loop.

2. Create a program named stat that takes an integer array as an input(delimited by some character such as $). The program then creates 3 child processes each of which does exactly one task from the following.

a) Adds them and print the result on the screen. (done by child 1)

b) Shows the average on the screen. (done by child 2)

c) Print the maximum number on the screen. (done by child 3)