National University of Computer and Emerging Sciences

Lab Manual 03
Operating Systems

Topic: System Calls

Department of Computer Science
FAST-NU, Lahore, Pakistan

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.”

- System Calls provides the **Interface** between a process and the Operating System. These calls are generally available as Assembly language instruction.
- System Calls can also be **made directly** through High Level Language programs for certain systems.
- UNIX System calls can be invoked directly from a **C or C++ program**.

Services Provided by System Calls:

Services provided by system calls include the following:

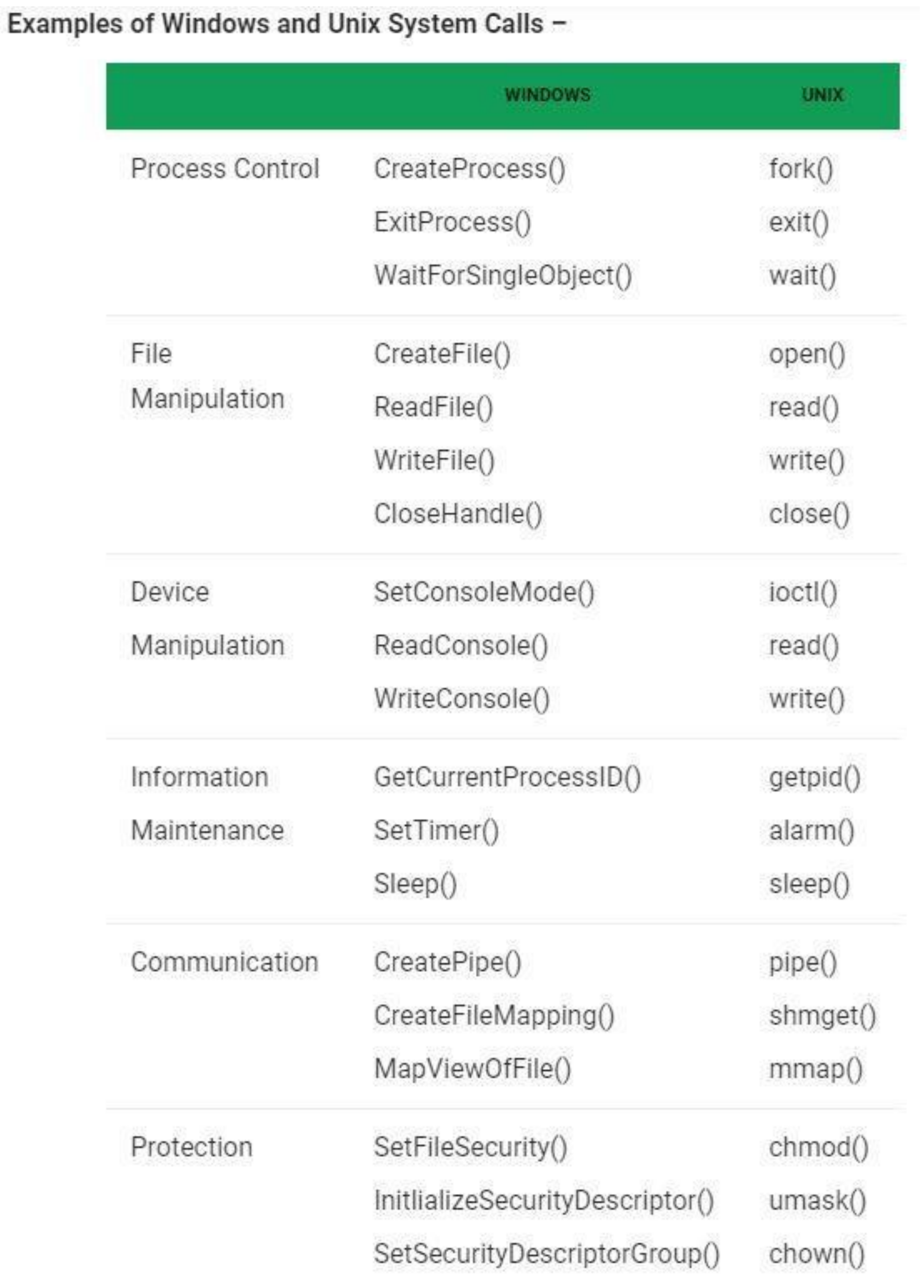
- Process creation and management
- Main memory management
- File Access, Directory and File system management
- Device handling(I/O)
- Protection
- Networking, etc.

Types of System Calls:

There are 5 different categories of system calls

1. Process control
2. File management
3. Device management
4. Information maintenance
5. Communication

Unix and Windows have different system calls as can be seen from the table below



Examples of Windows and Unix System Calls –

	WINDOWS	UNIX
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Reference –http://www.cs.columbia.edu/~jae/4118/L02-intro2-osc-ch2.pdf

Fork ()

"Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process)"

- After a new child process is created, both processes will execute the next instruction following the fork() system call.

- A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process.

```
Syntax:                     fork();
```

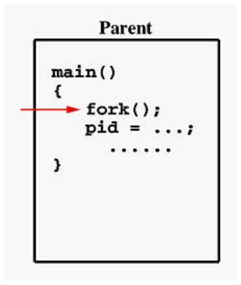
It takes no parameters but returns an integer value.

Code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        printf("Welcome to my Program\n");
      // make a new process which run same program after this instruction
        fork();
       printf("Hello world!\n");
        return 0;
}
```

Output

```
Welcome to my Program
Hello world!
Hello world!
```



Parent Process

Pid (Process identifier) =120

Code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
            printf("Welcome to my
Program\n");
          // make a new process
which run same program after this
instruction
            fork();
           printf("Hello world!\n");
            return 0;
}
```

Parent Process

Pid=120

Code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
            printf("Welcome to my
Program\n");
          // make a new process which
run same program after this
instruction
            fork();
           printf("Hello world!\n");
            return 0;
}
```

Child Process

Pid=121

Code

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

Program Before Fork **Program After Fork**

How to differentiate between a Parent and Child during execution?

Below are different values returned by fork():

- **Negative Value:** creation of a child process was unsuccessful.
- **Zero:** Returned to the newly created child process.
- **Positive value/integer:** Returned to parent or caller. The value contains process ID of newly created child process.

Code

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t a= fork();
if (a==0)
   {
printf("\nHello world!, I am child");
   }
   else if (a>0) {        printf("\nHello
world!, I am Parent");
   } else{        printf("\nBye Bye world!, Error
creating child");
   }
   return 0;
}
```

Output

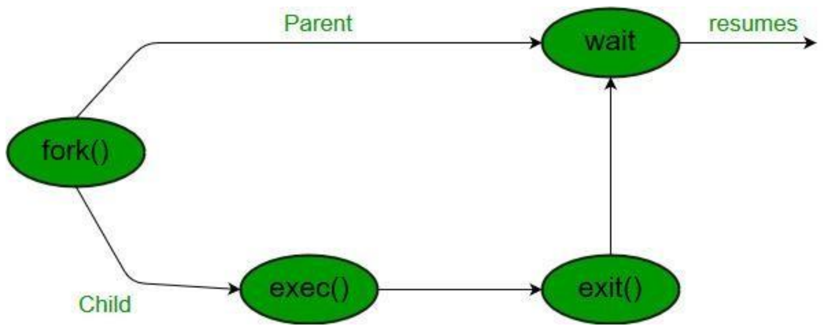
```
Hello world!, I am Parent
Hello world!, I am Child
```

Note: Order of parent and child execution can vary unless controlled

Some more useful System Calls

wait ()

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.



exit()

This call terminates the process and returns the process ID to the parent.

```
/ C program to demonstrate working of wait()
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>
int main()
{
pid_t p, cpid;
printf("main id %d\n",getpid());
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
  exit(0);
}
else if(p>0)
{
  cpid = wait(NULL);
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
else
{
    printf("ERROR");
}
printf("Your quiz is expected in next class\n");
}
```

Output

```
main id 72
before fork
I am child having id 73
My parent's id is 72
My child's id is 73
I am parent having id 72
Your quiz is expected in next class
```

Using getppid() and getpid() to get ID's of parent and the process itself.

getpid()	Whenever this function is called in a process, it returns the ID of that particular process
getppid()	When this function is called in a process it returns the ID of that process's parent.

Task 1: Fork ( ) (Estimated Time: 15 mins)

- Create a C program that takes an integer array of 10 values from the user in parent process through command line arguments.
- The program then creates a child process, and this Child sorts the array in ascending order and displays it on the screen.
- The child also prints its own ID before terminating.

Task 2: Fork ( ) and wait ( ) System Call (Estimated Time: 30 mins)

- Create a C program that takes an integer array of 10 values from the user in parent process through command line arguments.
- Create 2 child processes such that:
  - Child 1 prints its own and its parent's ID on screen.
  - Then it sorts the array in **ascending order** and displays it on the screen.
  - Child 2 prints its own and its parent's ID on screen.
  - Then it sorts the array in **descending order** and displays it on the screen.
- Except input and child-process creation no work should be done by the parent and parent **should terminate after** child process have terminated and also prints its **own and parent's ID**.

Example

```
gcc main.c –o main ./main
5 4 6 9 8 7 1 2 3 5
I am Child 1 (with ID = 120 and Parent’s ID = 115): 1 2 3 4 5 5 6 7 8 9
I am Child 2 (with ID = 121 and Parent’s ID = 115): 9 8 7 6 5 5 4 3 2 1
Parent Process terminating and my ID = 115 and parent’s ID = 100
```

Task 3:

- Write C program that asks the user for input between 1 to 10 via command line arguments.
- Then it creates that many processes such that each process is a parent of exactly one process, except one (last one).
- The last process is not the parent of any process.
- Each child process should print its own and parent’s ID.