National University of Computer and Emerging Sciences



Laboratory Manual

for

Operating Systems Lab

(CL-220)

|  |  |
| --- | --- |
| Course Instructor | Ms. Namra Absar |
| Lab Instructor (s) | Rasaal Ahmad, Mamona Majid |
| Section | B |
| Semester | Fall 2023 |

Department of Computer Science

FAST-NU, Lahore, Pakistan

**Objectives:**

* Understand Process control system calls
* Why we need system call
* Difference between windows and linus system calls
* Working with Linux system calls
* Creating process using Fork system call
* MakeFile Utility

Important Note:

* Comment your code intelligently.
* Indent your code properly.
* Use meaningful variable names.
* Use meaningful prompt lines/labels for input/output.
* Use meaningful project and C++ file name

**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, depending on 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.

(OS Concepts 9th Edition)

**Services Provided by System Calls :**

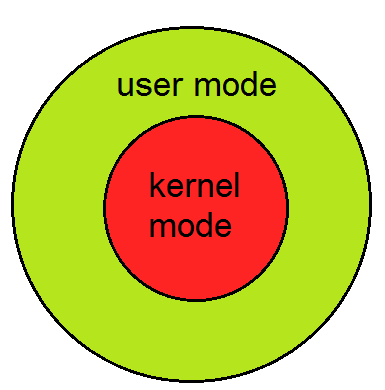
1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

**Types of System Calls :** There are 5 different categories of system calls –

* 1. **Process control:** end, abort, create, terminate, allocate and free memory.
  2. **File management:** create, open, close, delete, read file etc.
  3. Device management
  4. Information maintenance
  5. Communication

# Introduction to System Calls

To understand system calls, first one needs to understand the difference between **kernel mode** and **user mode** of a CPU. Every modern operating system supports these two modes.



**Modes supported by the operating system**

### Kernel Mode

* When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
* Hence kernel mode is a very privileged and powerful mode.
* If a program crashes in kernel mode, the entire system will be halted.

### User Mode

* When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
* In user mode, if any program crashes, only that particular program is halted.
* That means the system will be in a safe state even if a program in user mode crashes.
* Hence, most programs in an OS run in user mode.

**Command:**

1. Ps: If you run **ps command** without any arguments, it displays processes for the current shell. This command stands for 'Process Status'. It is similar to the "Task Manager" that pop-ups in a Windows Machine when we use Cntrl+Alt+Del. This command is similar to 'top' command but the information displayed is different.
2. Top: The easiest way to find out what processes are running on your server is to run the top command:
3. Kill: The easiest way to find out what processes are running on your server is to run the top command:
4. Jobs: List background processes
5. & :Run the command in the background
6. Nice: Starts a process with a given priority

* Often, you will want to adjust which processes are given priority in a server environment.
* Some processes might be considered mission critical for your situation, while others may be executed whenever there might be leftover resources.
* Linux controls priority through a value called niceness.
* Nice values can range between "-19/-20" (highest priority) and "19/20" (lowest priority) depending on the system.

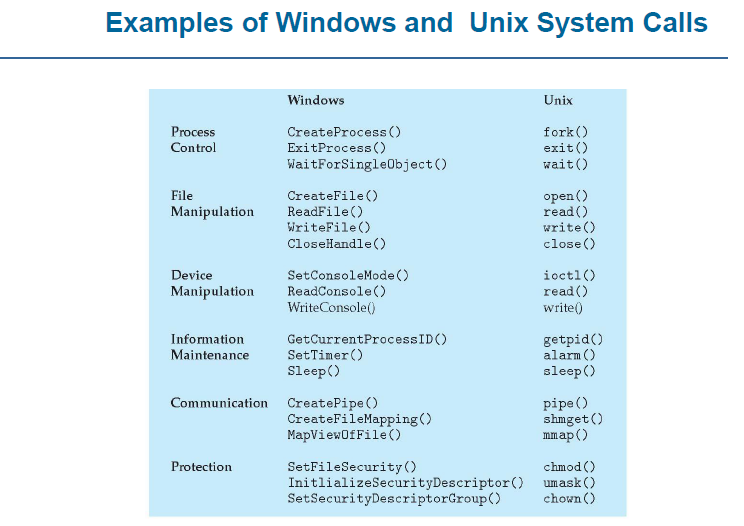
1. Renice: Changes priority of an already running process
   1. **What is a Process?**

An instance of a program is called a Process. In simple terms, any command that you give to your Linux machine starts a new process.

Types of Processes:

* Foreground Processes: They run on the screen and need input from the user. For example Office Programs
* Background Processes: They run in the background and usually do not need user input. For example Antivirus.

Init is the parent of all Linux processes. It is the first process to start when a computer boots up, and it runs until the system shuts down. It is the ancestor of all other processes.



1. Process Creation:

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

1. **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

1. **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>

1. **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
* \_exit()
* Terminates the current process without any extra program clean-up
* Usually used by the child process to prevent from erroneously release of resources belonging to the parent process

1. **execlp() is a version of exec()**

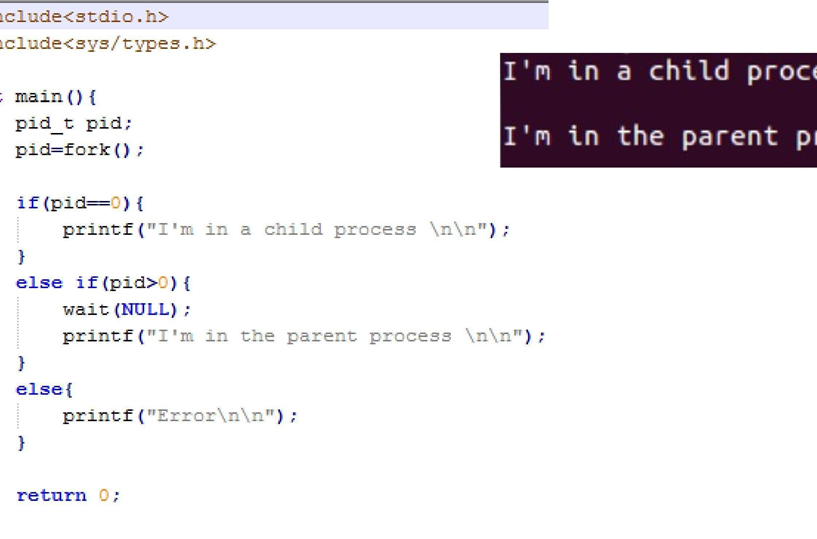
* **exec()**
* The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program.an executable file
* Called by an already existing process //child process
* Replaces the previous executable //overlay
* Has an exist status but cannot return anything (if exec() is successful) to the program that made the call //parent process
* Return value is -1 if not successful
* Overlay => replacement of a block of stored instructions or data with another int execlp(char const \*file\_path, char const \*arg0, );
* Arguments beginning at arg0 are pointers to arguments to be passed to the new process.
* The first argument arg0 should be the name of the executable file.
* Example
* **execlp( /bin/ls , ls ,NULL) //lists contents of the directory**
  1. **but exec or execlp is a system call which overwrites an already existing process (calling process), so if you want to execute some code after execlp system call, then write this system call in a child process of an existing process, so it only overwrite child process.**
* Header file used -> unistd.h

1. **Information Maintenance**
2. **sleep()**

* Process goes into an inactive state for a time period
* Resume execution if
* Time interval has expired
* Signal/Interrupt is received
* Takes a time value as 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 Windows

1. **getpid() // returns the PID of the current process**

* getppid() // returns the PID of the parent of the current process
* Header files to use
* #include <sys/types.h>
* #include <unistd.h>
* getppid() returns 0 if the current process has no parent

**Example**

Command line arguments are a way to pass data to the program. Command line arguments are passed to the main function. Suppose we want to pass two integer numbers to main function of an executable program called a.out. On the terminal write the following line:

./a.out 1 22

./a.out is the usual method of running an executable via the terminal. Here 1 and 22 are the numbers that we have passed as command line argument to the program. These arguments are passed to the main function. In order for the main function to be able to accept the arguments, we have to change the signature of main function as follows:

int main(int argc, char \*arg[]);

* argc is the counter. IT tells how many arguments have been passed.
* arg is the character pointer to our arguments.

argc in this case will not be equal to 2, but it will be equal to 3. This is because the name ./a.out is also passed as command line argument. At index 0 of arg, we have ./a.out; at index 1, we have 1; and at index 2, we have 22. Here 1 and 22 are in the form of character string, we have to convert them to integers by using a function atoi. Suppose we want to add the passed numbers and print the sum on the screen:

cout<< atoi(arg[1]) + atoi(arg[2]);

**Question No 1**

Write a C++ which perform following tasks.

Create a parent and child process using the fork command where the child tells if the number is odd or not and parent process will write the number on screen.