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
Section	B
Semester	Fall 2023

Department of Computer Science

FAST-NU, Lahore, Pakistan

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:

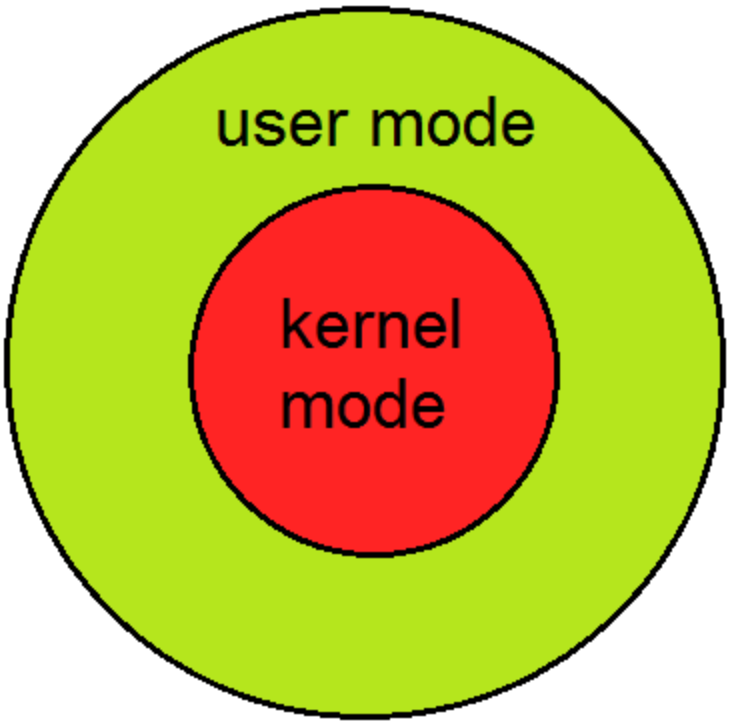
1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

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

1 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

1.1.1 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.

1.1.2 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.
7) Renice: Changes priority of an already running process

a. 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.

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()

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.

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
- _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

IV. 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**
    a. **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

2. Information Maintenance

i. 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

ii. 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

3. Pipes

Ordinary pipes allow two processes to communicate in standard producer consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used with each pipe sending data in a different direction.

References: Operating System concepts Page no 142 section 3.6.3.1

A pipe has a read end and a write end.

Data written to the write end of a pipe can be read from the read end of the pipe.

4. Creating a pipe

On UNIX and Linux systems, ordinary pipes are constructed using the function.

- int pipe(int fd[2]) -- creates a pipe
- returns two file descriptors, fd[0], fd[1].
- fd[0] is the read-end of the pipe
- fd[1] is the write-end.

- fd[0] is opened for reading,
- fd[1] for writing. pipe() returns 0 on success, -1 on failure and sets errno accordingly.
- The standard programming model is that after the pipe has been set up, two (or more) cooperative processes will be created by a fork and data will be passed using read() and write().
- Pipes opened with pipe() should be closed with
- close(int fd).

Reference: http://linux.die.net/man/2/pipe

Question No 1 (System Calls)

Create a C++ program that demonstrates the use of the wait () system call. The program should fork exactly 4 child processes and ensure that the parent process waits for all child processes to complete before proceeding. Provide code and explain the purpose of using wait () in this context.

Question No 2 (System Calls)

Write a C++ program which uses fork () system-call to create a child process. The child process prints the contents of the current directory, and the parent process waits for the child process to terminate.

Question No 3 (System Calls)

Invoke at least 4 commands from your programs, such as **cp, mkdir, rmdir**, etc (The calling program must not be destroyed)

Question No 4 (Pipes)

Create a C++ program that forks a child process. The parent process reads an integer from the user and sends it to the child process through a pipe. The child process receives the integer, calculates its square root, and sends the result back to the parent process through another pipe. Display both the original integer and its square root in the parent process.