

National University of Computer and Emerging Sciences



Laboratory Manual

For

Operating Systems Lab

(BSE-5C)

Course Instructor	Mr. Mubashar Hussain
Lab Instructor	Haiqa Saman
Section	BSE-5C
Semester	Fall 2024

Department of Computer Science
FAST-NU, Lahore, Pakistan

Objectives:

- Understand Process control system calls.
- Why do we need a system call?
- The difference between Windows and Linux system calls
- Working with Linux system calls
- Creating process using Fork system call

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 :

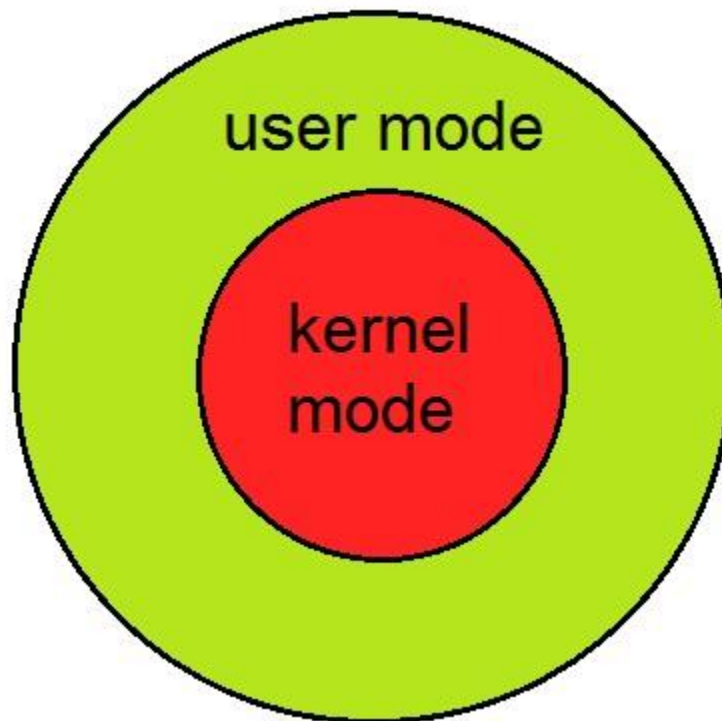
1. Process creation and management.
2. Main memory management
3. File Access, Directory, and File System Management.
4. Device handling(I/O)
5. Protection
6. Networking, etc.

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 files, 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 Kernel Mode

- When the 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.

1.2 User Mode

- When the 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 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 the **ps command** without any arguments, it displays processes for the current shell. This command stands for 'Process Status'. It is like the "Task Manager" that pops up in a Windows Machine when we use Cntrl+Alt+Del. This command is like the '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
- 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 the priority of an already running process.

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

2.1 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.

2.1.1 fork()

- Has a return value?
- Parent process => invokes fork() system call
- Continue execution from the next line after fork()
- Is it a copy of any data?
- The return value is > 0 //it's the process ID of the child process. This value is different from the Parent's process ID.
- Child process => process created by fork() system call
- Duplicate/Copy of the parent process //LINUX.
- Separate address space
- The same code segments as the parent process
- Execute independently of the parent process.
- Continue execution from the next line right after fork()
- Has it copied any data?
- The return value is 0

2.1.2 wait ()

- Used by the parent process.
- The parent's execution is suspended.
- The child remains its execution.
- On termination of a child, returns an exit status to the OS.
- Exit status is then returned to the waiting parent process //retrieved by wait ()

- The parent process resumes execution.
- `#include <sys/wait.h>`
- `#include <sys/types.h>`

2.1.3 `exit()`

- The process terminates its execution by calling the `exit()` system call.
- It returns exit status, which is retrieved by the parent process using the `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 to the OS.
- `_exit()`
- 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.

2.1.4 `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 existing status but cannot return anything (if `exec()` is successful) to the program that made the call //parent process?
- The 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 that 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 overwrites child process.**
- Header file used -> `unistd.h`

2.2 Information Maintenance

2.2.1 `sleep()`

- The process goes into an inactive state for a time.
- Resume execution if.
- The time interval has expired.
- Signal/Interrupt is received.
- 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 Windows.

2.2.2 `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.

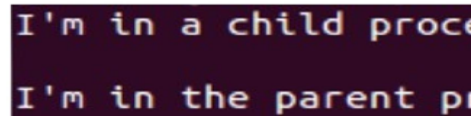
Example

```
#include<stdio.h>
#include<sys/types.h>

int main() {
    pid_t pid;
    pid=fork();

    if(pid==0) {
        printf("I'm in a child process \n\n");
    }
    else if(pid>0) {
        wait(NULL);
        printf("I'm in the parent process \n\n");
    }
    else{
        printf("Error\n\n");
    }

    return 0;
}
```



```
I'm in a child process
I'm in the parent process
```

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 the 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 a command-line argument to the program. These arguments are passed to the main function. For the main function to be able to accept the arguments, we must change the signature of the 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 a 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 strings, we must convert them to integers by using a function `atoi`. Suppose we want to add the past numbers and print the sum on the screen:

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

In Lab Tasks

Forks Dry Run

Code 1:

```
int main() {
    pid_t pid = fork();
    if (pid == -1) {
        std::cerr << "Error in fork." << std::endl;
        return 1;
    }
    if (pid == 0) {
        // Child process
        std::cout << "Child process." << std::endl;
        exit(0);
    } else {
        // Parent process
        std::cout << "Parent process." << std::endl;
        wait(NULL);
    }
    return 0;
}
```

Code 2:

```
int main() {
    for (int i = 0; i < 3; ++i) {
        pid_t pid = fork();
        if (pid == -1) {
            std::cerr << "Error in fork." << std::endl;
            return 1;
        }
        if (pid == 0) {
            // Child process
            std::cout << "I am child" << i << std::endl;
            exit(0);
        } else {
            // Parent process
            wait(NULL);
        }
    }
    return 0; }
```


Code 3:

```
int main() {  
    if (fork() || fork()) {  
        std::cout << "Process Created" <<std::endl;  
    } else {  
        wait(NULL);  
    }  
    return 0;  
}
```

Code 4:

```
int main() {  
    if (fork() && fork()) {  
        std::cout << "Process Created" <<std::endl;  
    } else {  
        wait(NULL);  
    }  
    return 0;  
}
```

Code 5:

```
int main() {  
    if (fork() || (fork() && fork()) || fork()) {  
        std::cout << "Process Created" <<std::endl;  
    } else {  
        wait(NULL);  
    }  
    return 0;  
}
```

Code 6:

```
int main() {  
    if (fork() && (fork() || fork())) {  
        std::cout << "Process Created" <<std::endl;  
    } else {  
        wait(NULL);  
    }  
    return 0;  
}
```

System Calls (Process Control & Information Maintenance):

Question 1: Write a C++ that performs the following tasks.

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

Question 2: Write a C++ program that takes two numbers as command line arguments. The program should create two child processes:

Child 1: Calculate and print the square of the first number.

Child 2: Determine and print whether the second number is a prime number.

- Also prints the PID of child processes.

Question 3: Create a stat program that takes an integer array as a command line argument (delaminated by some character such as \$). The program then creates 3 child processes each of which does exactly one task from the following:

- Adds them and print the result on the screen. (Done by child 1)
- Shows average on the screen. (Done by child 2)
- Prints the maximum number on the screen. (Done by child 3)