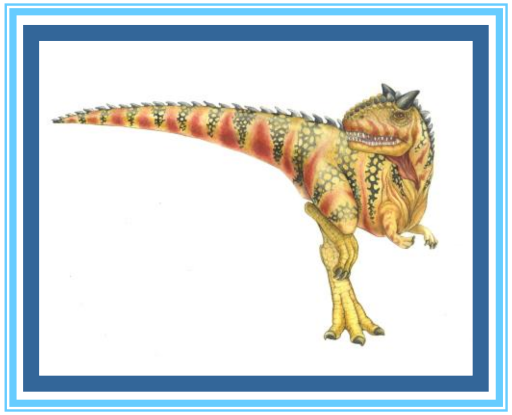
Chapter 1: Introduction to Operating Systems



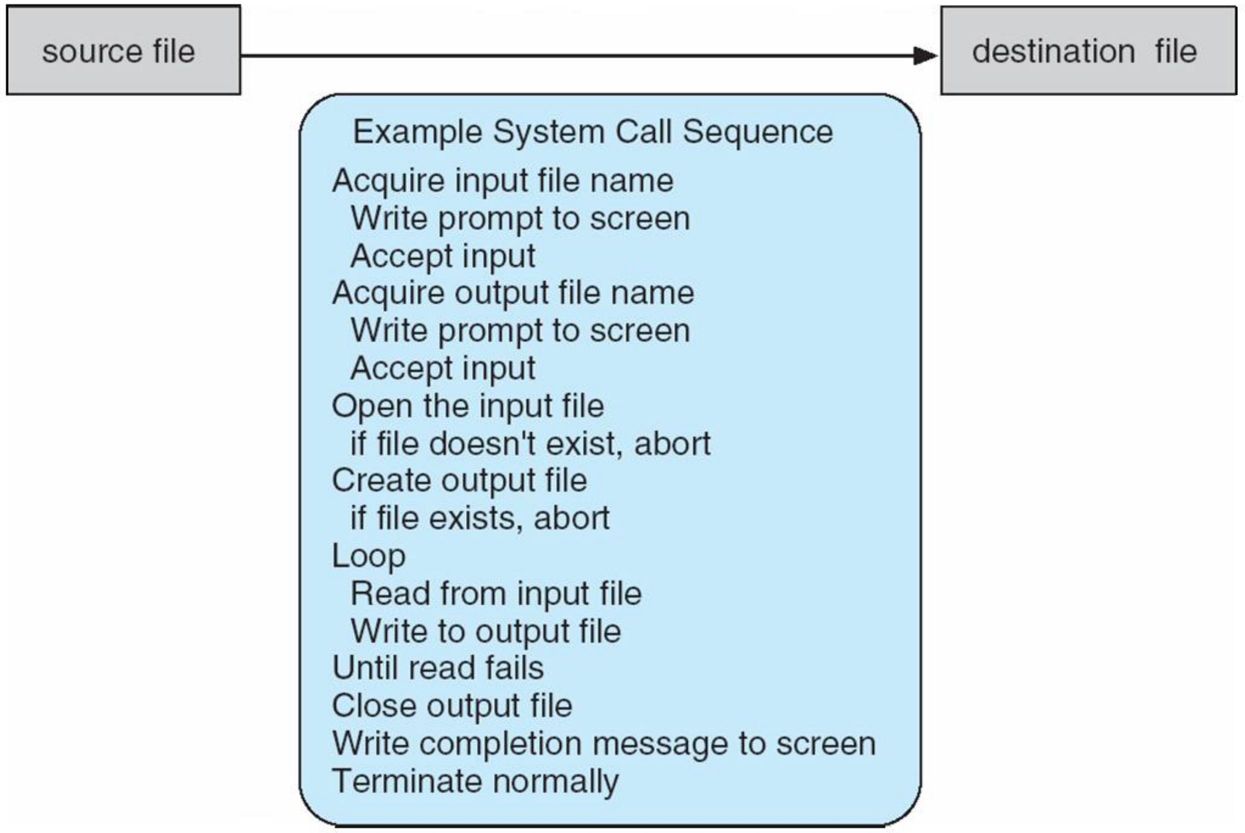
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

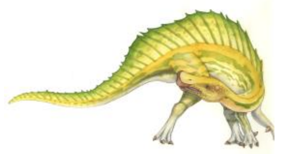
Note that the system-call names used throughout this text are generic

Example of System Calls

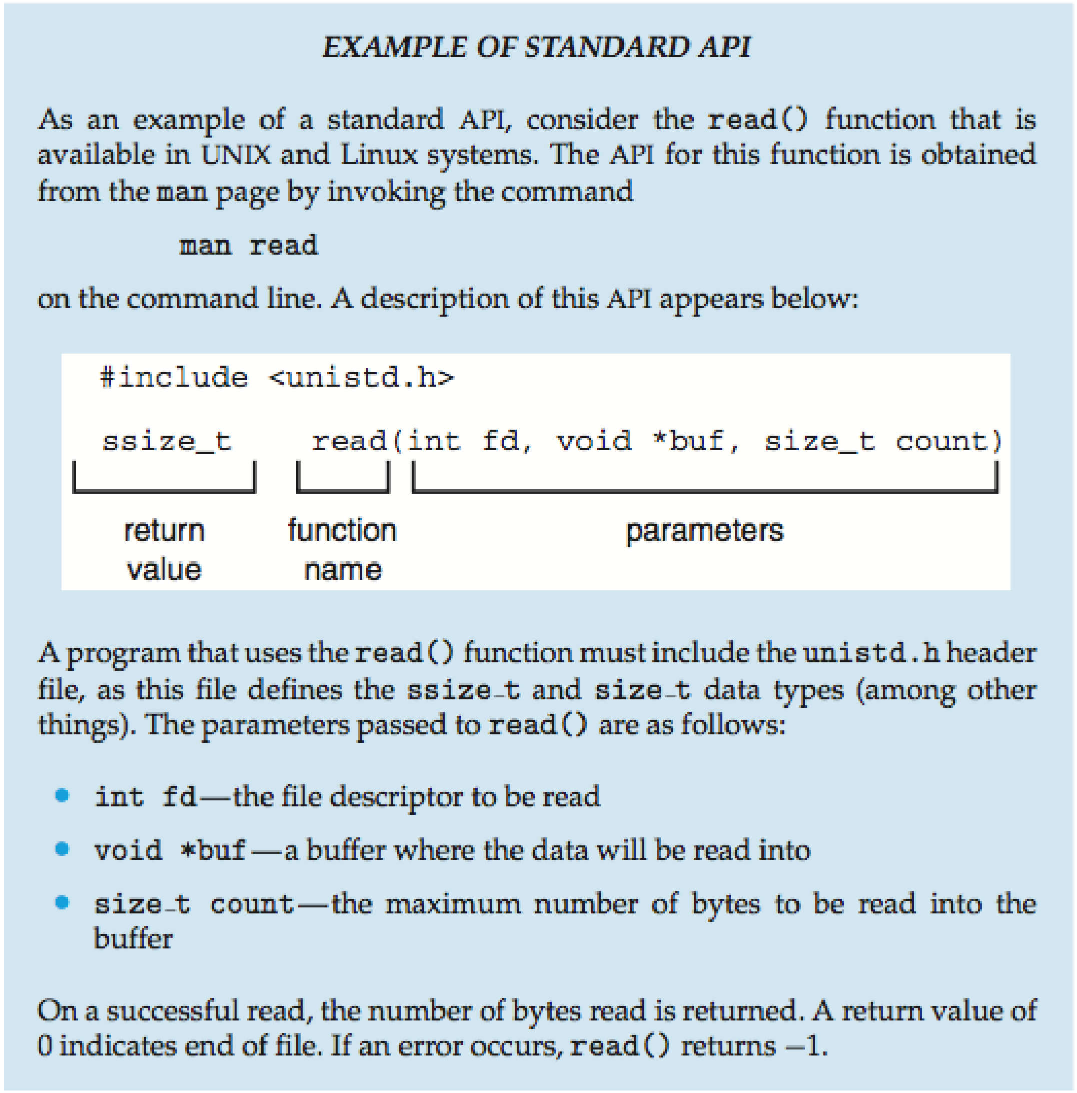
- System call sequence to copy the contents of one file to another file



```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```



Example of Standard API



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>
```

$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\quad \underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

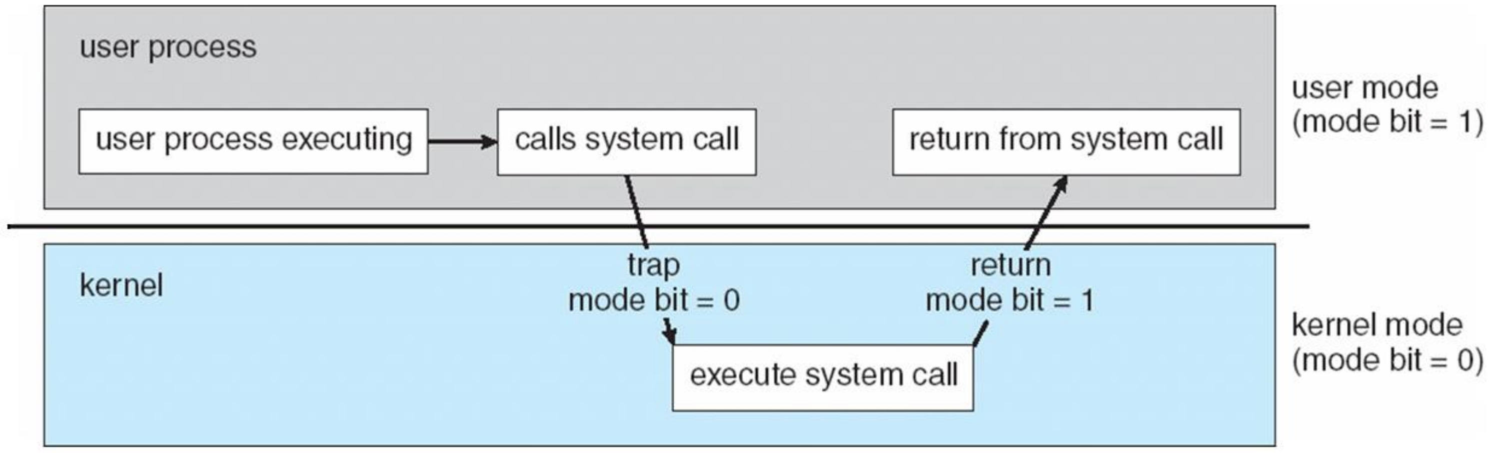
A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

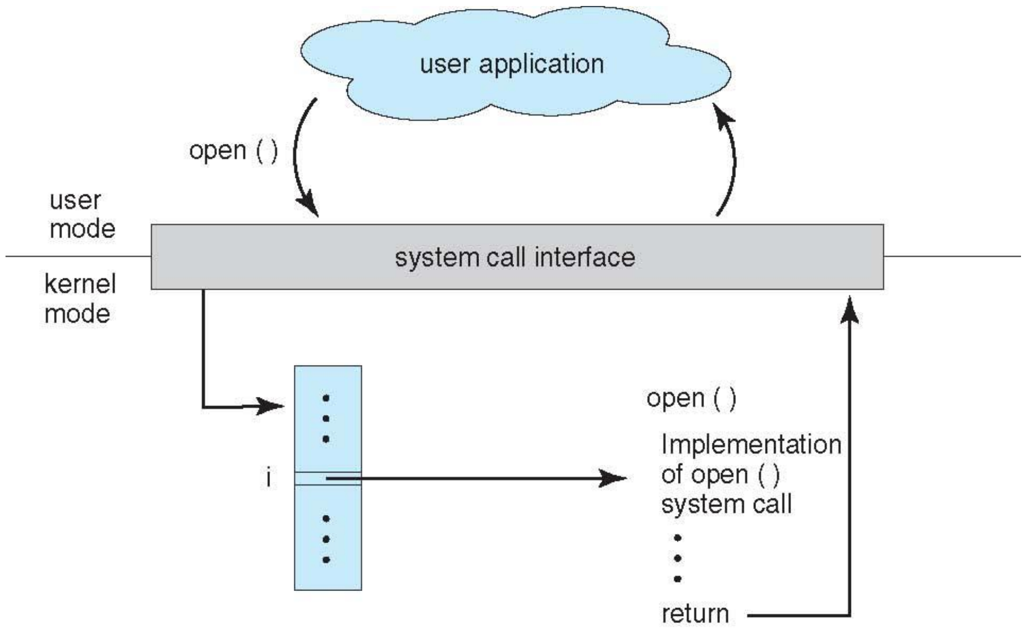
System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values



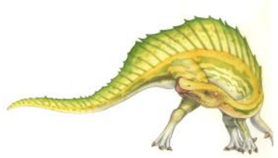
API – System Call – OS Relationship

- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)



Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

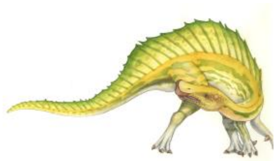


Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

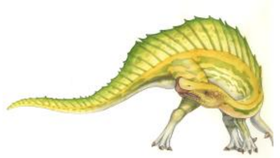


Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

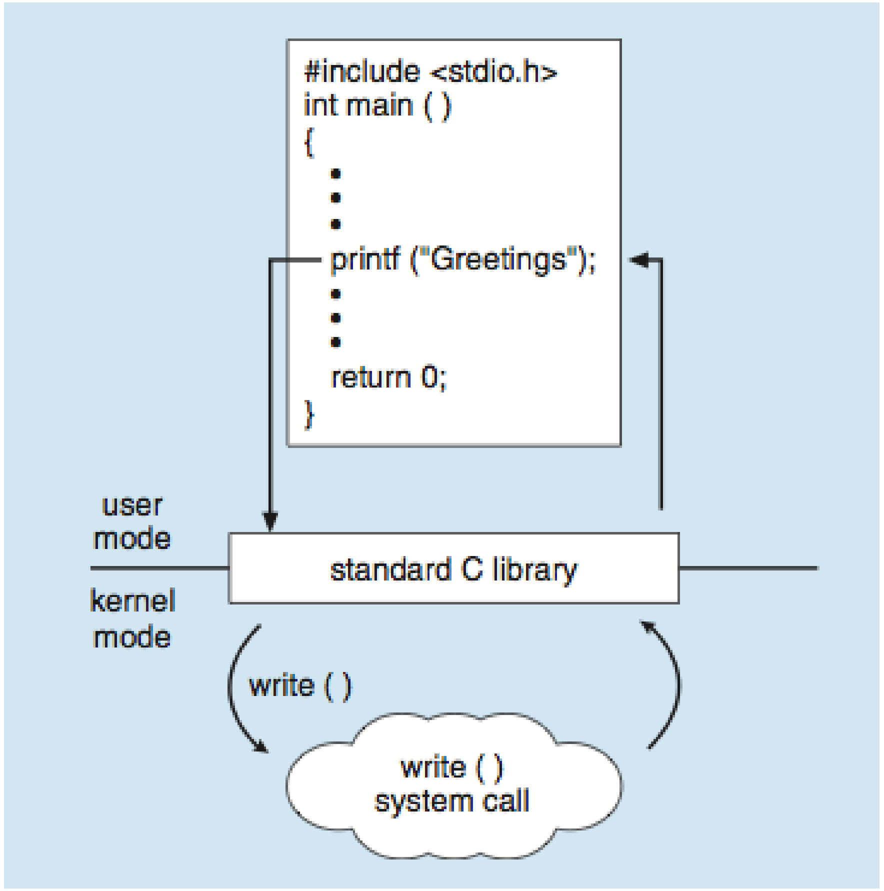
Examples of Windows and Unix System Calls

	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()



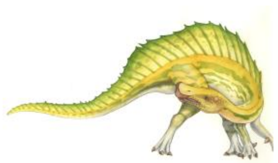
Standard C Library Example

- C program invoking printf() library call, which calls write() system call



exec System Calls

- When you use the shell to run a command (ls, say) then at some point the shell will execute a fork() call to get a new process running. Having done that, how does the shell then get ls to run in the child process instead of the duplicate copy of the shell, which is what will be running immediately after the fork() call?
- The solution in this case is to use an **exec() system call**
  - execl()
  - execlp()
  - execvp()



execl()

- int execl(char *patbname, char *arg0, ...);
  - The pathname is the full path to the command to execute.
  - Followed by a variable length list of pointers to character strings.
  - The list of pointers in execl() should be terminated by a NULL pointer.

execl() Example

```
#inciude <stdio.h>
#include <unistd.h>

        main()
        {
                execl("/bin/ls", "ls", "-l", NULL);
                printf("Can only get here on error\n");
        }
```

- The first parameter to execl() in this example is the full pathname to the ls command.
- The rest of the execl() parameters provide the strings to which the argv array elements in the new program will point.
- E.g., the string ls pointed to by its argv[0], and the string -l pointed to by its argv[1].

exec() System Calls

- Six exec() variants that you can use
  - int execl(pathname, argo, ..., argn, 0);
  - int execv(pathname, argv);
  - **int execlp(cmdname, arg0, ..., argn, 0);**
  - **int execvp(cmdname, argv);**
  - int execle(patbname, arg0, ..., arga, 0, envp);
  - int execve(pathname, argv, envp);

In Class Practice 1

- Write a simple program to execute the “ls -l” command using execvp.

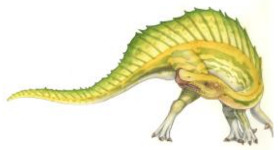
File Operations

- #include <stdio.h>
- #include <stdlib.h>

- FILE *source, *destination;
- source = fopen("source.txt", "r");
- destination = fopen("destination.txt", "w");

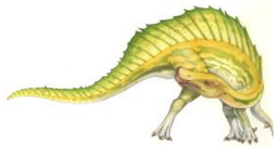
- fgetc(source)
- fputc(character, destination)

- fclose(source) or fclose(destination)



In Class Practice 2

- Write a simple program to copy a source file “source.txt” to an output file “destination.txt”.



Project 1

- Question 1: MyCompress.c

- One possible solution:

```
for(i=0;i<number;i++) {
  if((*(data+i)-*(data+i+1))==0)
        counter++;
  else{
        if(counter<16)
              //print them out directly
        else
              //print out as the compressed format
        counter=1;
  }
}
```