BITS, PILANI – K. K. BIRLA GOA CAMPUS

Operating Systems

by

Mrs. Shubhangi Gawali

Dept. of CS and IS

General-System Architecture

Why we need protection?

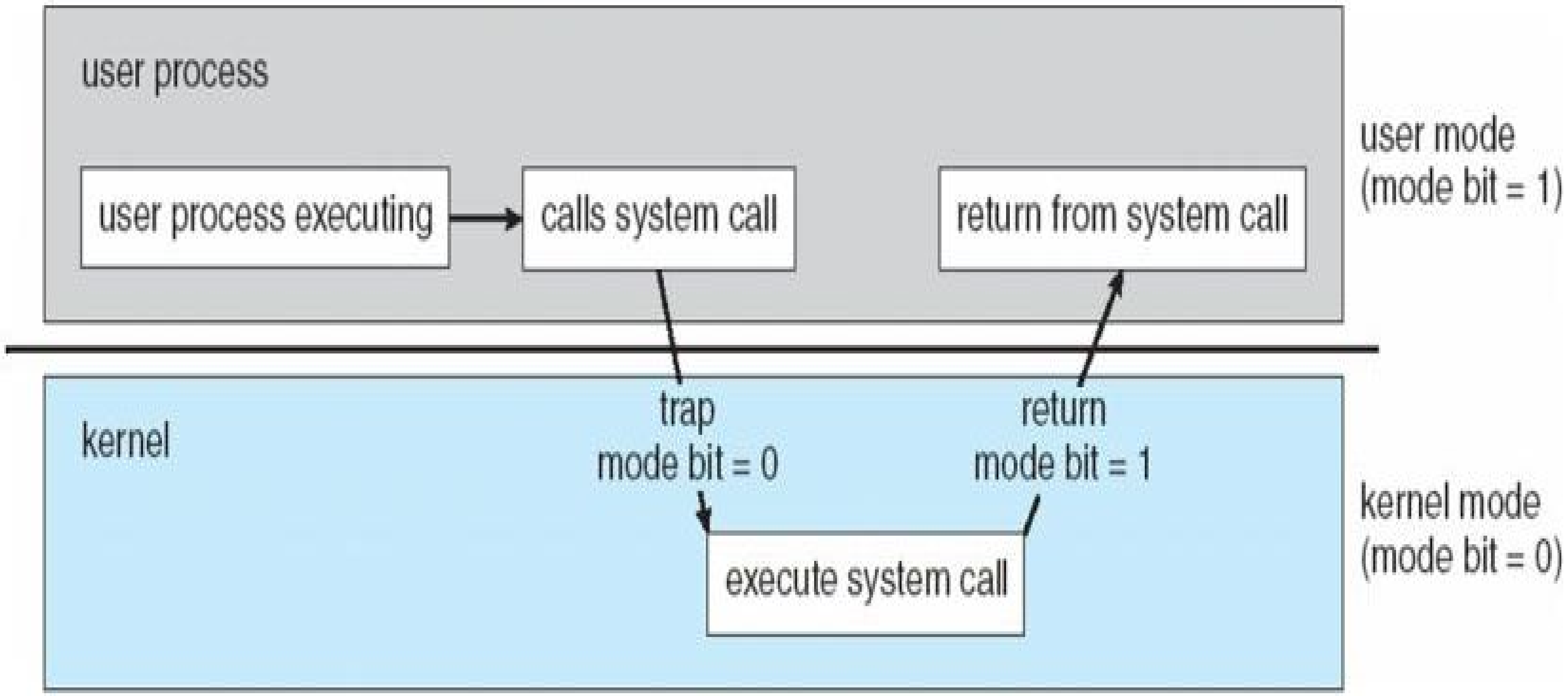
- In single task / programming environment
  - To protect OS from incorrect program
- In multiprogramming / multitasking environment
  - To protect OS and other programs from incorrect program
- Given the I/O instructions are privileged, how

does the user program perform I/O?

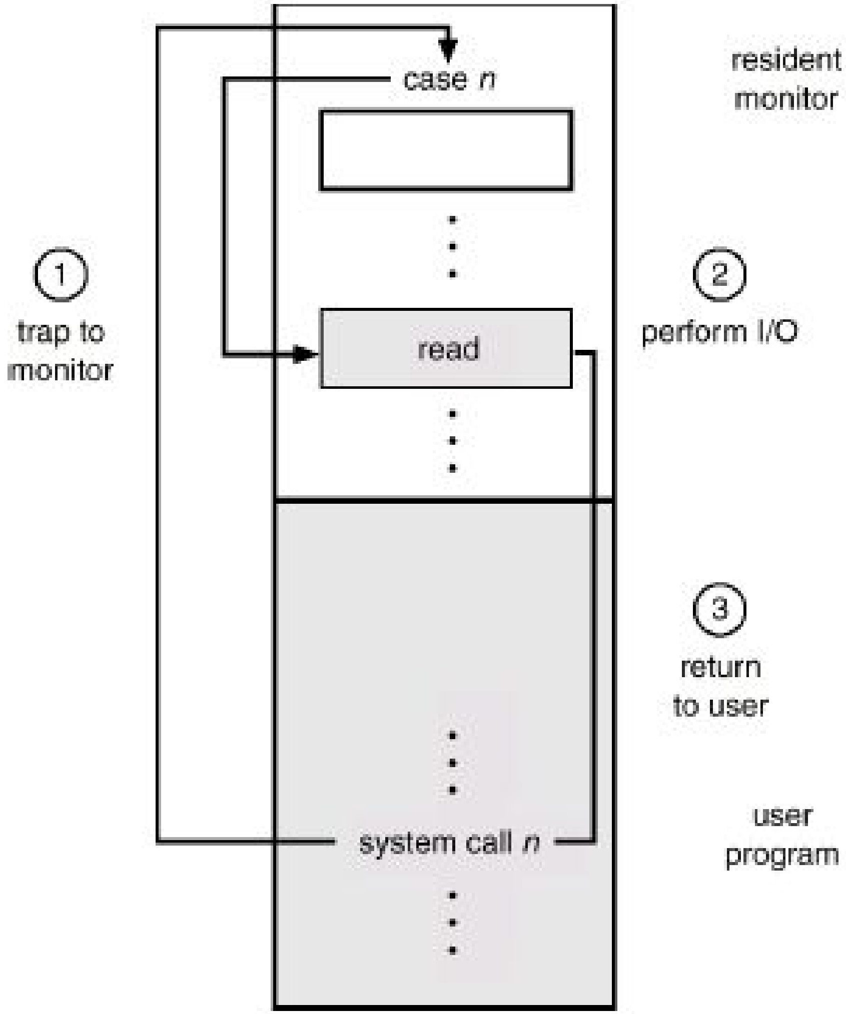
Operating-System Operations

- Dual-mode operation allows OS to protect itself and other system components
- User mode and kernel mode
- Mode bit provided by hardware

Transition from User to Kernel Mode



Use of A System Call to Perform I/O



User
Shell, compiler, interpreter and system libraries
System call interface to the kernel
Kernel
Kernel interface to HW
HW

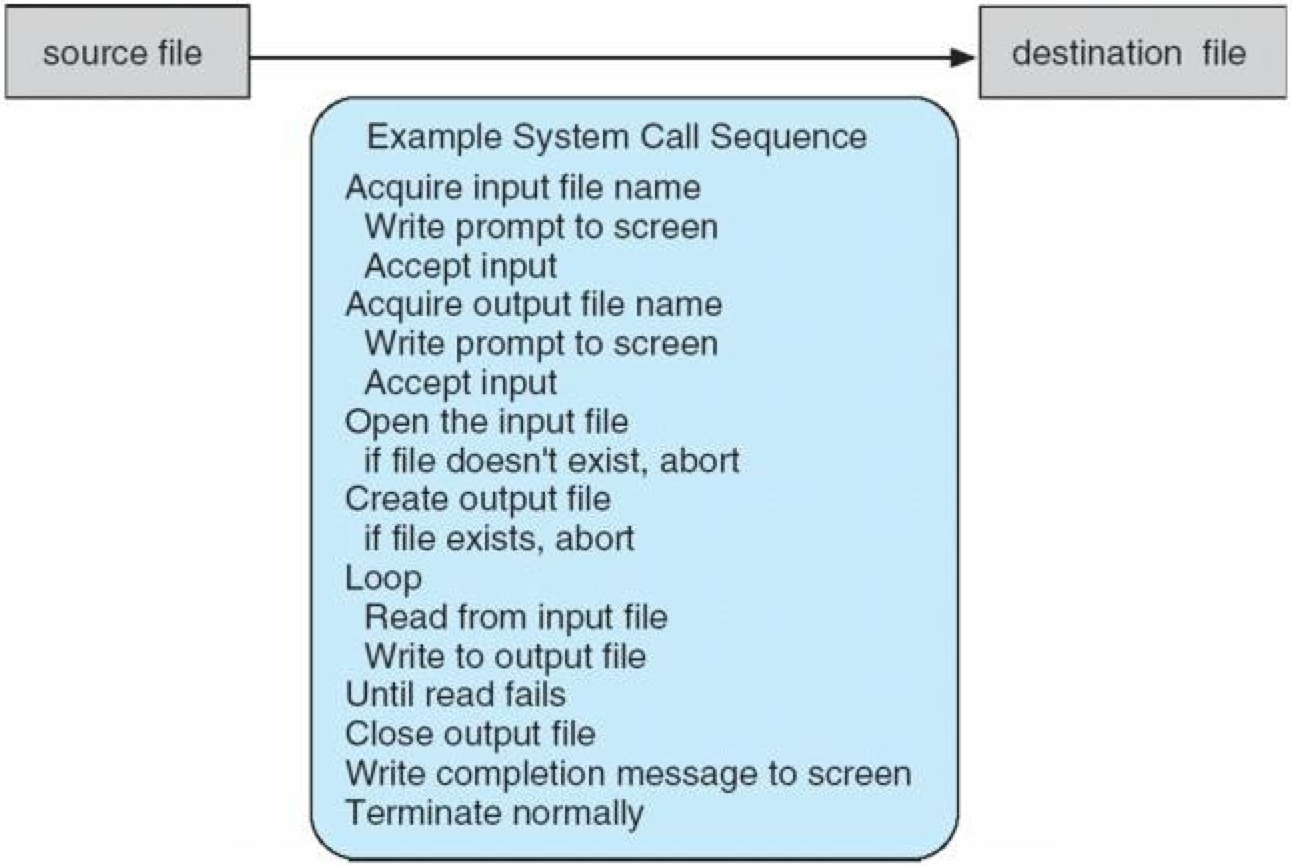
How to communicate ?

How to communicate between two modes?

- System calls have always been the means through which user space programs can access kernel services. (Typically written in a high-level language (C or C++)
- System call is the only legal entry point to the kernel
- System call provides an abstract hardware interface for user space
- Application need not worry about type of disk, media and file system in use
- System call ensures system stability and security.
- Kernel can keep track of application’s activity

Example of System Calls

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



System Calls

- System call – the method used by a process to request action by the operating system provide the interface between a running program and the operating system

- It is the mechanism used by an application program to request service from the operating system

- Often use a special machine code instruction (i.e. software interrupt or trap) which causes the processor to transfer control to a specific location in the interrupt vector (kernel code)

- Usually takes the form of a trap to a specific location in the interrupt vector

System Call Implementation

- The process fills the registers with the appropriate values and calls a special instruction which jumps to a previously defined location in the kernel (under Intel CPUs, this is done by means of interrupt 0x80)

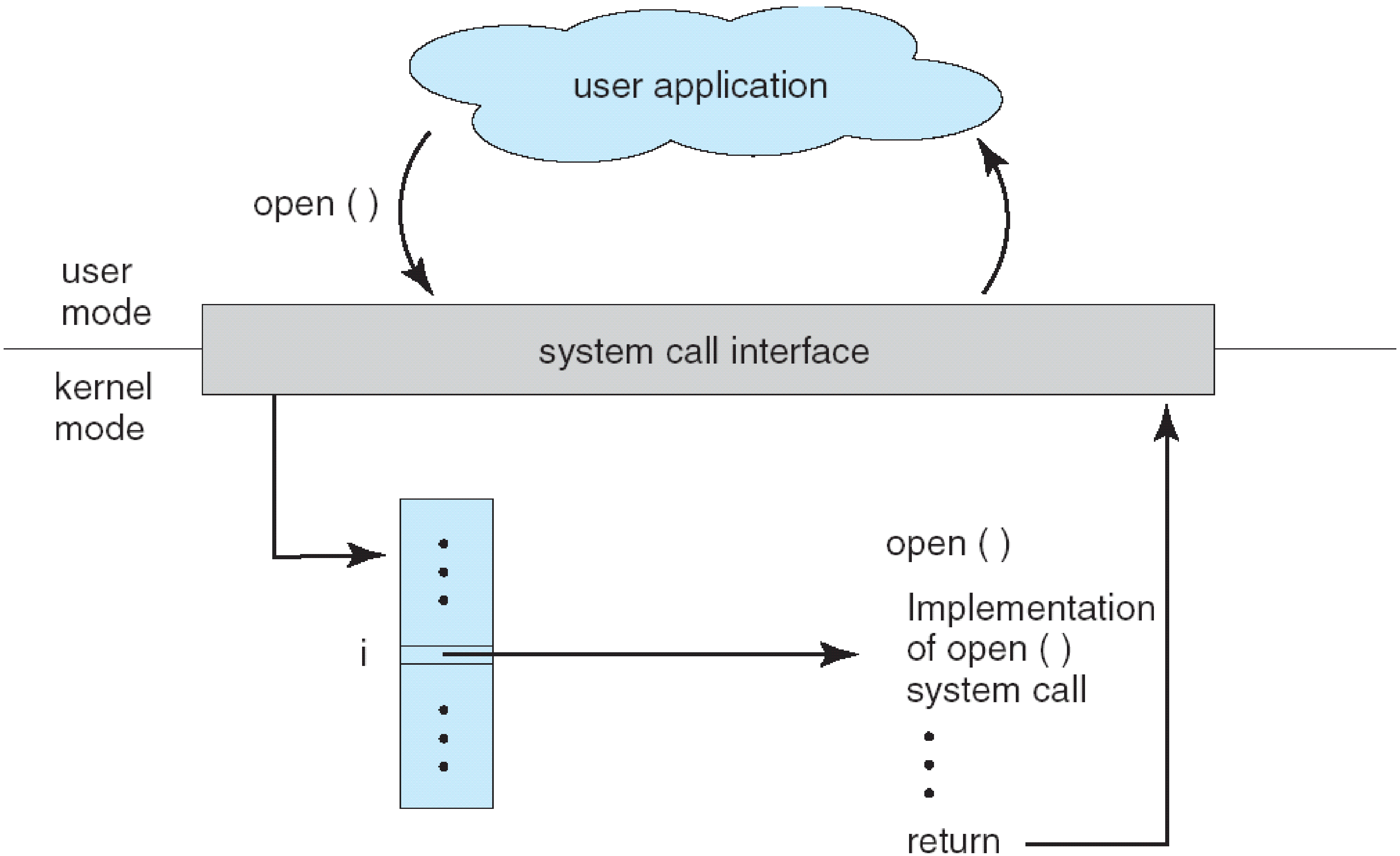
- Control passes through the interrupt vector to a service routine in the OS, and the mode bit is set to monitor mode.

- The monitor mode verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

System Call Implementation

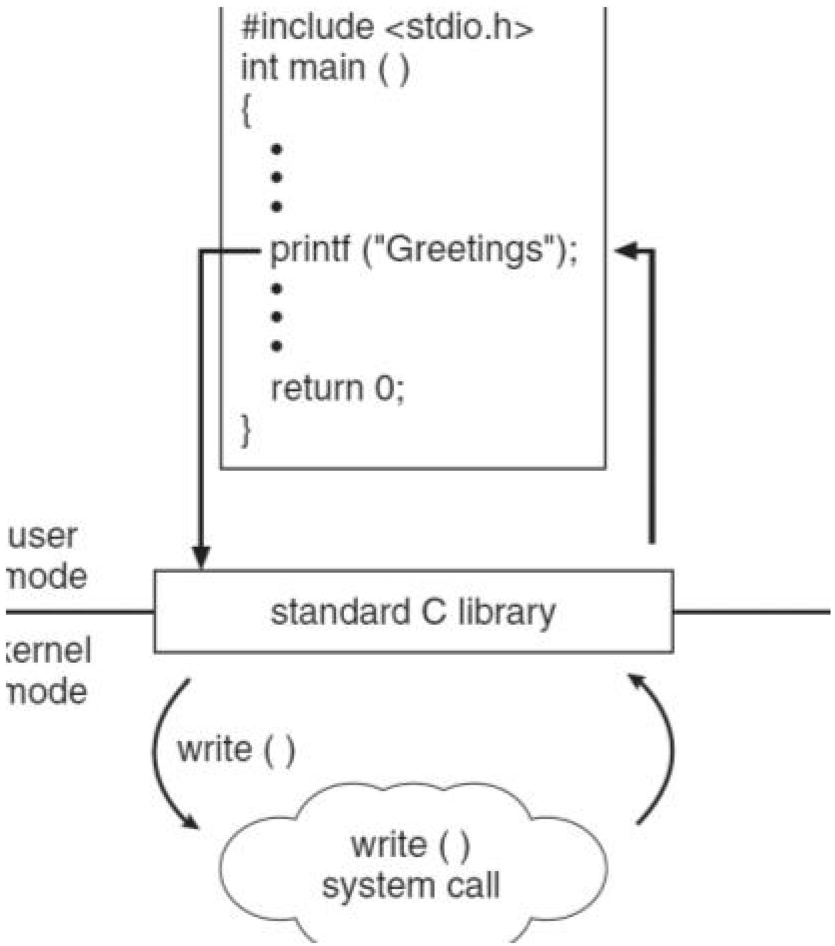
- Typically, a number associated with each system call.
- System-call interface maintains a table indexed according to these numbers.
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values.
- The caller need not know any thing 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)

API –System Call –OS Relationship



Standard C Library Example

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



System Call Parameter Passing

- Often, more information is required than simply identity of desired system call.

- Exact type and amount of information vary according to OS and call.

- Most of the system calls require one or more parameter to be passed to them

- Parameters are stored in registers EBX, ECX, EDX, ESI and EDI (if the parameters are less than six)

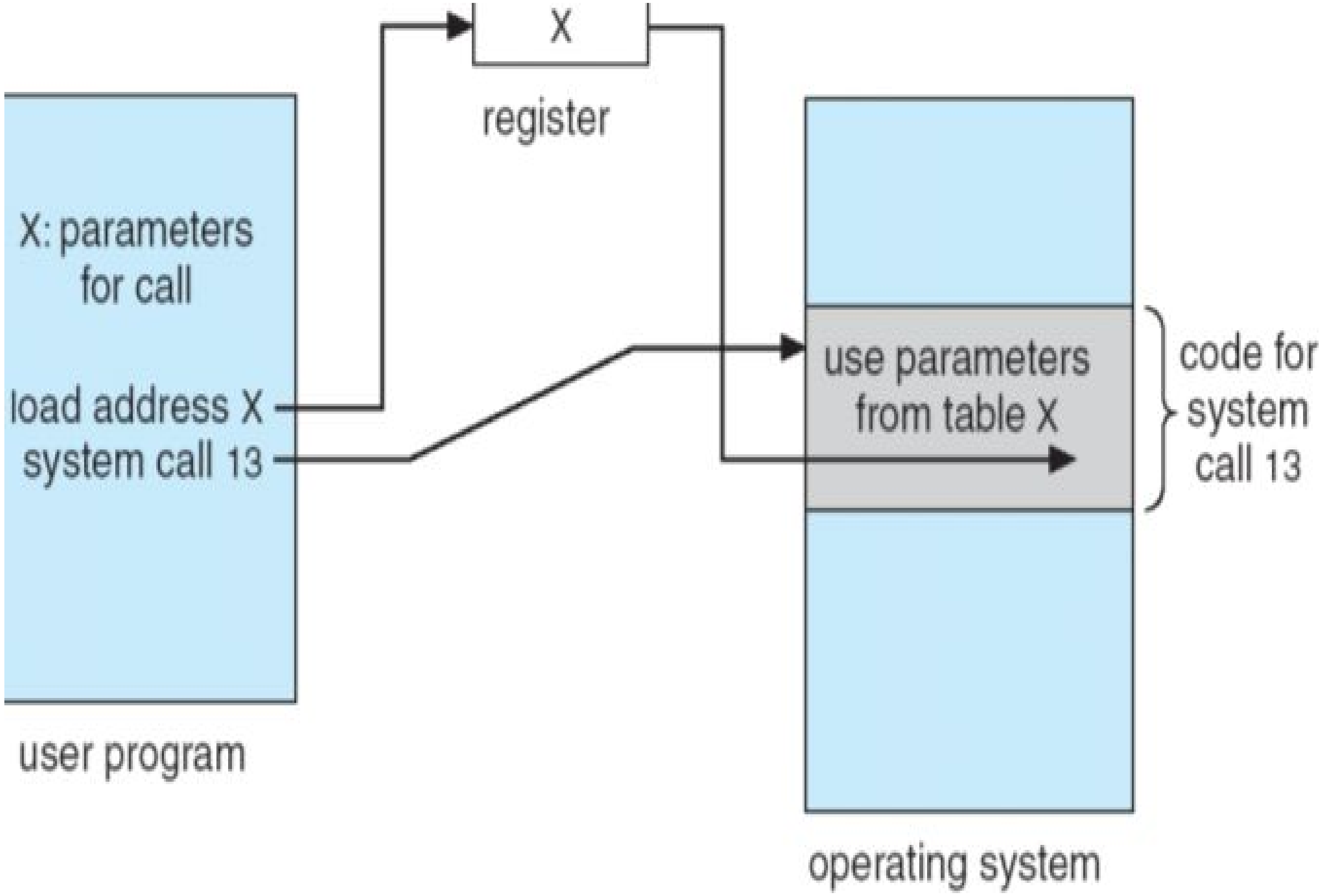
- If number of parameters are more than five (very rare) a single register is used to hold a pointer to user space where all parameters exist.

Methods to pass parameters to OS

Three general methods used to pass parameters to the OS

- Simplest: pass the parameters in *registers*
- In some cases, may be more parameters than registers
- Parameters stored in a *block, or table, in memory,* and address of block passed as a parameter in a register
  - This approach is taken by Linux and Solaris
- Parameters placed, or *pushed, onto the stack by the* program and *popped off the stack by the operating* system
- Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



New System calls – Design considerations

- What is the purpose of new system call?
  - If possible it should have only one purpose
- What are the new system call's arguments, return value and error codes?
- Is it portable and robust?
- Verifying system call parameters
  - Ensure it is valid and legal
  - Important to check the validity of pointers a user gives
  - Before following a pointer into user space, the system must ensure
    - The pointer points to a region of memory in user space
    - The pointer points to a region of memory in the process's address space
    - If reading, memory is marked readable. If writing memory is marked writable.

Difference between System call and function call

- System call typically accessed via function calls.
- System call involves **context switching** (from user to kernel and back) where as function call does not.
- Takes much longer time than function calls.
- Avoiding excessive system calls might be a wise strategy for programs that need to be tightly optimized.
- Most of the system calls return a value (error if failed) where as it is not necessary for subroutine.
- If an error (return value –1) use perror (“Message”)
- to print the error.

Pros and Cons of system calls

- Pros
  - Simple to implement and easy to use
  - In Linux it is very fast
- Cons
  - Need a system call number, which needs to be officially assigned to you during the developmental kernel series.
  - Once stabilized, the interface can not change with out breaking user space application
  - Each architecture needs to separately register the system call and support it.
  - For simple exchanges of information, a system call is overkill (overheads because of cache miss and context switch)

Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications
- Protection

System Calls for process control

- fork()
- wait(), waitpid()
- execl(), execlp(), execv(), execvp()
- exit()
- Signal - signal(sig, handler), kill(sig, pid), alarm(), pause()
- getpid(), getppid()
- nice()

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() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()