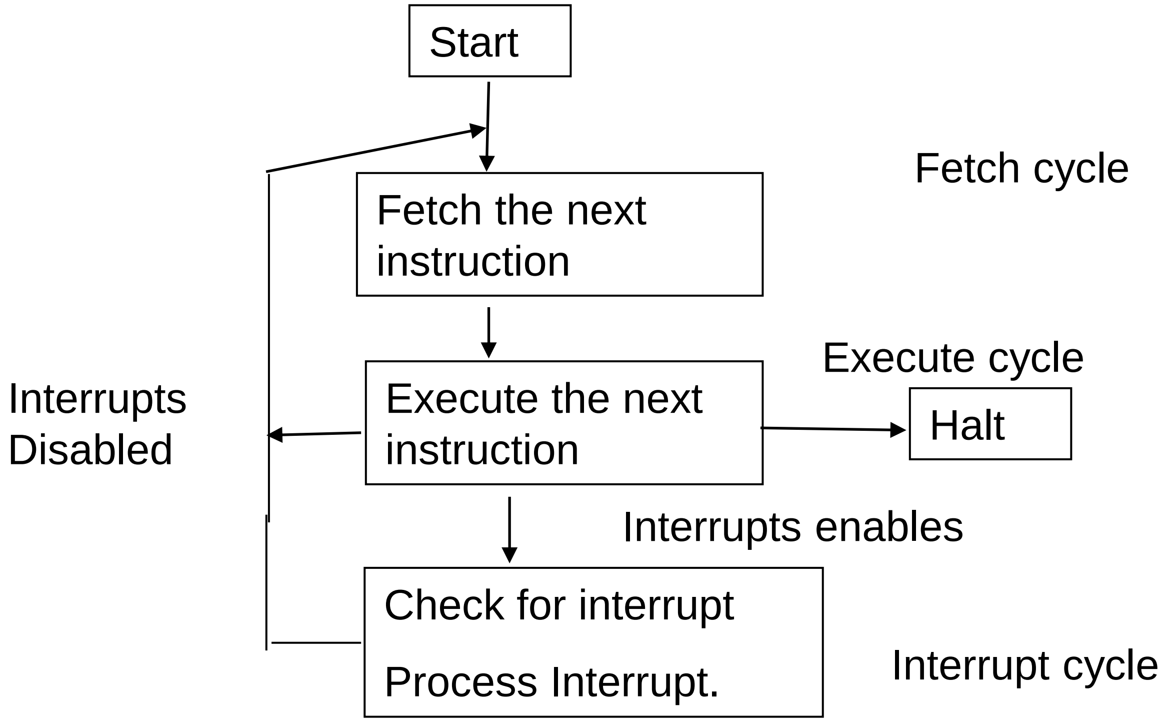
Interrupts and System Calls

Manish Shrivastava

Interrupt Processing

- To improve the performance, interrupts are provided.
- With interrupts, the processor can be engaged in executing other processes while an I/O operation is in progress.
- Interrupt cycle is added to the instruction cycle.



Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the ***interrupt vector***, which contains the addresses of all the service routines.
- Interrupt architecture must save the address of the interrupted instruction.
- Incoming interrupts are *disabled* while another interrupt is being processed to prevent a *lost interrupt*.
- A ***trap*** is a software-generated interrupt caused either by an error or a user request.
- **An operating system is *interrupt* driven.**

I/O Controller Interrupting

- To start I/O operation, CPU loads the appropriate registers within the device controller
- The device controller examines the contents of these registers to determine what action to take.
- If it s a read operation the controller transfers the data into local buffer.
- Then it informs the CPU through interrupt.
- The operating system preserves the state of the CPU by storing registers and the program counter.

Interrupt Handling

- **Interrupt handler:** The CPU hardware has a wire called interrupt request line; that CPU senses after executing every instruction.
  - When a CPU senses a signal, it saves the PC, and PSW on a stack and jump to the interrupt handler routine at a fixed address in memory.

- **Interrupt vector:** It contains the memory addresses of specialized interrupt handlers.

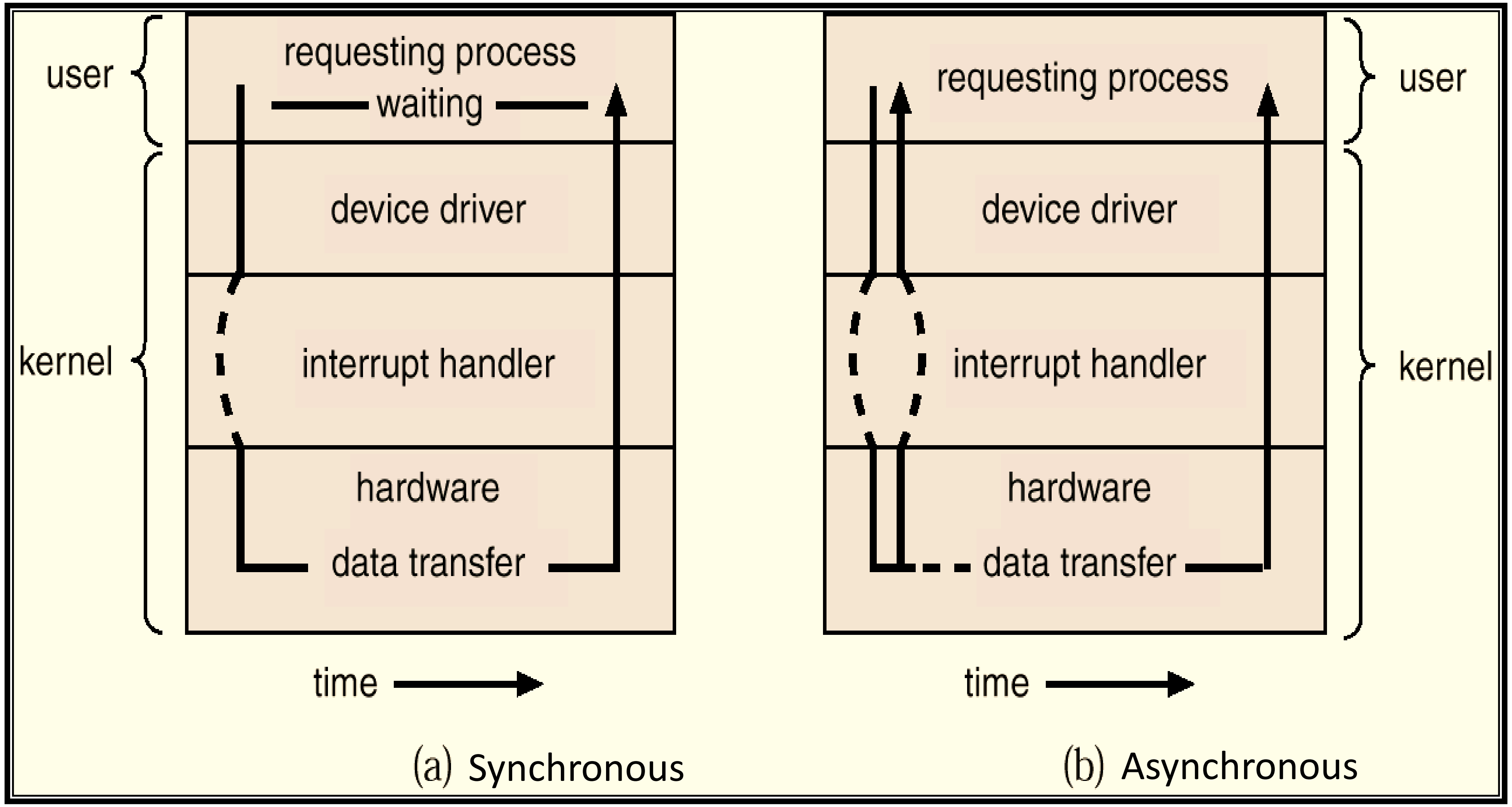
Interrupt Handling

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an ***interrupt-request line*** that is sensed after every instruction.
  - A device's controller ***raises*** an interrupt by asserting a signal on the interrupt request line.
  - The CPU then performs a state save, and transfers control to the ***interrupt handler*** routine at a fixed address in memory. (The CPU ***catches*** the interrupt and ***dispatches*** the interrupt handler.)
  - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a ***return from interrupt*** instruction to return control to the CPU. (The interrupt handler ***clears*** the interrupt by servicing the device.)
    - (Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off.)

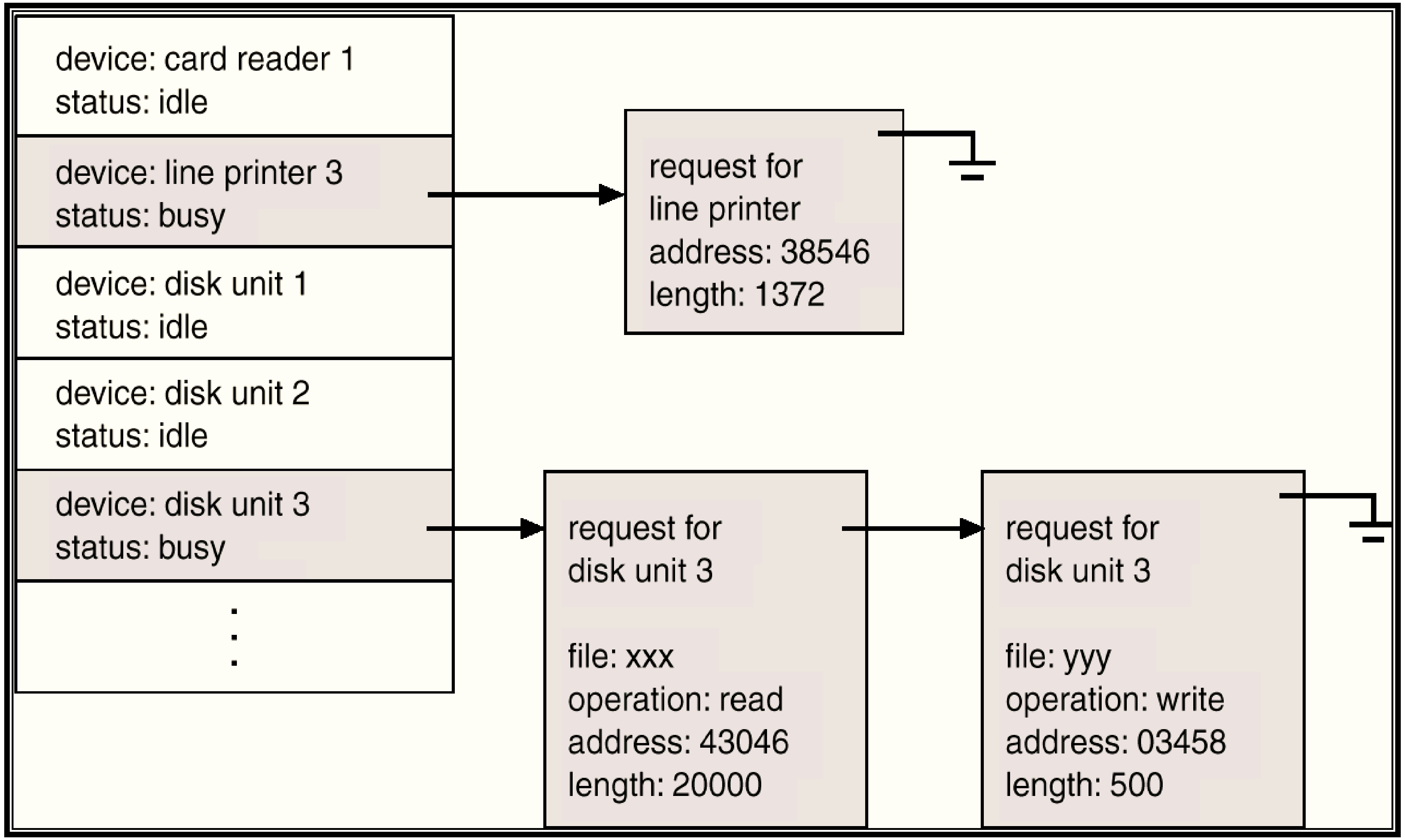
I/O Structure

- **Synchronous I/O**: After I/O starts, control returns to user program only upon I/O completion.
    - Wait instruction idles the CPU until the next interrupt
    - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- **Asynchronous I/O**: After I/O starts, control returns to user program without waiting for I/O completion.
- ***non-blocking I/O:*** the I/O request returns immediately, whether the requested I/O operation has (completely) occurred or not
- *Device-status table* contains entry for each I/O device indicating its type, address, and state.
- Operating system indexes into I/O device-status table to determine device status and to modify table entry.

Two I/O Methods



Device-Status Table



I/O communication techniques

- Two kinds of data transfer
  - **Program data transfer**: CPU checks the I/O status
    - The I/O module does not interrupt the processor.
    - Processor is responsible for extracting the data from main memory and shifting data out of main memory.
    - It is a time-consuming process that keeps the processor busy unnecessarily.
  - **Interrupt driven data transfer:** when I/O is ready it interrupts the CPU
    - In Interrupt driven data transfer, the I/O module will interrupt the processor to request service when it is ready to exchange data with the processor.

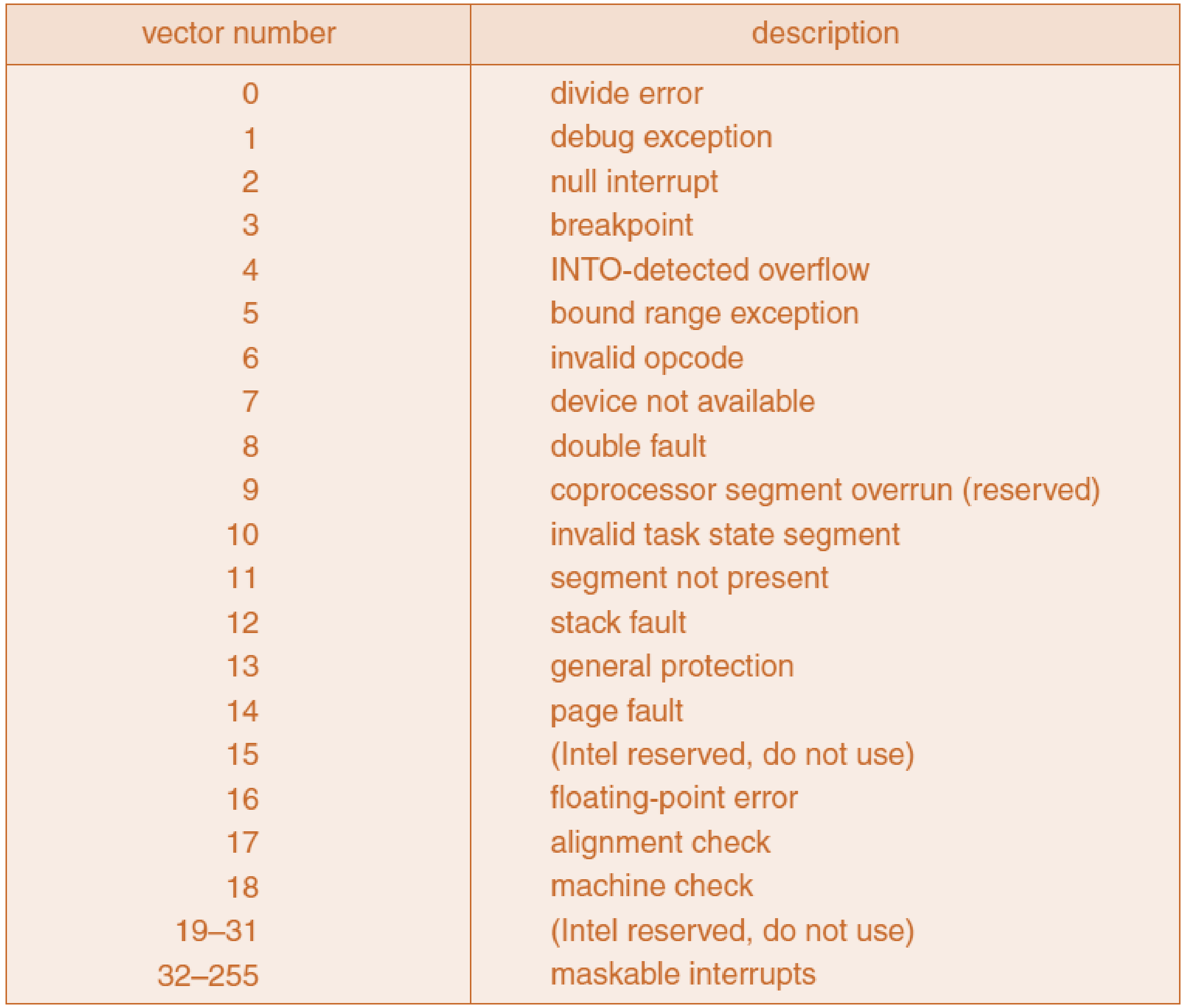
Modern Computer

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
  - The need to defer interrupt handling during critical processing,
  - The need to determine ***which*** interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
  - The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.

Interrupt Controller

- These issues are handled in modern computer architectures with ***interrupt-controller*** hardware.
  - Most CPUs now have two interrupt-request lines: One that is ***non-maskable*** for critical error conditions and one that is ***maskable,*** that the CPU can temporarily ignore during critical processing.
  - The interrupt mechanism accepts an ***address,*** which is usually one of a small set of numbers for an offset into a table called the ***interrupt vector.*** This table holds the addresses of routines prepared to process specific interrupts.
  - The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be ***interrupt chained***. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
  - Modern interrupt hardware also supports ***interrupt priority levels***, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

Interrupt Vector Table



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Interrupt Controller

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.
- Time slicing and context switches can also be implemented using the interrupt mechanism.
  - The scheduler sets a hardware timer before transferring control over to a user process.
  - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
  - The scheduler does a state-restore of a ***different*** process before resetting the timer and issuing the return-from-interrupt instruction.

Some Interrupts

- A similar example involves the paging system for virtual memory –
    - A page fault causes an interrupt, which in turn issues an I/O request and a context switch, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue.

- Some OSs (Solaris OS) use a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

Some Interrupts

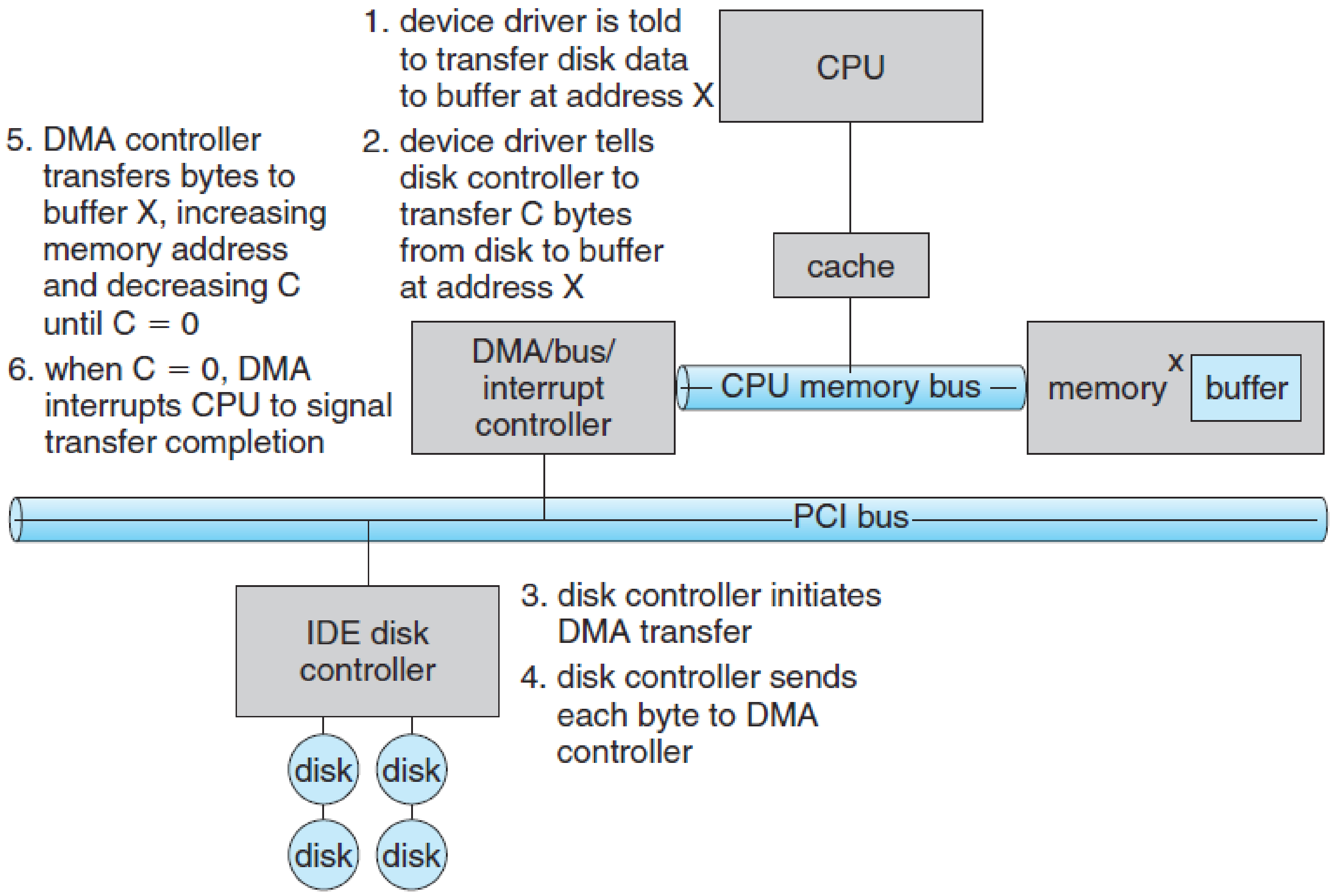
- System calls are implemented via ***software interrupts,*** a.k.a. ***traps.*** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt.
- The completion of a disk read operation involves **two** interrupts:
  - A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
  - A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.

Direct Memory Access

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time

- Instead this work can be off-loaded to a special processor, known as the ***Direct Memory Access, DMA, Controller.***
  - The host issues a command to the DMA controller, indicating the details of transfer operation. The DMA interrupts the CPU when the transfer is complete.
  - A simple DMA controller is a standard component in modern PCs, and many ***bus-mastering*** I/O cards contain their own DMA hardware.
  - Handshaking between DMA controllers and their devices is accomplished through two wires called the **DMA-request** and **DMA-acknowledge** wires.
  - While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.

Direct Memory Access


1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
CPU memory bus
memory
x
buffer
DMA/bus/ interrupt controller
PCI bus
IDE disk controller
disk
disk
disk
disk

Direct Memory Access

- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA,*** and allows direct data transfer from one memory-mapped device to another without using the main memory chips.

- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons.

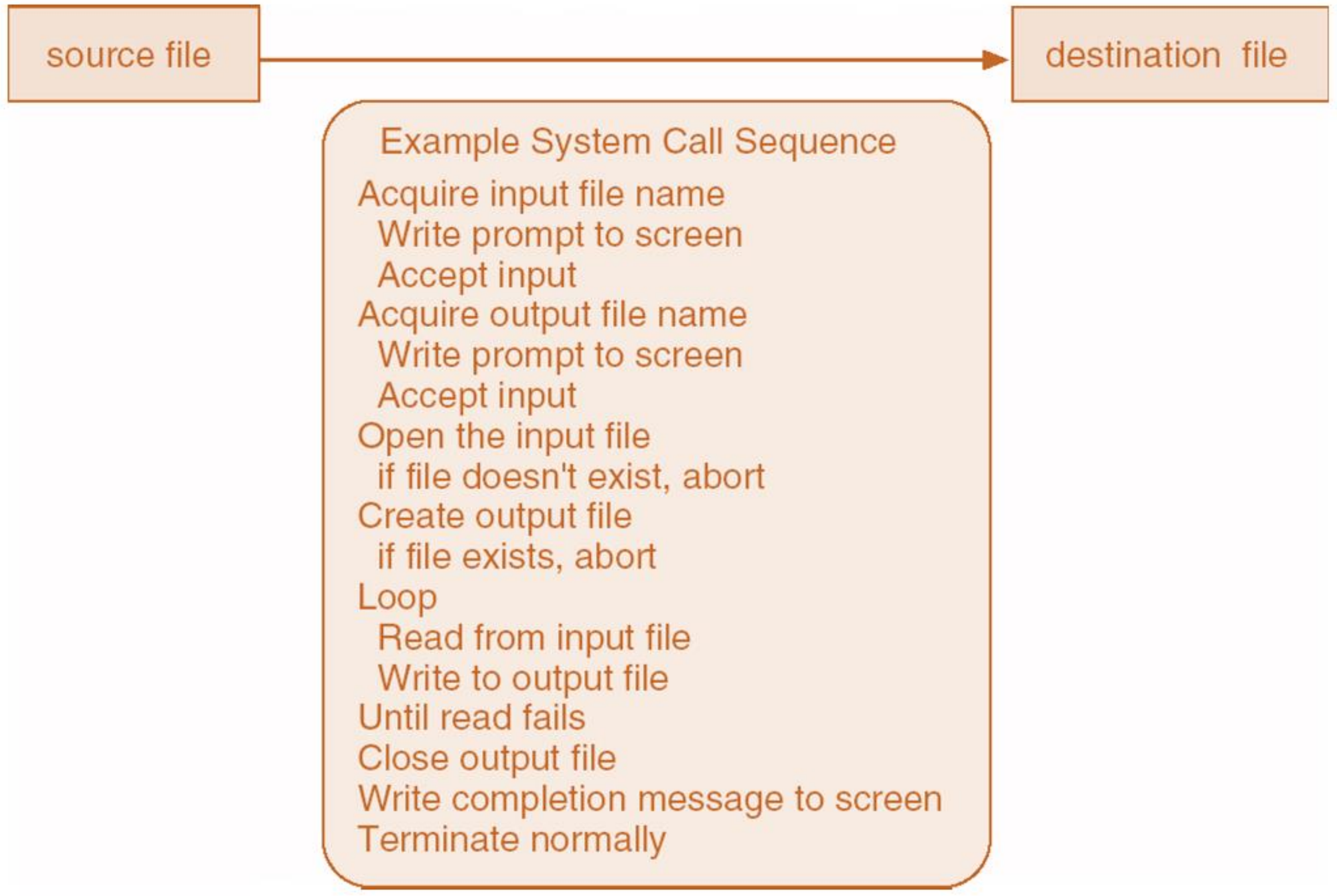
System Calls

- The system calls are the instruction set of the OS virtual processor.
- 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 Program 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).

*Portable Operating System Interface (POSIX)

Example of System Calls

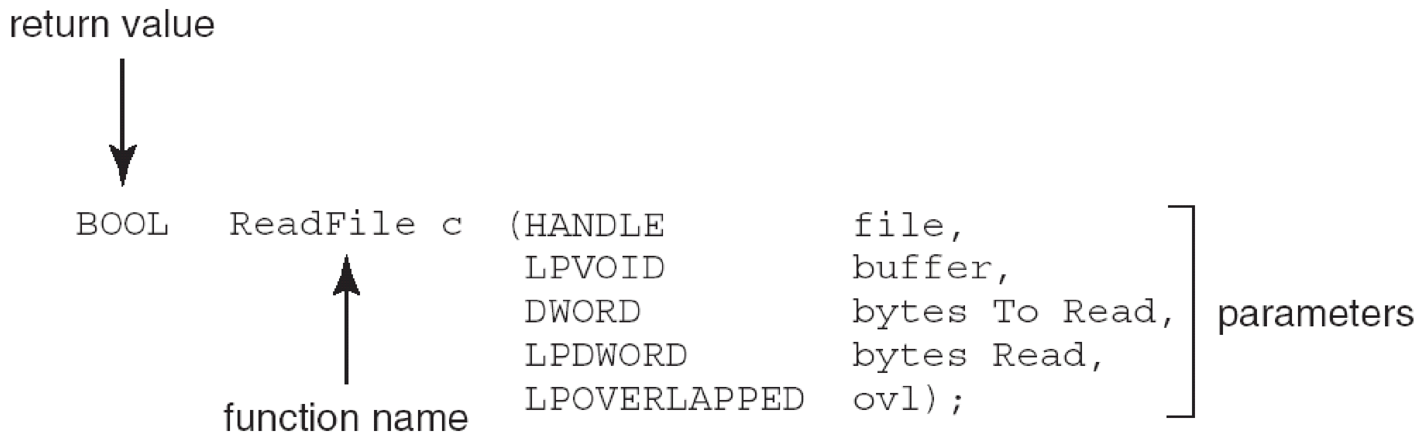
- System call sequence to copy the contents of one file to another file



```
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

- Consider the ReadFile() function in the Win32 API—a function for reading from a file



```
BOOL   ReadFile c  (HANDLE         file,
                    LPVOID         buffer,
                    DWORD          bytes To Read,
                    LPDWORD        bytes Read,
                    LPOVERLAPPED   ovl);
```

A description of the parameters passed to ReadFile()

- HANDLE file—the file to be read
- LPVOID buffer—a buffer where the data will be read into and written from
- DWORD bytesToRead—the number of bytes to be read into the buffer
- LPDWORD bytesRead—the number of bytes read during the last read
- LPOVERLAPPED ovl—indicates if overlapped I/O is being used

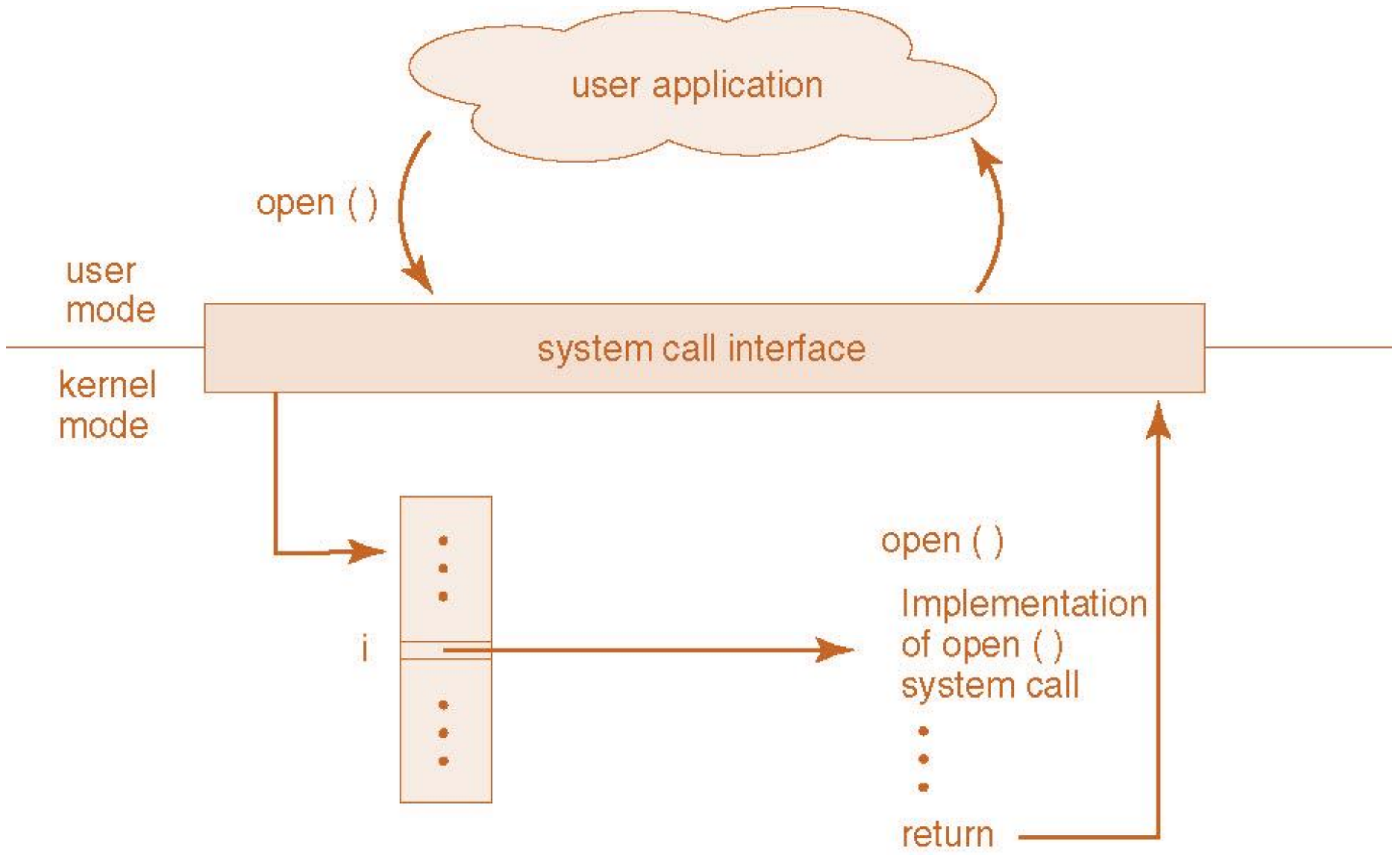
System Call Implementation

- 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

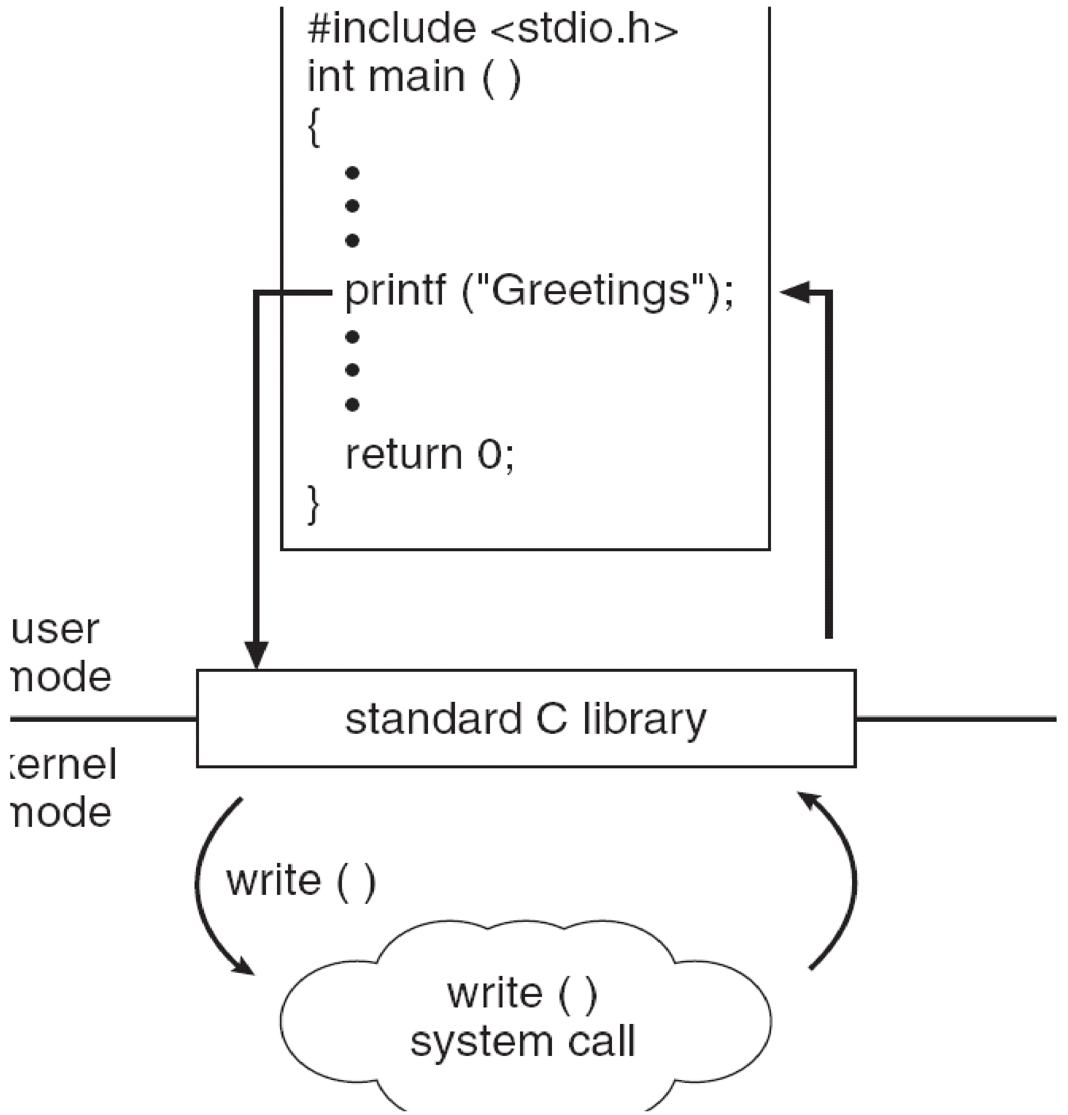
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with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

API – System Call – OS Relationship



Standard C Library Example

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

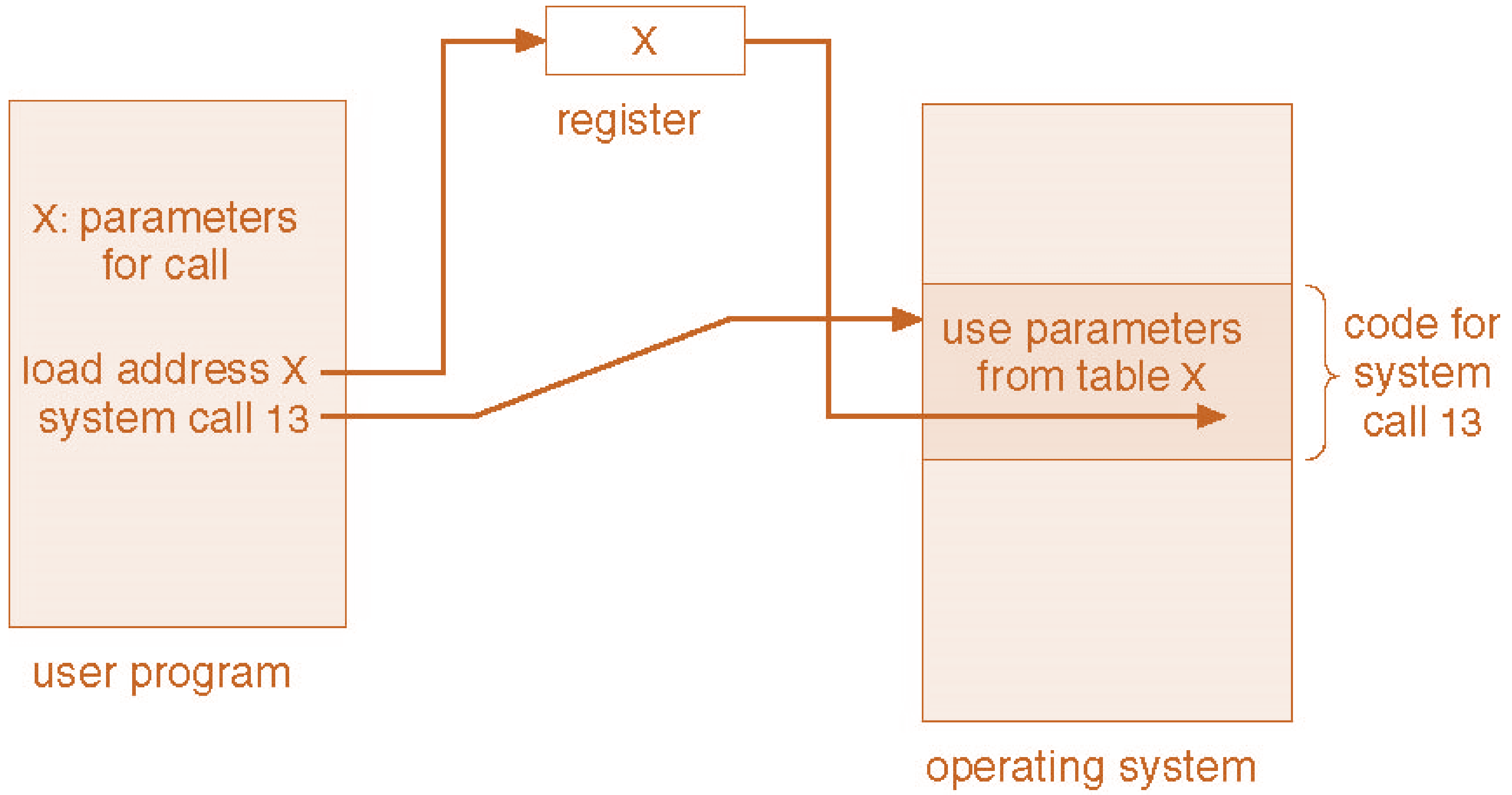

```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

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

- 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 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



Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes

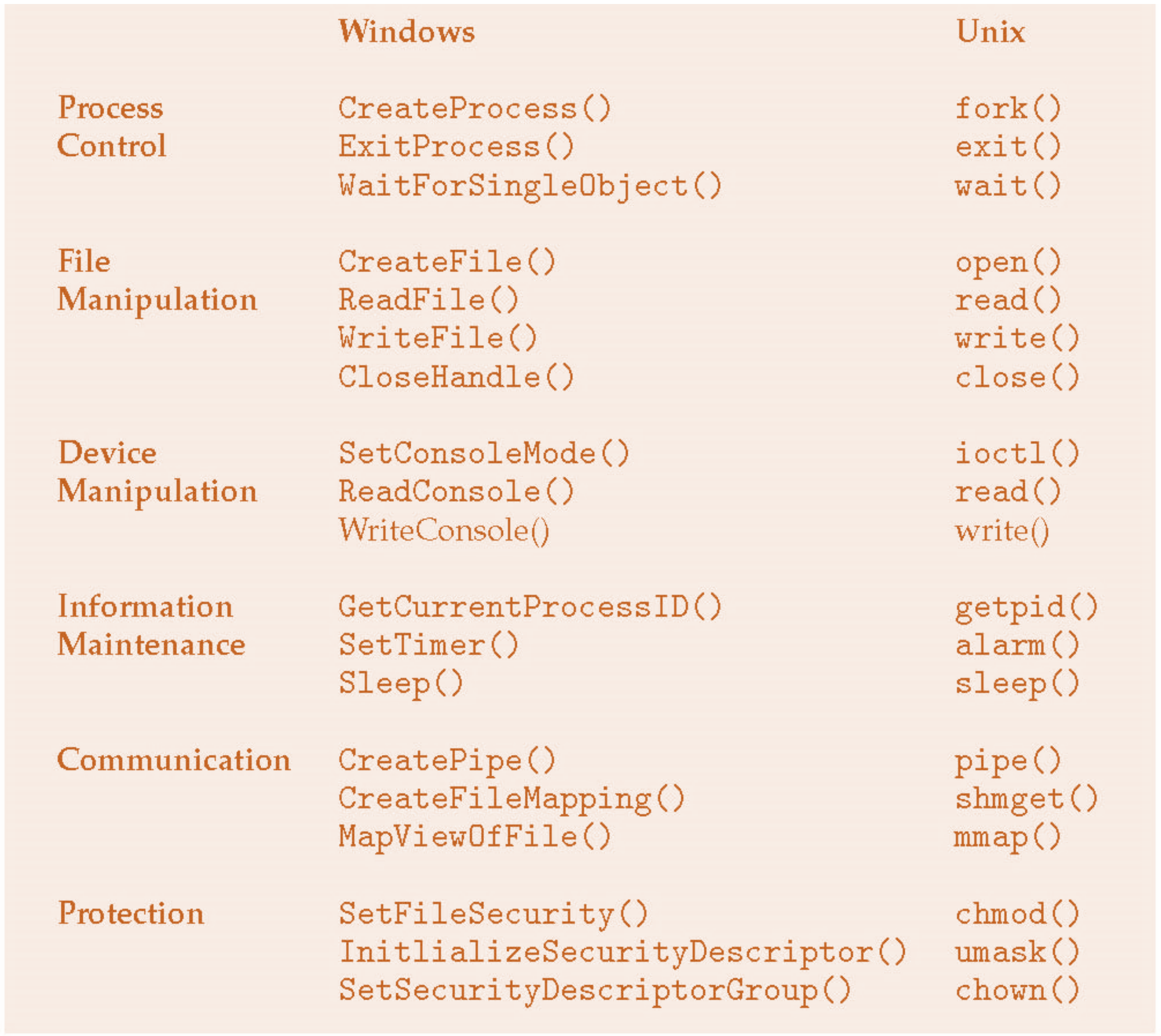
Types of System Calls

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

- 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
  - transfer status information
  - attach and detach remote devices

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()`

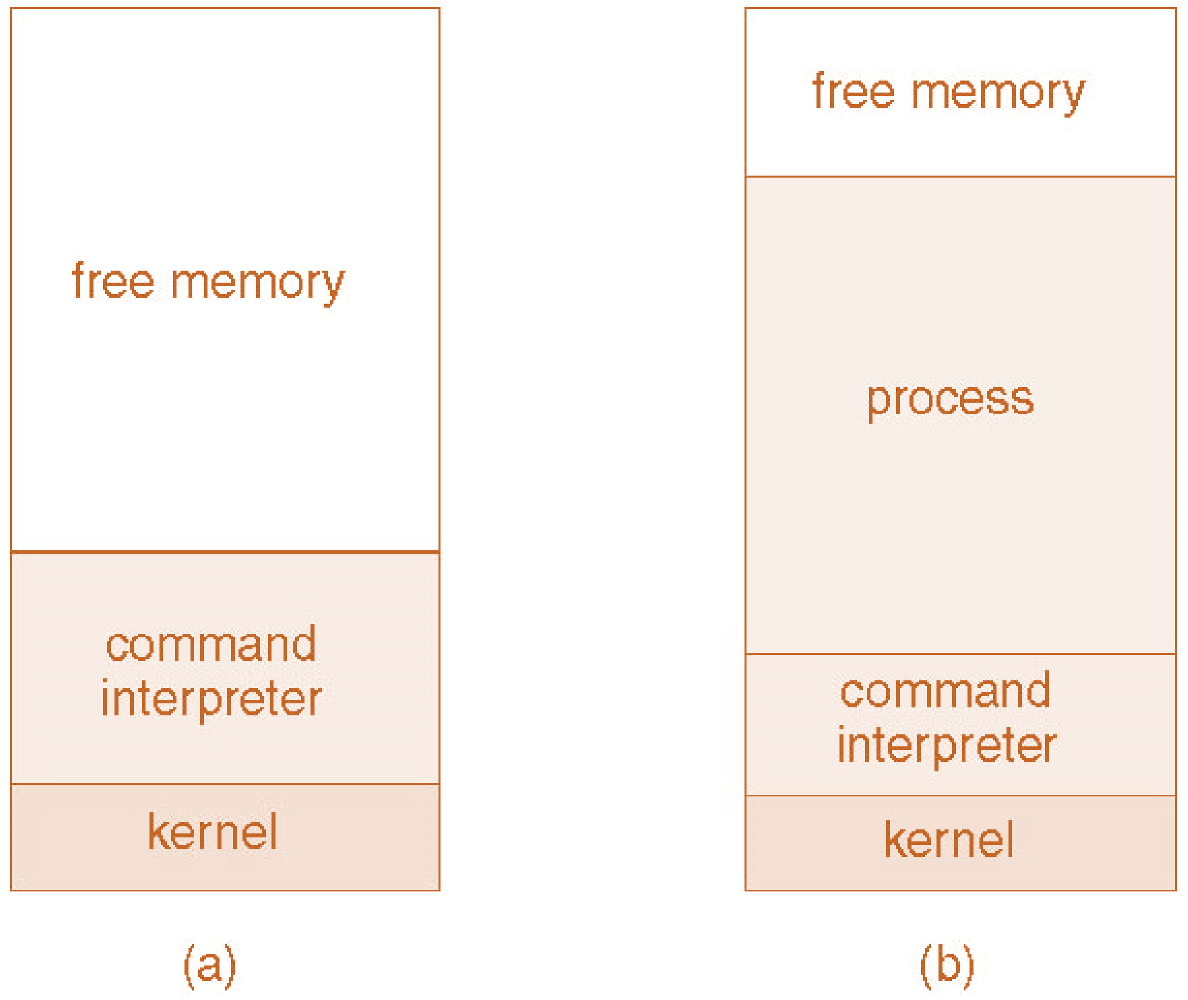
Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shmget()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Types of System Calls: Process Control Example: MS-DOS

- Single-tasking
- Shell invoked when system booted
- Simple method to run program
  - No process created
- Single memory space
- Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded

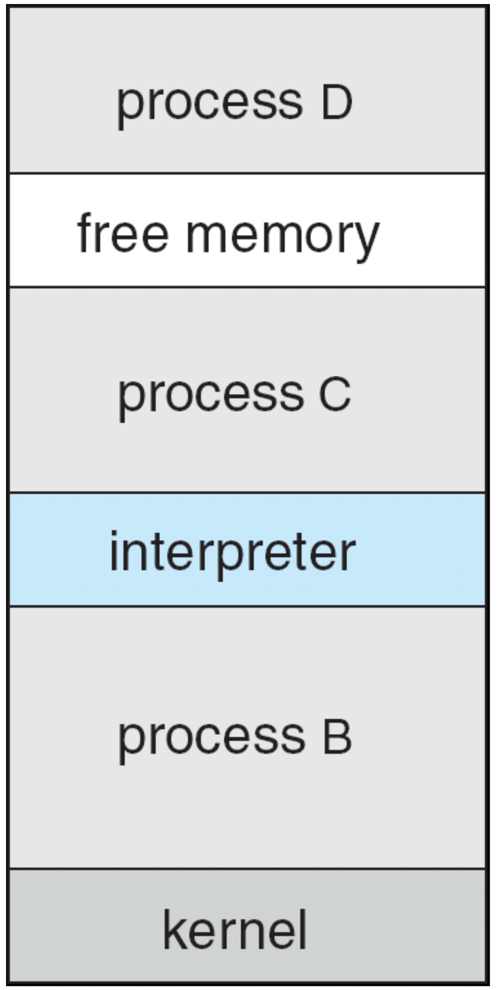
MS-DOS execution



Process Control Example: FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user’s choice of shell
- Shell executes fork() system call to create process
    - Executes exec() to load program into process
    - Shell waits for process to terminate or continues with user commands
- Process exits with code of 0 – no error or > 0 – error code

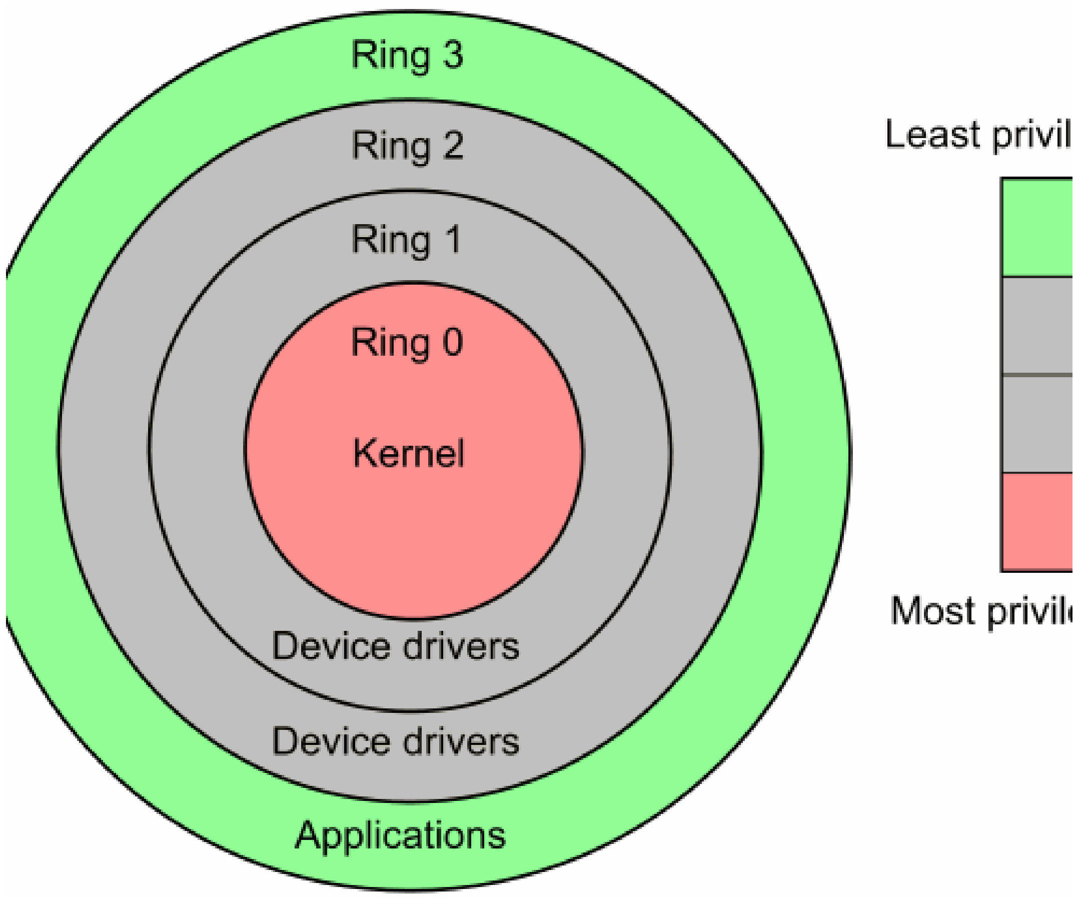
FreeBSD Running Multiple Programs



Linux system calls

- Operating systems offer processes running in User Mode ***a set of interfaces*** to interact with ***hardware devices*** such as
  - the **CPU**
  - disks
  - printers.

- **Unix** systems implement most interfaces between ***User Mode processes*** and ***hardware devices*** by means of ***system calls*** issued to the kernel.



POSIX APIs vs. System Calls

- An *application programmer interface* (API) is a function definition that specifies how to obtain a given service.

- A *system call* is an explicit request to the kernel made via a software interrupt.

From a Wrapper Routine to a System Call

- Unix systems include several libraries of functions that provide APIs to programmers.

- Some of the APIs defined by the ***libc*** standard C library refer to wrapper routines (routines whose only purpose is to issue a system call).

- Usually, each system call has a corresponding wrapper routine, which defines the API that application programs should employ.

APIs and System Calls

- An **API** does not necessarily correspond to a specific system call.
  - First of all, the **API** could offer its services directly in User Mode. (For something abstract such as math functions, there may be no reason to make system calls.)
  - Second, a single **API** function could make several system calls.
  - Moreover, several **API** functions could make the same system call, but wrap extra functionality around it.

Example of Different APIs Issuing the Same System Call

- In Linux, the *malloc( )*, *calloc( )*, and *free( )* APIs are implemented in the *libc* library.

- The code in this library keeps track of the allocation and deallocation requests and uses the *brk( )* system call to enlarge or shrink the process heap.

- Linux defines a set of seven macros called _syscall0 through _syscall6.

- Example: _syscall3(int,write,int,fd,const char *,buf,unsigned int,count)

The Return Value of a Wrapper Routine

- Most wrapper routines return an integer value, whose meaning depends on the corresponding system call.
- A return value of -1 usually indicates that the kernel was unable to satisfy the process request.
- A failure in the system call handler may be caused by
  - invalid parameters
  - a lack of available resources
  - hardware problems, and so on.
- The specific error code is contained in the ***errno*** variable, which is defined in the *libc* library.

Execution Flow of a System Call

- When a User Mode process invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function.

- As we will see in the next section, in the 80x86 architecture a Linux system call can be invoked in two different ways.

- The net result of both methods, however, is a jump to an assembly language function called the system call handler.

System Call Number

- Because the kernel implements many different system calls, the User Mode process must pass a parameter called the system call number to identify the required system call.
- The ***eax*** register is used by Linux for this purpose.
- Additional parameters are usually passed when invoking a system call.

The Return Value of a System Call

- All system calls return an integer value.
- The conventions for these return values are different from those for wrapper routines. In the kernel:
    - positive or 0 values denote a successful termination of the system call
    - negative values denote an error condition
        - In the latter case, the value is the negation of the error code that must be returned to the application program in the **`errno`** variable.
  - The **`errno`** variable is not set or used by the kernel. Instead, the wrapper routines handle the task of setting this variable after a return from a system call.

Operations Performed by a System Call

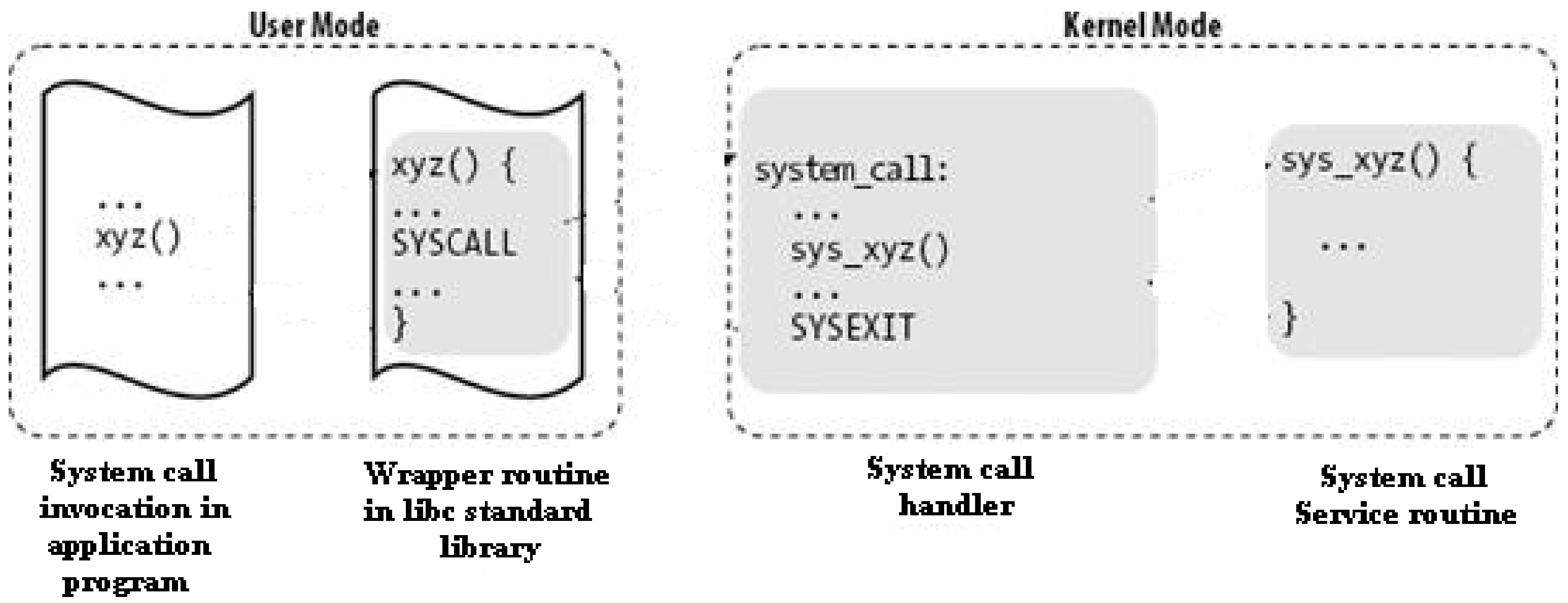
- The ***system call handler***, which has a structure similar to that of the other ***exception handlers***, performs the following operations:
  - *Saves the contents of most registers in the Kernel Mode stack.
  - Handles the system call by invoking a corresponding **C** function called the ***system call service routine***.
  - *Exits from the handler:
    - the registers are loaded with the values saved in the Kernel Mode stack
    - the **CPU** is switched back from Kernel Mode to User Mode.

*This operation is common to all system calls and is coded in assembly language.

Naming Rules of System Call Service Routines

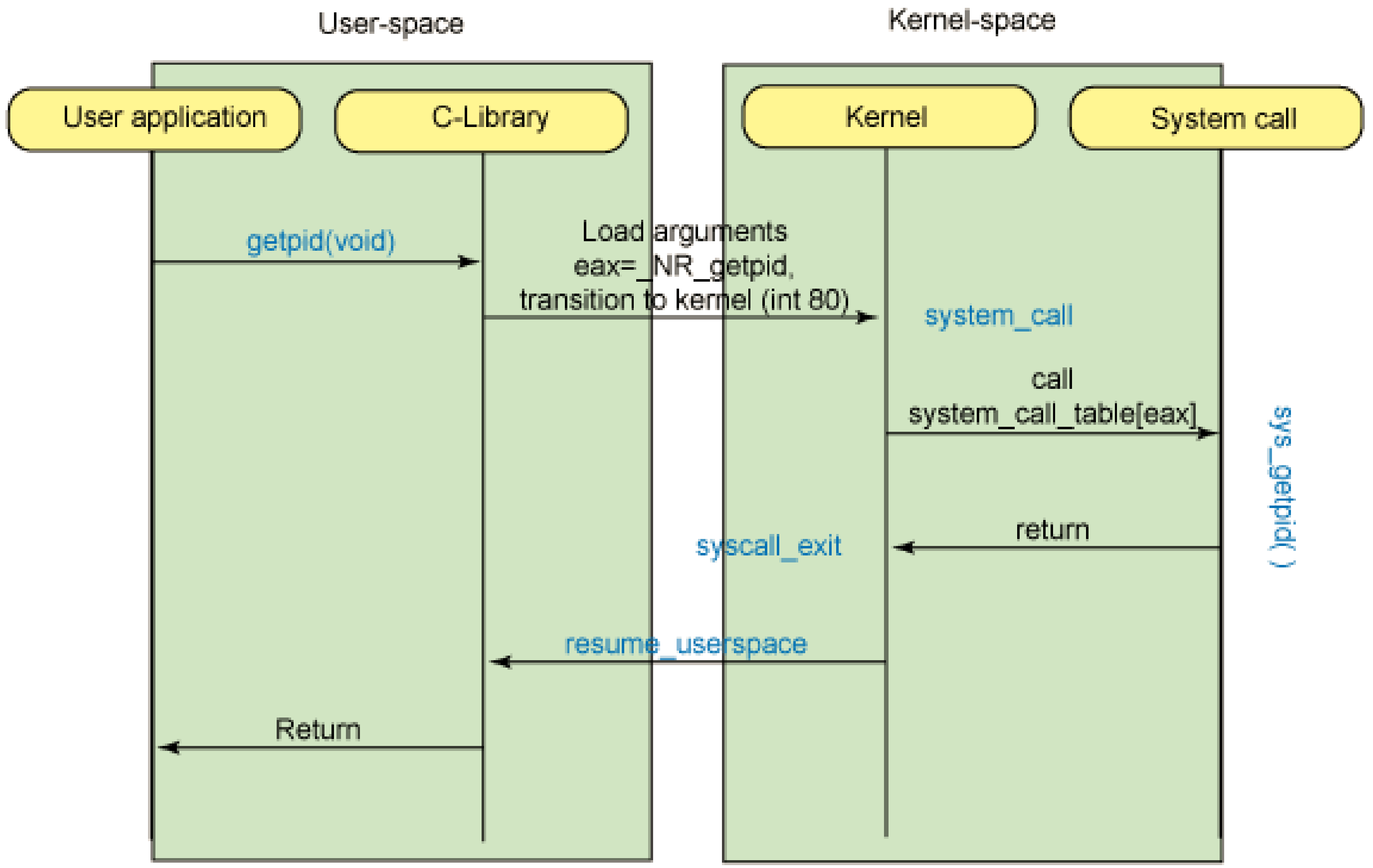
- The name of the service routine associated with the ***`xyz( )`*** system call is usually **`sys_`*****`xyz`*****`( )`**; there are, however, a few exceptions to this rule.

Control Flow Diagram of a System Call



- The **arrows** denote the execution flow between the functions.
- The terms "**SYSCALL**" and "**SYSEXIT**" are placeholders for the actual assembly language instructions that switch the **CPU**, respectively, from User Mode to Kernel Mode and from Kernel Mode to User Mode.

Control Flow Diagram of a System Call



Process Related System Calls

- The UNIX system provides several system calls to
  - create and end program,
  - to send and receive software interrupts,
  - to allocate memory, and to do other useful jobs for a process.
- Four system calls are provided for creating a process, ending a process, and waiting for a process to complete.
  - These   system calls are fork(), the "exec" family, wait(), and exit().

exec() system calls

- The UNIX system calls that transform a executable binary file into a process are the "exec" family of system calls. The prototypes for these calls are:

```
int execl(file_name, arg0 [, arg1, ..., argn], NULL)
char *file_name, *arg0, *arg1, ..., *argn;

int execv(file_name, argv)
char *file_name, *argv[];

int execle(file_name, arg0 [, arg1, ..., argn], NULL, envp)
char *file_name, *arg0, *arg1, ..., *argn, *envp[];

int execve(file_name, argv, envp)
char *file_name, *argv[], *envp[];

int execlp(file_name, arg0 [, arg1, ..., argn], NULL)
char *file_name, *arg0, *arg1, ..., *argn;

int execvp(file_name, argv)
char *file_name, *argv[];
```

exec() system calls (Cont’d)

- Unlike the other system calls and subroutines, a successful exec system call does not return. Instead, control is given to the executable binary file named as the first argument.
- When that file is made into a process, that process replaces the process that executed the exec system call -- a new process is not created.

exec() system calls (Cont’d)

- Letters added to the end of exec indicate the type of arguments:
  - l argn is specified as a list of arguments.
  - v argv is specified as a vector (array of character pointers).
  - e environment is specified as an array of character pointers.
  - p user's PATH is searched for command, and command can be a shell program

fork() system call

- To create a new process, you must use the fork() system call.
  - The prototype for

  the fork() system call is:

  int fork()

- fork() causes the UNIX system to create a new process, called the "child process", with a new process ID. The *contents* of the child process are identical to the *contents* of the parent process.

fork() system call (Cont’d)

- The new process inherits several characteristics of the old process. Among the characteristics inherited are:
  - The environment.
  - All signal settings.
  - The set user ID and set group ID status.
  - The time left until an alarm clock signal.
  - The current working directory and the root directory.
  - The file creation mask as established with umask().
- fork() returns zero in the child process and non-zero (the child's process ID) in the parent process.

wait() system call

- You can control the execution of child processes by calling wait() in the parent.
- wait() forces the parent to suspend execution until the child is finished.
- wait() returns the process ID of a child process that finished.
- If the child finishes before the parent gets around to calling wait(), then when wait() is called by the parent, it will return immediately with the child's process ID.

wait() system call (Cont’d)

- The prototype for the wait() system call is:

```
int wait(status)
int *status;
```

- “status” is a pointer to an integer where the UNIX system stores the value returned by the child process. wait() returns the process ID of the process that ended.

exit() system call

- The exit() system call ends a process and returns a value to it parent.
- The prototype for the exit() system call is:

```
 void exit(status)
int status;
```

- where status is an integer between 0 and 255. This number is returned to the parent via wait() as the exit status of the process.
- By convention, when a process exits with a status of zero that means it didn't encounter any problems; when a process exit with a non-zero status that means it did have problems.

Software Interrupt

- The UNIX system provides a facility for sending and receiving software interrupts, also called SIGNALS.
- Signals are sent to a process when a predefined condition happens.
- The number of signals available is system dependent.
- The signal name is defined in /usr/include/sys/signal.h as a manifest constant.

Signal

- Programs can respond to signals three different ways.
  - **Ignore the signal.** This means that the program will never be informed of the signal no matter how many times it occurs. The only exception to this is the SIGKILL signal which can neither be ignored nor caught.
  - **A signal can be set to its default state**, which means that the process will be ended when it receives that signal. In addition, if the process receives any of SIGQUIT, SIGILL, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, or SIGSYS, the UNIX system will produce a core image (core dump), if possible, in the directory where the process was executing when it received the program-ending signal.
  - **Catch the signal**. When the signal occurs, the UNIX system will transfer control to a previously defined subroutine where it can respond to the signal as is appropriate for the program.

Signal

- Related system calls
  - signal
  - kill
  - alarm

signal() system call

- You define how you want to respond to a signal with the signal() system call. The prototype is:

```
 #include <sys/signal.h>

int (* signal ( signal_name, function ))
int signal_name;
int (* function)();
```

kill() system call

- The UNIX system sends a signal to a process when something happens, such as typing the interrupt key on a terminal, or attempting to execute an illegal instruction. Signals are also sent to a process with the kill() system call. Its prototype is:

```
 int kill (process_id, signal_name )
int process_it, signal_name;
```

alarm() system call

- Every process has an alarm clock stored in its system-data segment. When the alarm goes off, signal SIGALRM is sent to the calling process. A child inherits its parent's alarm clock value, but the actual clock isn't shared.
- The alarm clock remains set across an exec. The prototype for alarm() is:

```
unsigned int alarm(seconds)
unsigned int seconds;
```

- Check
  - timesup.c

Examples : File Creation

```
/* creat.c */
#include <stdio.h>
#include <sys/types.h> /* defines types used by sys/stat.h */
#include <sys/stat.h> /* defines S_IREAD & S_IWRITE */
int main() {
    int fd;
    fd = creat("datafile.dat", S_IREAD | S_IWRITE);
    if (fd == -1)
            printf("Error in opening datafile.dat\n");
    else {
            printf("datafile.dat opened for read/write access\n");
            printf("datafile.dat is currently empty\n");
    }
    close(fd);
    exit (0);
}
```

Stat.h

```
#define S_IRWXU 0000700 /* -rwx------ */
#define S_IREAD 0000400 /* read permission, owner */
#define S_IRUSR S_IREAD
#define S_IWRITE 0000200 /* write permission, owner */
#define S_IWUSR S_IWRITE
#define S_IEXEC 0000100 /* execute/search permission, owner */
#define S_IXUSR S_IEXEC
#define S_IRWXG 0000070 /* ----rwx--- */
#define S_IRGRP 0000040 /* read permission, group */
#define S_IWGRP 0000020 /* write " " */
#define S_IXGRP 0000010 /* execute/search " " */
#define S_IRWXO 0000007 /* -------rwx */
#define S_IROTH 0000004 /* read permission, other */
#define S_IWOTH 0000002 /* write " " */
#define S_IXOTH 0000001 /* execute/search " " */
```

File Open

```
/* open.c */

#include <fcntl.h> /* defines options flags */

#include <sys/types.h> /* defines types used by sys/stat.h */

#include <sys/stat.h> /* defines S_IREAD & S_IWRITE */

static char message[] = "Hello, world";

int main() {

    int fd; char buffer[80];

    fd = open("datafile.dat",O_RDWR | O_CREAT | O_EXCL, S_IREAD | S_IWRITE);

    if (fd != -1) {

                printf("datafile.dat opened for read/write access\n");

                write(fd, message, sizeof(message));

                lseek(fd, 0L, 0); /* go back to the beginning of the file */

                if (read(fd, buffer, sizeof(message)) == sizeof(message))

                            printf("\"%s\" was written to datafile.dat\n", buffer);

                else

                            printf("*** error reading datafile.dat ***\n");

                close (fd);

                } else printf("*** datafile.dat already exists ***\n");

    exit (0);

}
```