CISC 3320 MW3

C02a: OS Functions and Services

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

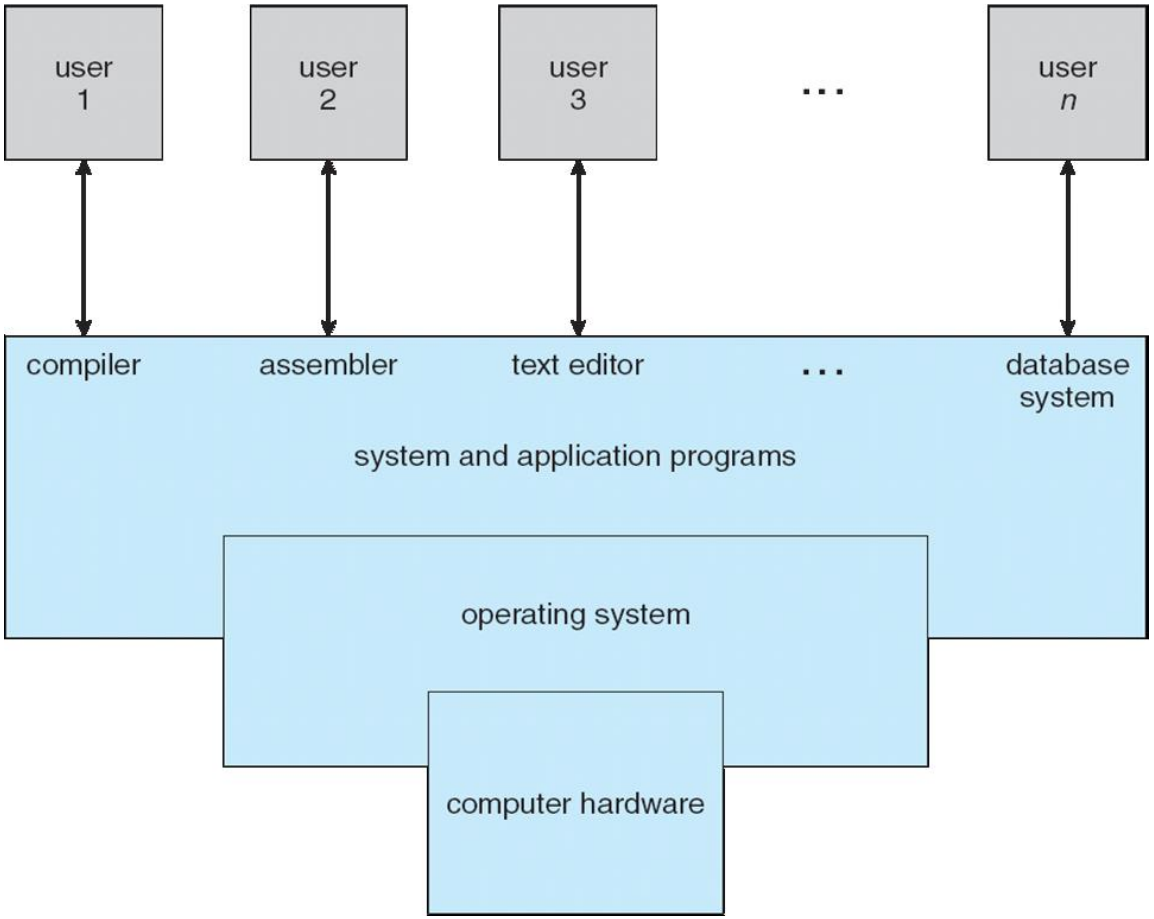
Acknowledgement

- This slides are a revision of the slides by the authors of the textbook

Outline

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls

A Computer System: Four Components



OS Services and Functions

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user
- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

User Services

- User interface
- Program execution
- I/O operation
- File-system manipulation
- Communication
- Error detection

User Interface

- Almost all operating systems have a user interface (UI)
  - Command-Line Interface (CLI)
  - Graphical User Interface (GUI)
  - Batch Interface
  - Touchscreen interface

Program Execution

- The system must be able
  - to load a program into memory,
  - to run that program, and
  - end execution, either normally or abnormally (indicating error)

I/O Operation

- A running program may require I/O, which may involve a file or an I/O device

File-system Manipulation

- Programs need
  - to read and write files and directories,
  - to create and delete them,
  - to search them,
  - to list file Information, and
  - to manage permissions

Communications

- Processes may exchange information
  - on the same computer or
  - between computers over a network
- Communications may be
  - via shared memory or
  - through message passing (packets moved by the OS)

Error Detection

- OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

Questions?

System Functions

- Resource allocation
- Accounting
- Protection and security

Resource Allocation

- When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - Many types of resources
    - CPU cycles
      main memory
    - file storage
    - I/O devices.

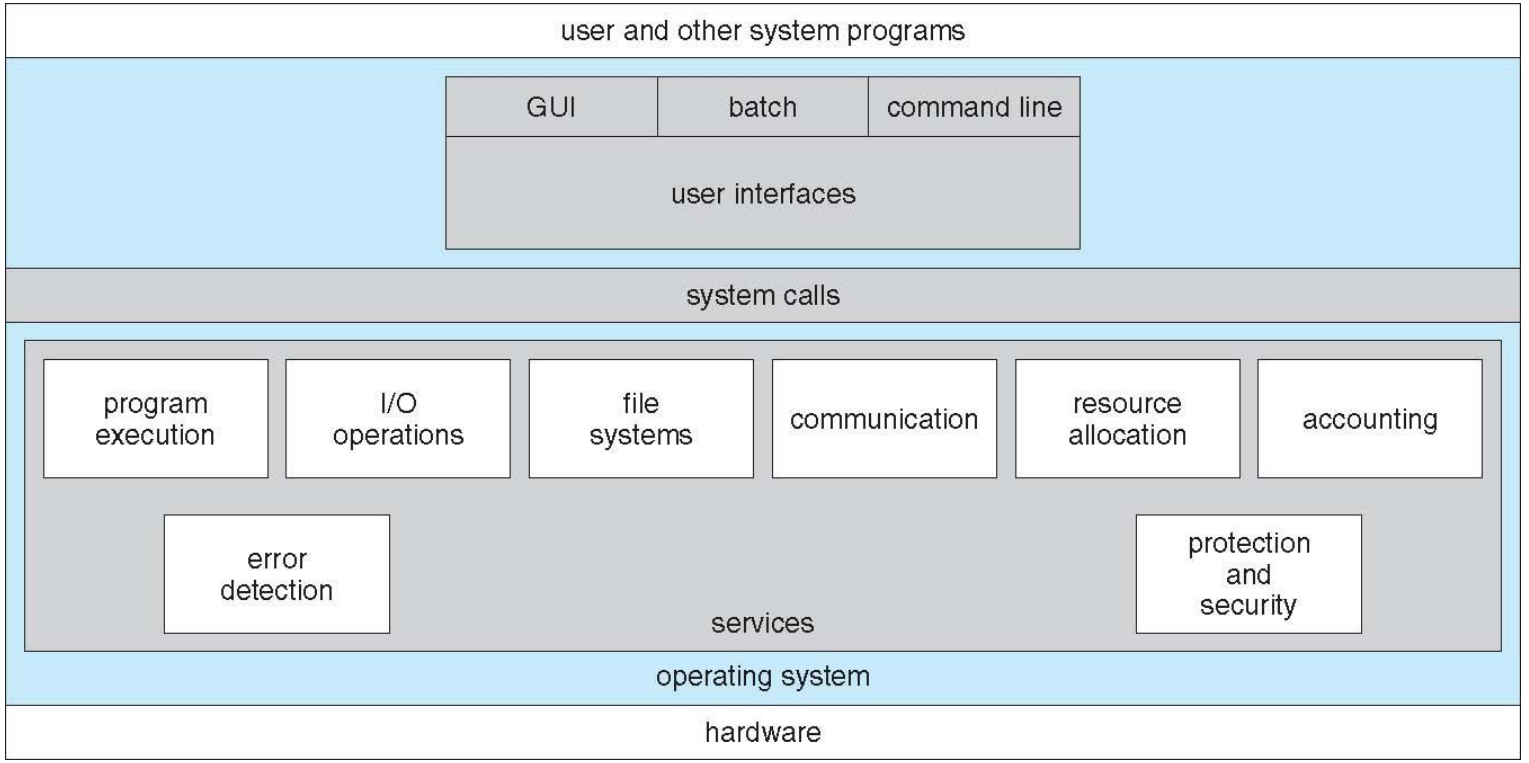
Accounting

- To keep track of which users use how much and what kinds of computer resources

Protection and Security

- The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - Protection involves ensuring that all access to system resources is controlled
  - Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A View of OS Services and Functions



Questions?

- Overview of services and functions for system itself

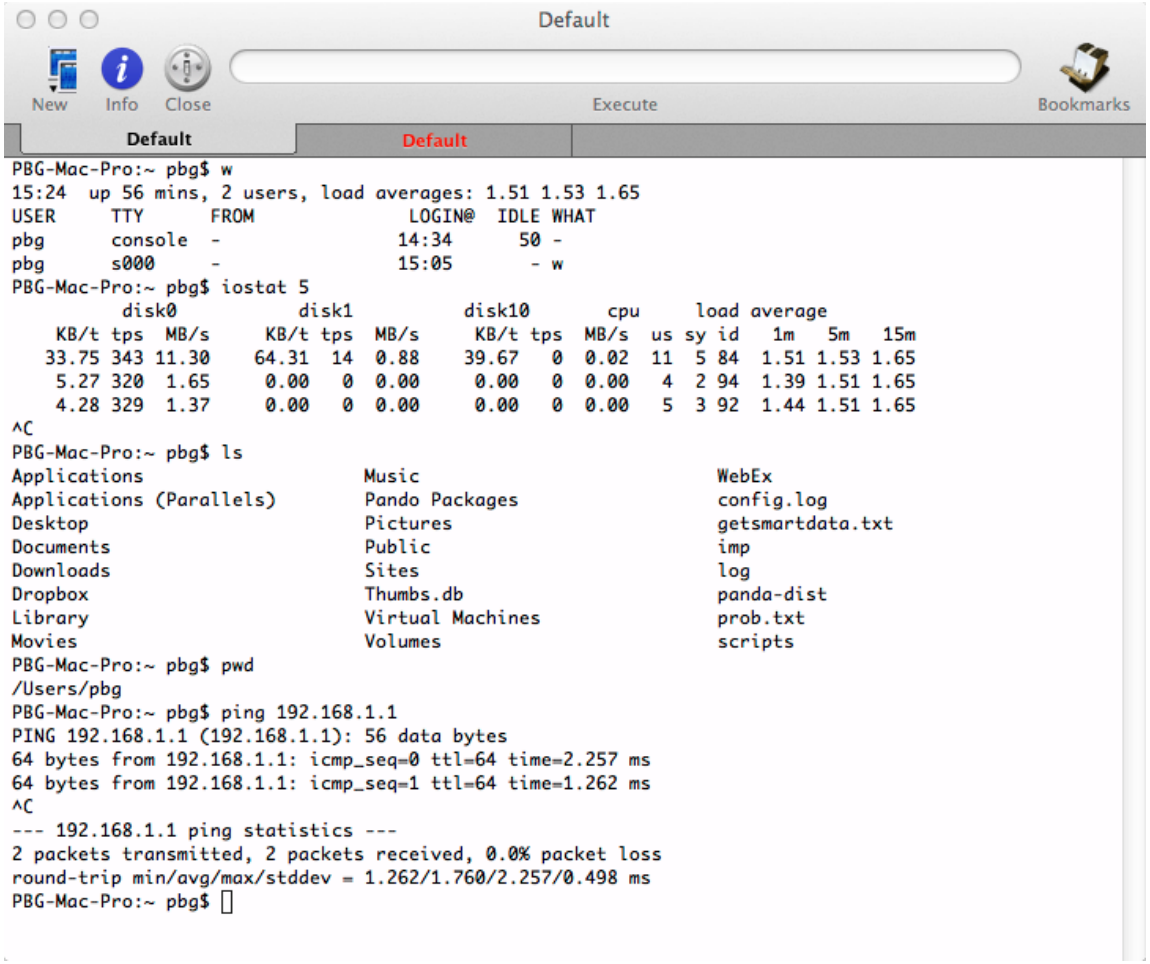
User Operating System Interface

- Command-Line Interface (CLI)
- Graphical User Interface (GUI)
- Touchscreen Interface

CLI

- Command Line Interface (CLI) or Command Interpreter
  - CLI or command interpreter allows direct command entry
- Sometimes implemented in kernel, sometimes by systems program
- Sometimes multiple flavors implemented
  - Called shells
- Primarily fetches a command from user and executes it
- Sometimes commands built-in, sometimes just names of programs
- If the latter, adding new features doesn't require shell modification

Example: Bourne Shell Command Interpreter



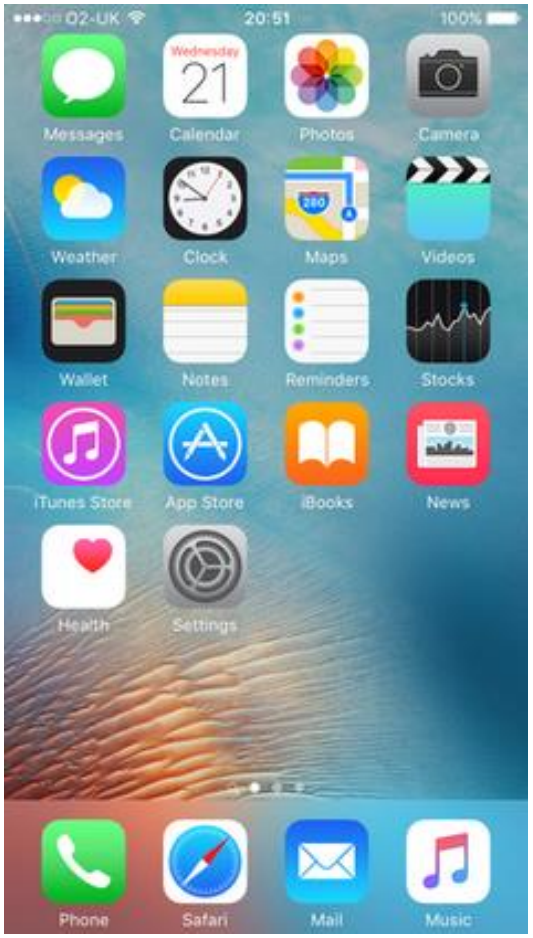
GUI

- User-"friendly" desktop metaphor interface
  - Usually mouse, keyboard, and monitor
  - Icons represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a folder)
- Invented at Xerox PARC
  - Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X is "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Unix and Linux have CLI with optional GUI interfaces (CDE, KDE, GNOME)

The Mac OS X GUI

Touchscreen Interfaces

- Touchscreen devices require new interfaces
  - Mouse not possible or not desired
  - Actions and selection based on gestures
  - Virtual keyboard for text entry
  - Voice commands.



Questions?

- Overview of different types of user interface

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)

Examples of System Calls

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

Example System Call Sequence

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

Standard API vs. System Calls

- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs
  - 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)

Example of Standard API

- The read API in UNIX/Linux (POSIX read)
- The ReadFile API in Windows

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>

ssize_t     read(int fd, void *buf, size_t count)
```

return value — function name — 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.

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `ReadFile()` function in the Win32 API—a function for reading from a file. The API for this function appears in Figure 2.5.

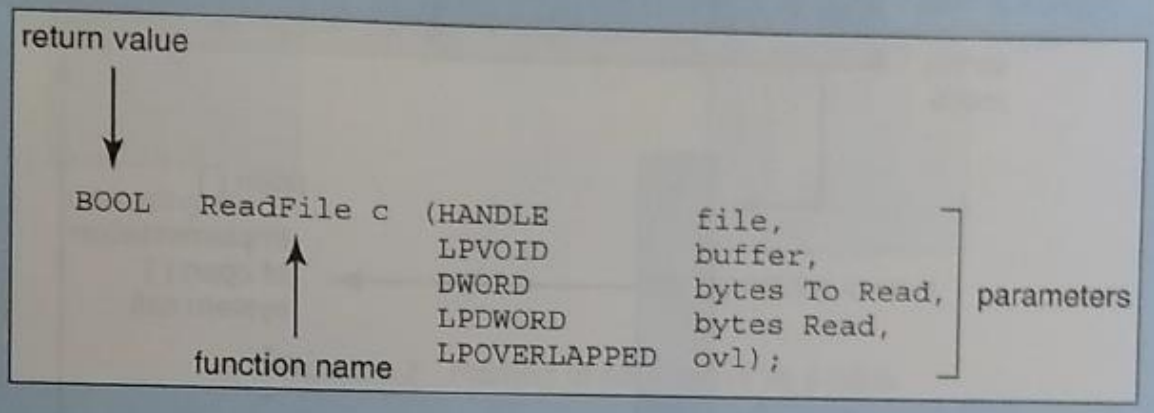


Figure 2.5 The API for the `ReadFile()` function.

A description of the parameters passed to `ReadFile()` is as follows:

- 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

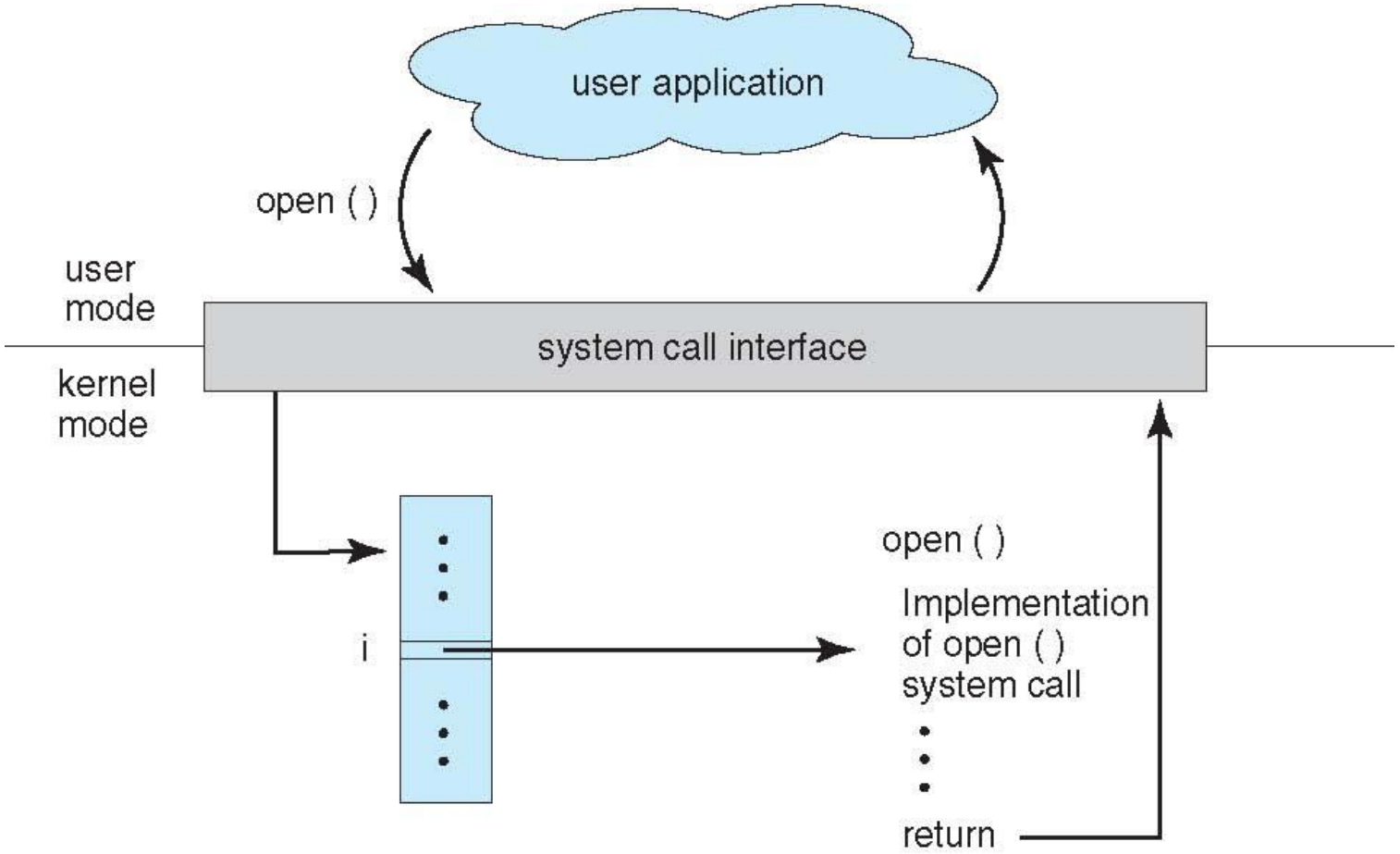
Questions?

- System calls
- System call vs Library API

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
- 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)

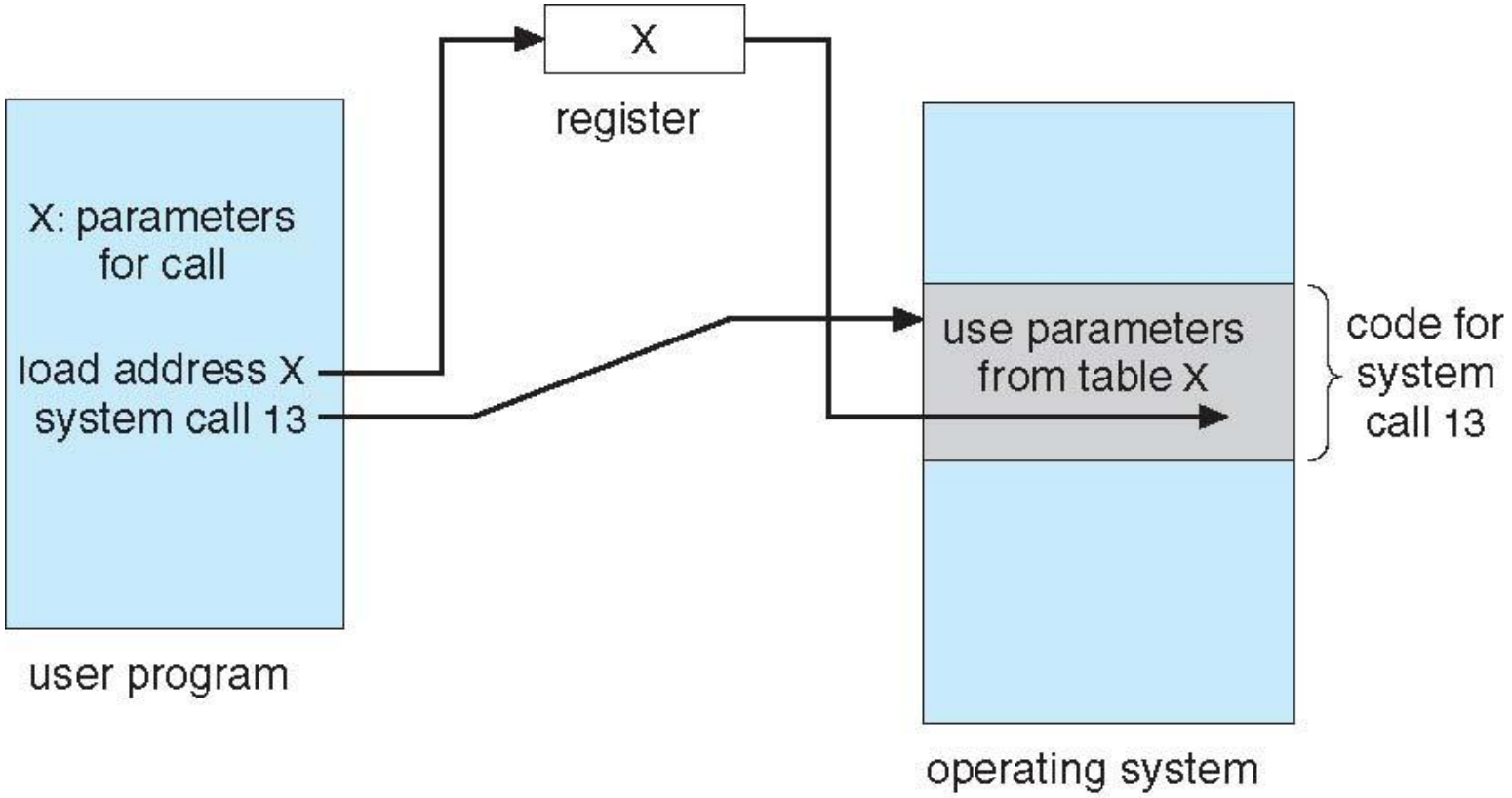
API vs. System Call vs. OS



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
  - Via registers: Simplest: pass the parameters in registers
    - In some cases, may be more parameters than registers
  - Via table: 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
  - Via stack: 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



Questions?

- Conceptual idea of system call implementation
- How to pass parameters to system calls?

Types of System Calls

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

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

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

Information Maintenance

- get time or date, set time or date
- get system data, set system data
- get and set process, file, or device attributes

Communication

- 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

Protection

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

Examples of Windows and UNIX System Calls

- Loosely speaking, the examples show the APIs. The APIs wrap around 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()`

Questions?

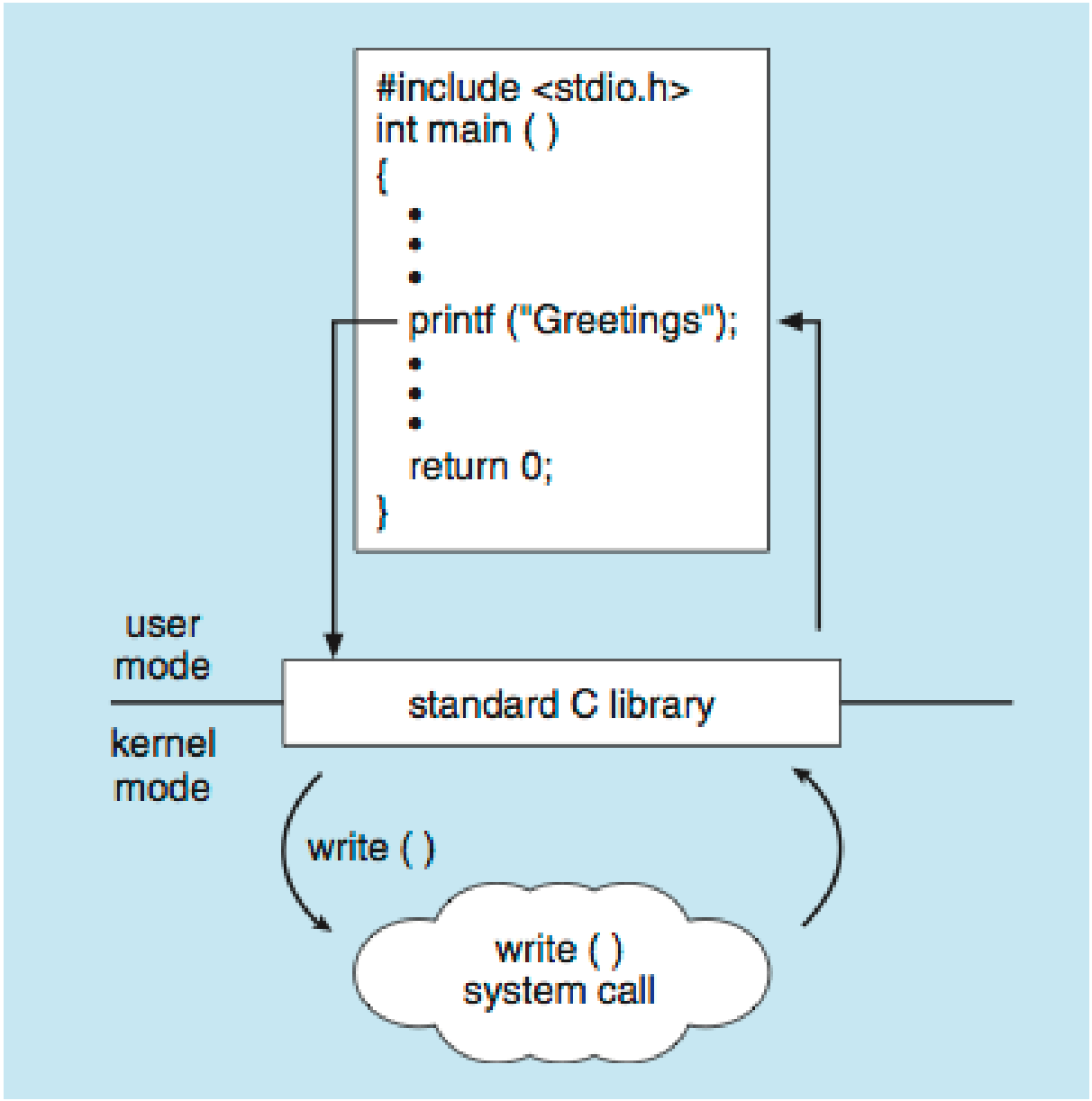
- Overview of different types of system calls.

Standard C Library

- The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux

Example of the standard C Library API

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

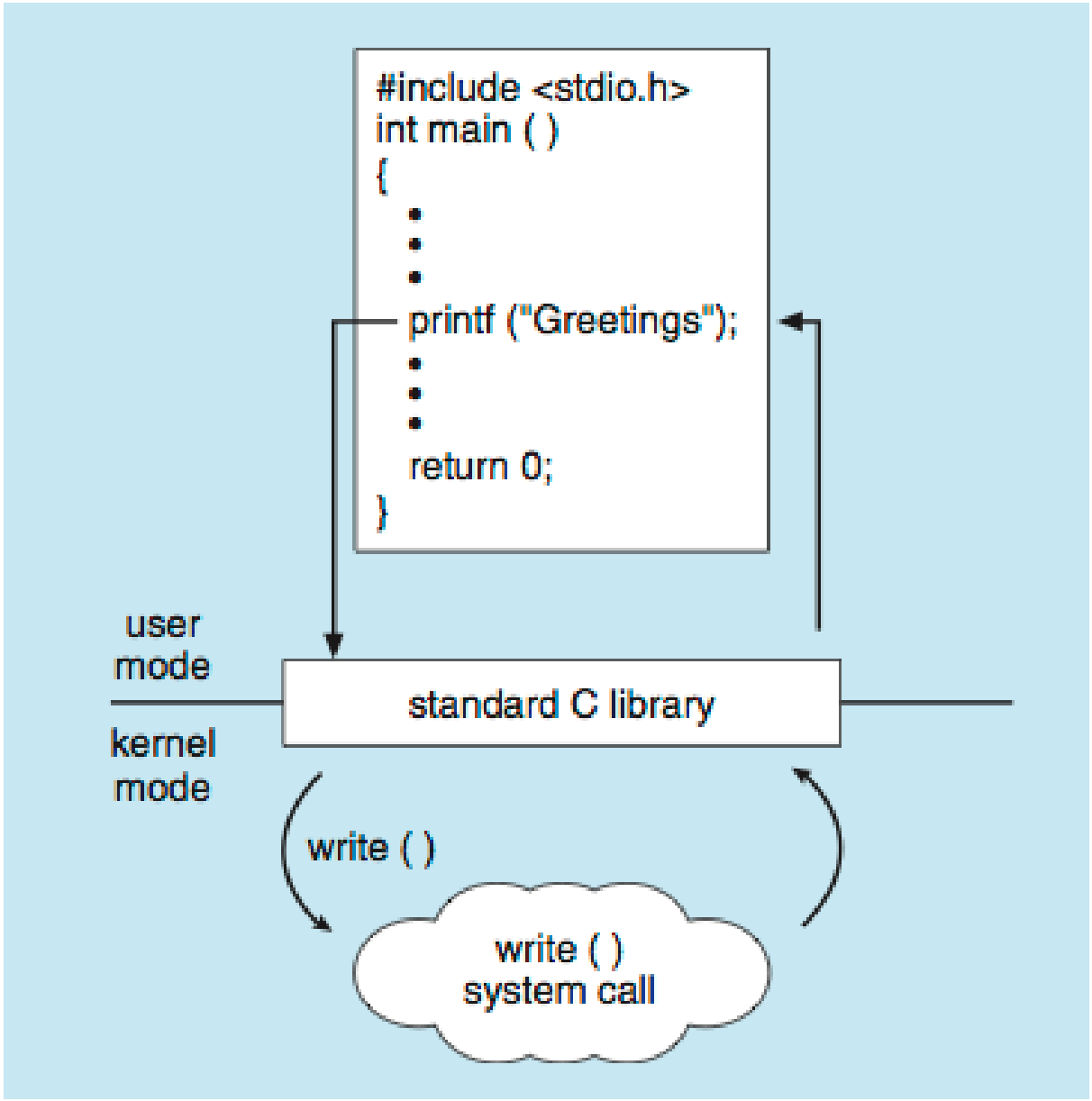

#include <stdio.h>
int main ( )
{
printf ("Greetings");
return 0;
}
user mode
kernel mode
standard C library
write ( )
write ( )
system call

Standard C Library

- The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux

Example of the standard C Library API

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


#include <stdio.h>
int main ( )
{
printf ("Greetings");
return 0;
}
user mode
kernel mode
standard C library
write ( )
write ( )
system call

Questions?

- Standard C library and system calls