Chapter 2
OS Structure

CS 3423 Operating Systems
National Tsing Hua University

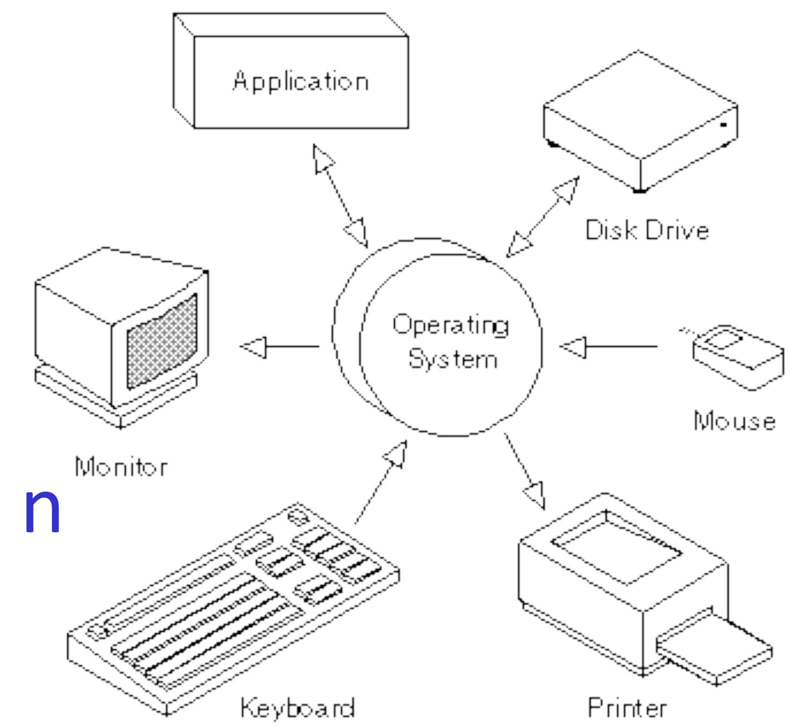
Chapter 2: OS Structures

- OS Services
- User OS Interface
- System Calls
- Types of System Calls
- System Programs
- OS Design and Implementation
- OS Structure
- OS Debugging
- OS Generation
- System Boot

OS Services

OS Services

- User interface
- program execution
- I/O operations
- file-system manipulation
- communication
- error detection
- resource allocation
- logging and accounting
- protection and security



OS Services (1/3)

- User interface
  - CLI, GUI, Batch
- Program execution
  - loader loads a program into memory to run
  - program ends execution, either normally or abnormally (indicating error) - gives control back to OS
- I/O operations
  - file or I/O device

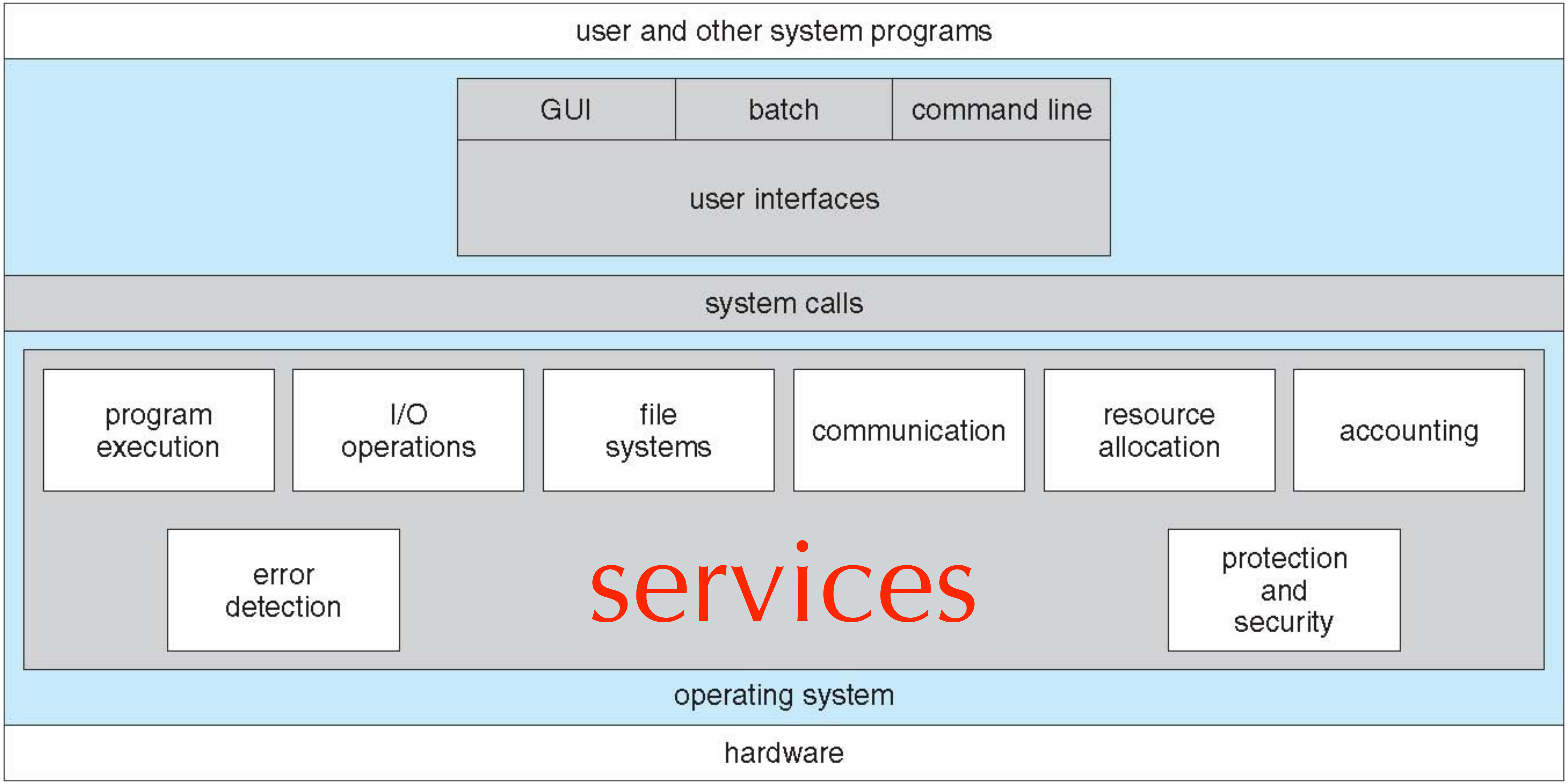
OS Services (2/3)

- File system manipulation
  - read, write, create, delete files and directories
  - search them, list file info, manage permission
- Communications
  - between processes on same host, or between hosts over network
  - shared memory vs. message passing
- Error detection
  - CPU and memory hardware, I/O devices, user program
  - Debugging facilities

OS Services (3/3)

- Resource allocation
  - for multiple users or multiple jobs running concurrently
  - including CPU cycles, main memory, file storage, I/O devices.
- Accounting
  - keep track of which users use how much and what kinds of computer resources
- Protection and security
  - Protection: ensuring all access to system resources is controlled
  - Security: user authentication, defending external I/O devices from invalid access attempts

A View of Operating System Services



User Interfaces

- CLI: Command-line Interface
  - for user to type command as text and execute code (could be built-in command or name of program)
  - shell (command-line interpreter) csh, bash,
- GUI: graphical user interface
  - usually mouse, keyboard, display, now touch screen
  - graphical elements to represent data object or control
  - direct manipulation and visual / audio feedback
- Others:
  - Gesture-based, Brain-Computer Interface (BCI), Voice

Two approaches to shell

- Shell that understands all commands
  - self-contained, bigger shell, but efficient per command
  - to add commands => need to modify the shell!
- Shell that invokes executable file
  - does not understand the command; only the syntax (e.g., command arg1 arg2 arg3 ...)
  - invokes executable corresponding to command
  - smaller shell, heavier weight per command,
  - very expandable, no need to modify shell to add command

Application-OS Interface

System Calls

API

System Calls

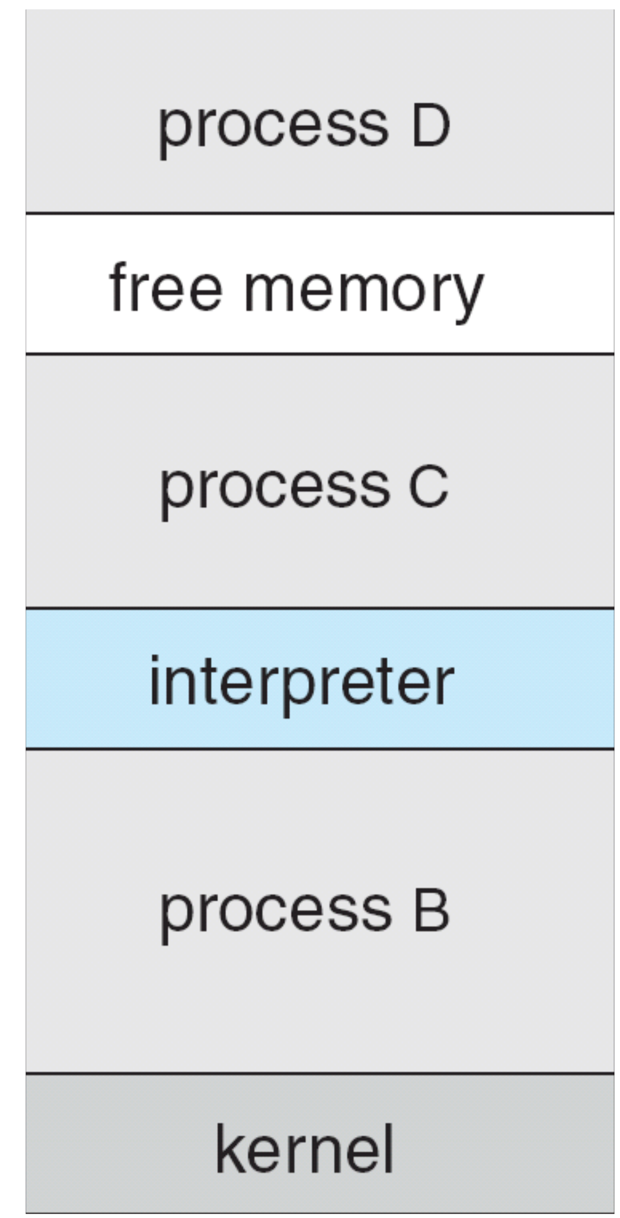
- Function calls to request OS services
- Process control
  - abort, create, terminate process, allocate/free memory
- File management
  - create, delete, open, close, read, write file
- Device management
  - configure, read/write, connect/disconnect devices
- System setting and context
  - date/time, location, proximity, authentication service
- Communication
  - send/receive data messages

Types of System Calls (1/3)

- Process Control
  - Create, terminate, end, abort, load, execute
  - Get and set process attributes
  - Wait for time, event, signal event
  - Memory: allocate, free
  - Error dump, single-step for debug
  - Locks for shared data
- Protection
  - Get and set permissions, Allow and deny user access

Example process control on FreeBSD

- Unix variant, Multitasking
- User login
  - invoke user's choice of shell: bash, tcsh, ksh, ...
- Shell executes `fork()` to create process
  - `exec()` to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0: no error
  - code > 0: error code



Types of System Calls (2/3)

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

Types of System Calls (3/3)

- Information maintenance
  - get/set time, date, get/set system data
  - get/set process, file, or device attributes
- Communications
  - create, delete communication connection
  - (**Message passing**) send, receive messages
  - (**Shared-memory**)create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

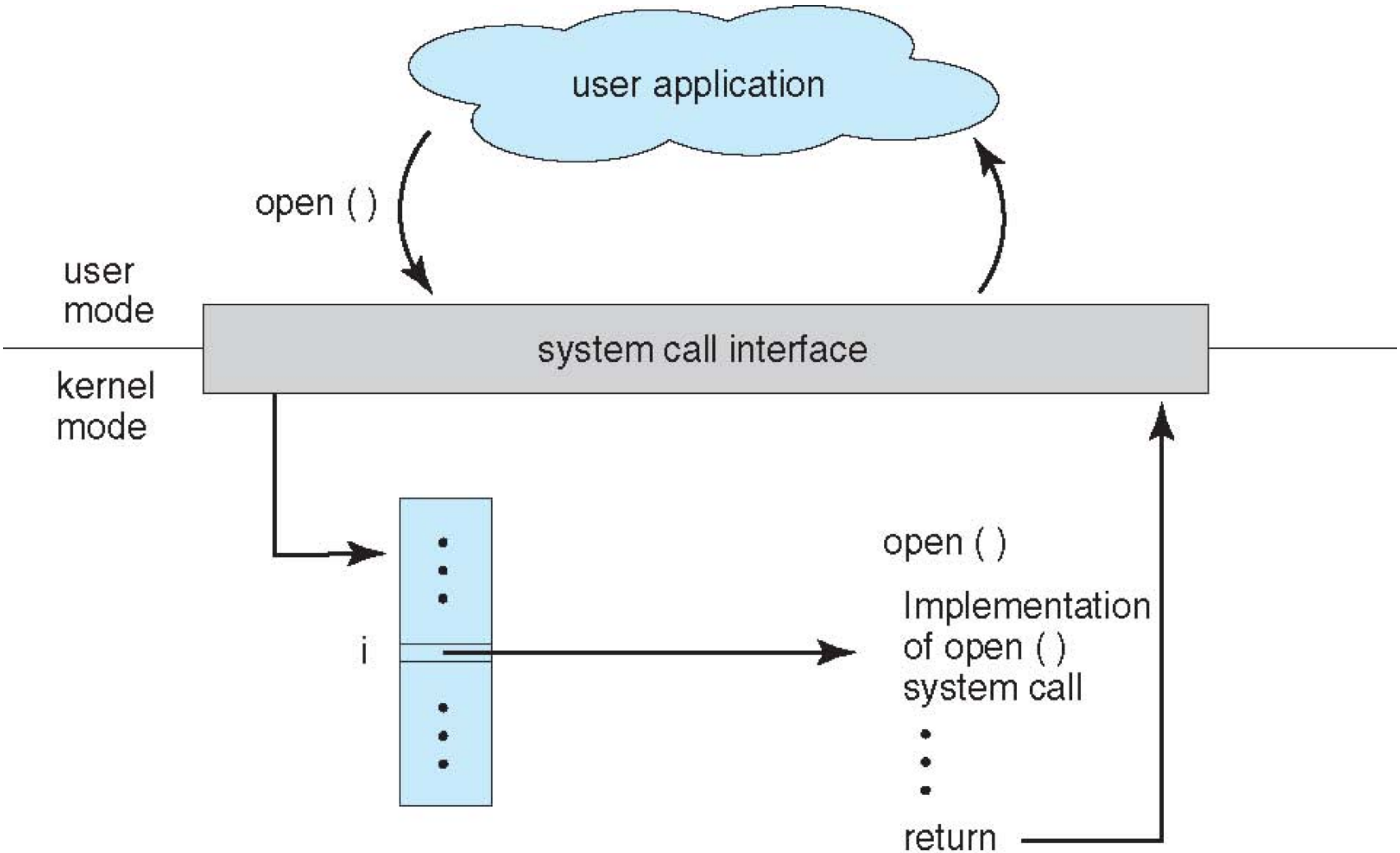
some systems calls have corresponding function or command

```
$ man -k chown                # keyword search
gchown(1), chown(1)           - change file owner and group
chown(8)                      - change file owner and group
$ man 2 chown
CHOWN(2)              BSD System Calls Manual              CHOWN(2)

NAME
  chown, fchown, lchown, fchownat -- change owner and group of a file

SYNOPSIS
  #include <unistd.h>
  int
  chown(const char *path, uid_t owner, gid_t group);
DESCRIPTION
  The owner ID and group ID of the file named by path or referenced by
  fildes is changed as specified by the arguments owner and group.  The
  ...
```

System Calls vs. API

- System calls
  - OS's interface to user code - traps to kernel
  - An explicit request to kernel made via trap `/usr/include/sys/syscall.h`
  - Generally done as assembly language instructions
- API
  - Set of **library** calls, <u>**with or without** system calls</u> (e.g., C library, standard-I/O library)
  - e.g., `malloc()` and `free()` /* not system calls */ both call `brk()` /* system call */
  - many math API's don't need system call

API-System Call-OS relationship



Run-time Environment (RTE)

- suite of software to run application
  - for applications written in specific language (or at least particular calling convention)
  - could include compiler, linker, interpreter, library, loader
- RTE support for system calls
  - maintains system-call numbers, provides calling interface

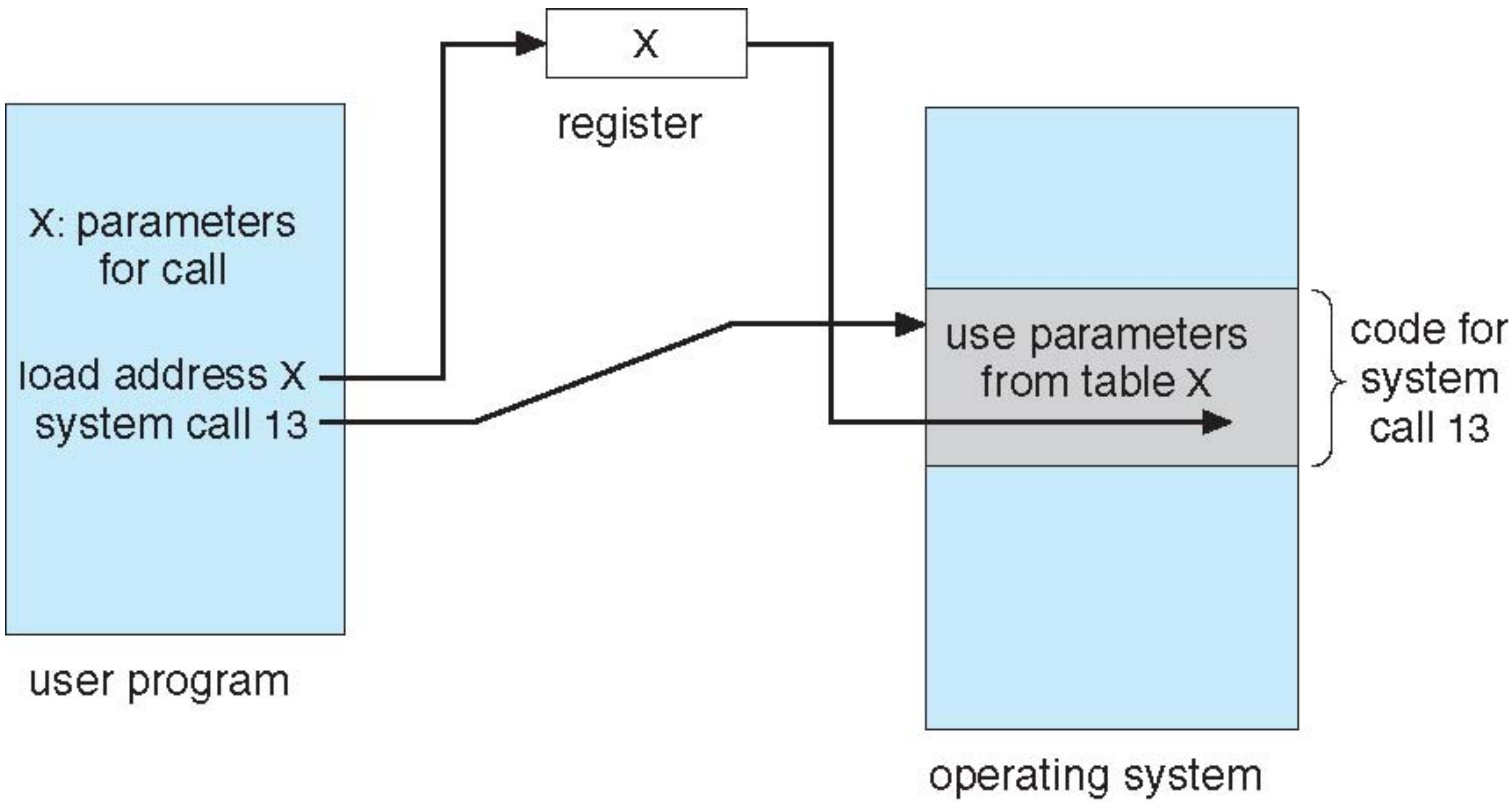
sys/syscall.h

```
/*
 * System call numbers.
 *
 * DO NOT EDIT-- this file is automatically generated.
 * created from  @(#)syscalls.master   7.26 (Berkeley) 3/25/91
 */

#define  SYS_exit 1
#define  SYS_fork 2
#define  SYS_read 3
#define  SYS_write    4
#define  SYS_open 5
#define  SYS_close    6
#define  SYS_wait4    7
                /* 8 is old creat */
#define  SYS_link      9
#define  SYS_unlink   10
                /* 11 is obsolete execv */
#define  SYS_chdir    12
#define  SYS_fchdir   13
#define  SYS_mknod    14
#define  SYS_chmod    15
#define  SYS_chown    16
#define  SYS_break    17
#define  SYS_getfsstat18
#define  SYS_lseek    19
#define  SYS_getpid   20
#define  SYS_mount    21
#define  SYS_unmount  22
#define  SYS_setuid   23
#define  SYS_getuid   24
#define  SYS_geteuid  25
#define  SYS_ptrace   26
#define  SYS_recvmsg  27
#define  SYS_sendmsg  28
#define  SYS_recvfrom 29
#define  SYS_accept   30
#define  SYS_getpeername   31
#define  SYS_getsockname   32
#define  SYS_access        33
#define  SYS_chflags       34
#define  SYS_fchflags      35
#define  SYS_sync          36
#define  SYS_kill          37
#define  SYS_stat          38
#define  SYS_getppid       39
#define  SYS_lstat         40
#define  SYS_dup           41
#define  SYS_pipe          42
#define  SYS_getegid       43
#define  SYS_profil        44
#define  SYS_ktrace        45
#define  SYS_sigaction     46
#define  SYS_getgid        47
#define  SYS_sigprocmask   48
#define  SYS_getlogin      49
#define  SYS_setlogin      50
#define  SYS_acct          51
#define  SYS_sigpending    52
#define  SYS_sigaltstack   53
#define  SYS_ioctl         54
#define  SYS_reboot        55
#define  SYS_revoke        56
#define  SYS_symlink       57
#define  SYS_readlink      58
#define  SYS_execve        59
#define  SYS_umask         60
#define  SYS_chroot        61
#define  SYS_fstat         62
#define  SYS_getkerninfo   63
#define  SYS_getpagesize   64
```

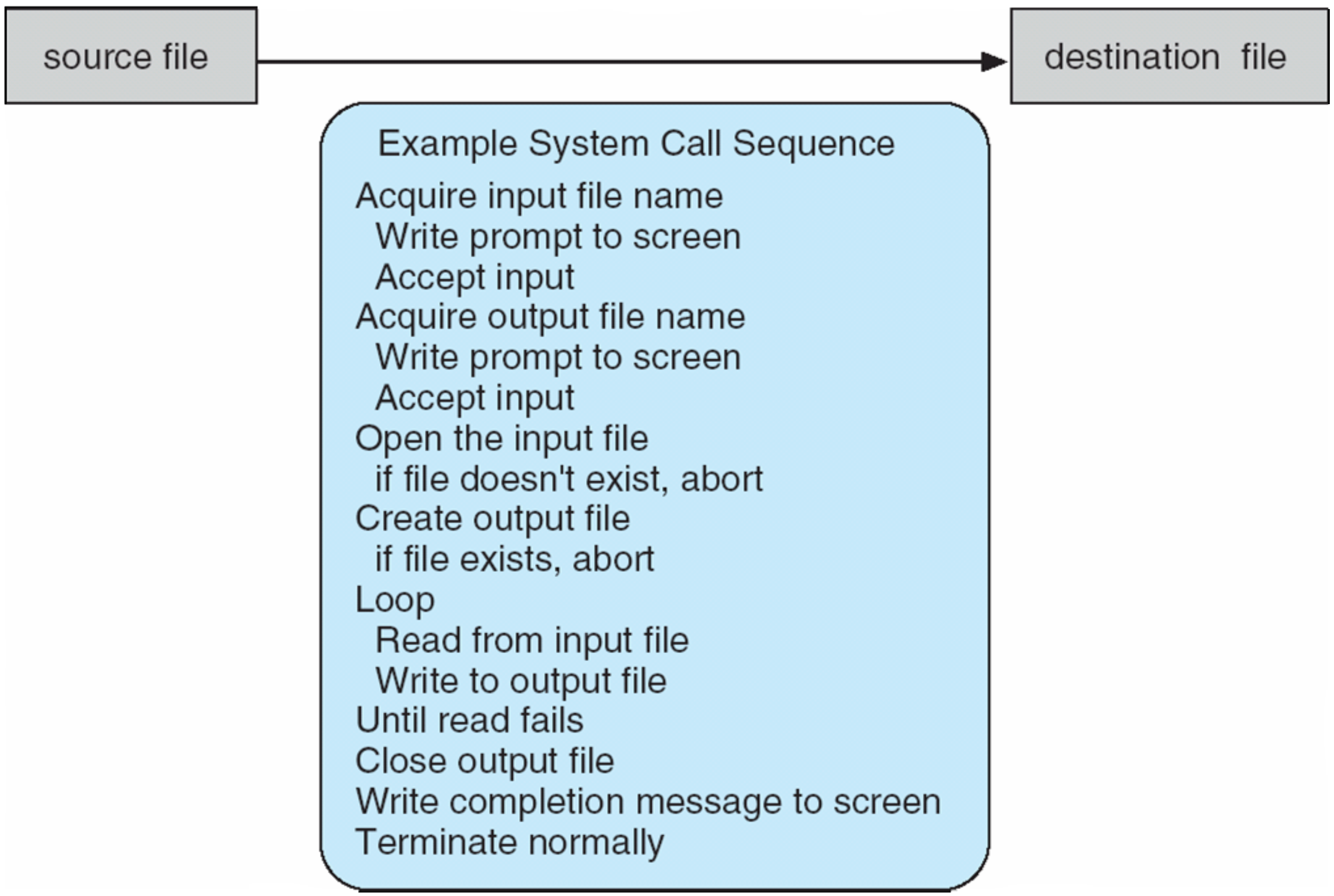
System Call: Passing Parameters

- in registers
  - (defined by the ISA)
- in Table
  - Store parameter values in a table in memory,
  - pass the table's address in a register
- on stack
  - User code pushes the parameters onto the stack,
  - OS pops params off the stack on return

Parameter Passing to System Call via Table



Example of System Calls used by a file-copy operation



Example standard API: read()

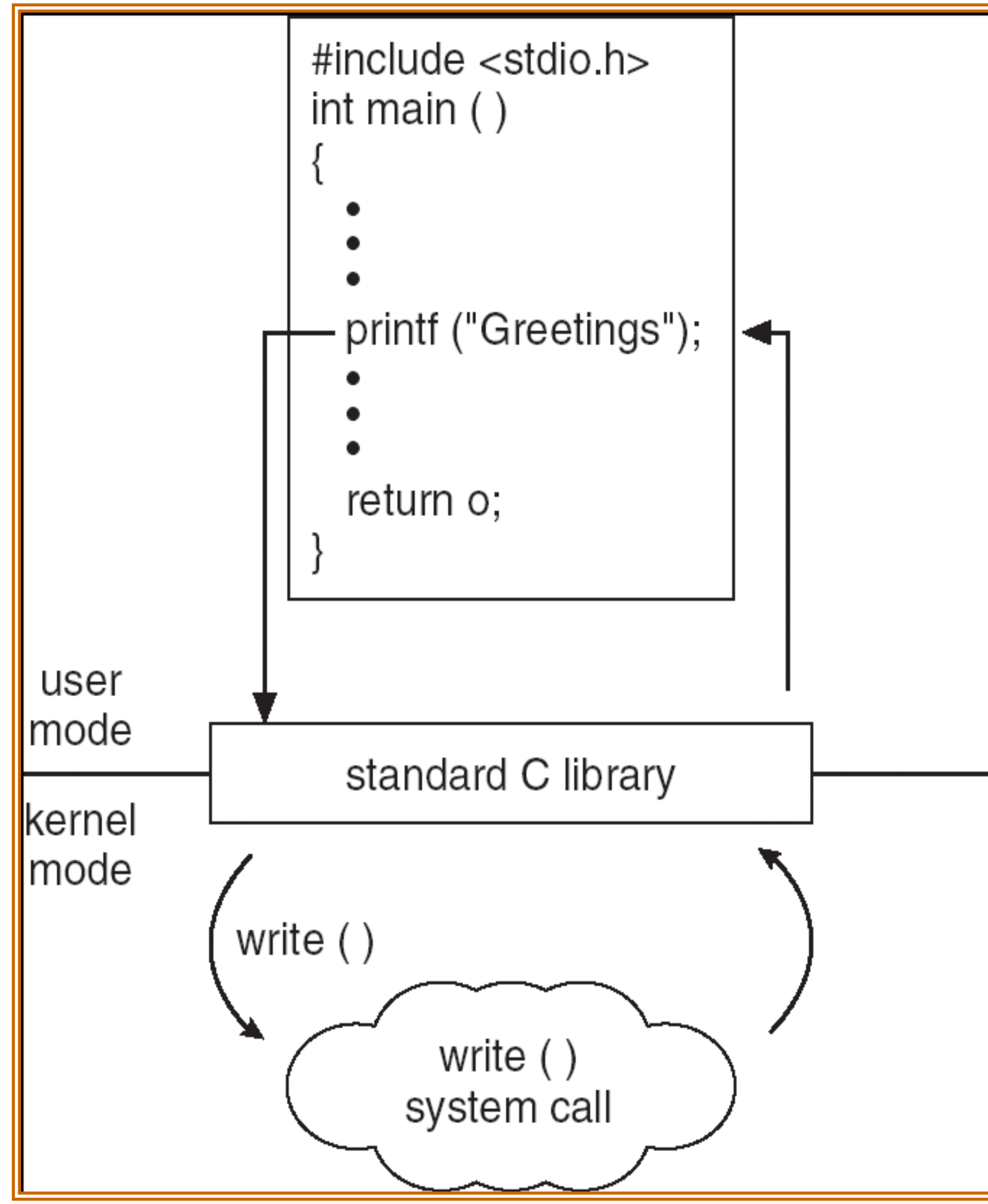
- Unix standard C library
  - #include <unistd.h>
  - `ssize_t read(int fd, void *buf, size_t count);`
- Parameters
  - `int fd;` // the file descriptor
  - `void *buf;` // pointer to data buffer to store read data
  - `size_t cout;` // max #bytes to read
- Return value:
  - #bytes, 0 of EOF, -1 if error.

Calls related to read()

- Open and close the file
  - `int open(const char *path, int flags);`
  - `int close(int fd);`
- Reading and Writing
  - `ssize_t read(int fd, void *buf, size_t count);`
  - `ssize_t write(int fd, const void *buf, size_t count);`

Standard C library example

- C program calls `printf()`
  - in `stdio` library
  - printf not a system call
- `printf()` calls `write()`
  - write() is an actual system call



Popular APIs

- Win32 API
  - defined by Microsoft for Windows UI, I/O, disk, …
- POSIX API
  - Posix = **P**ortable **OS** **I**nterface for Uni**x**
  - for most Unix-based systems, incl. Linux, macOS
- Java API
  - UI, I/O, … for Java virtual machine (JVM)
  - many are mapped to the host OS's API

Why use API?

- Simplicity
  - designed for application programmers
- Portability
  - same standard API (e.g., POSIX) across different platforms
- Efficiency
  - system call is more expensive; API might not need to make system call, could be more efficient
  - example: `sprintf()` doesn't perform I/O, just formatting
  - some API are designed for convenience

Review (1)

- What are the two communication models provided by OS?
- What is the relationship between system calls, API, and C library?
- Why use API rather than system calls directly?

System Programs (aka system services, system utilities)

- A layer of programs above system calls, for purpose of
  - Convenient for program development and execution
  - Defines most users' view of OS
- Categories
  - File manipulation, Status information
  - Programming language support, loading and execution
  - Communications
  - Background services
  - Application programs

System Programs (1/4)

- File management
  - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
  - Get date, time, available mem, disk space, #users
  - Performance, logging, and debugging information
  - Registry for configuration information

System Programs (2/4)

- File search and edit
  - Text editors (vim), search contents, transform text
- Programming-language support, loading, execution
  - Compilers, assemblers, debuggers (gdb) and interpreters (python)
  - Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

System Programs (3/4)

- Communications
  - Creating connections among processes, users
    - Interprocess vs. Network communication
  - examples: Send messages, browse web pages, send email, remote login, FTP
  - Main Models of communication:
    - message-passing vs. shared-memory

System Programs (4/4)

- Background Services
  - Launch at boot time
    - Some for system startup, then terminate
    - Some from system boot to shutdown
  - Disk checking, process scheduling, error logging, printing
- (bundled) Application programs
  - Not typically considered part of OS
  - Launched by command line, mouse click, finger poke,

ABI: Application Binary Interface

- definition for executable program
  - executable file format (e.g., ELF, COFF, Mach-O, EXE, PE)
  - ISA of the program code (native, bytecode, etc); could be "fat binary"
  - parameter passing convention (stack, register, ..)
  - data types (sizes, endian)
- Tools involved
  - linker: resolves addresses of all symbols
  - loader: loads linked image into memory to execute

System Structure

Simple OS Architecture
More Complex OS Architecture
Layer OS Architecture
Microkernel OS
Modular OS Structure
Virtual Machine
Java Virtual Machine

User Goals and System Goals

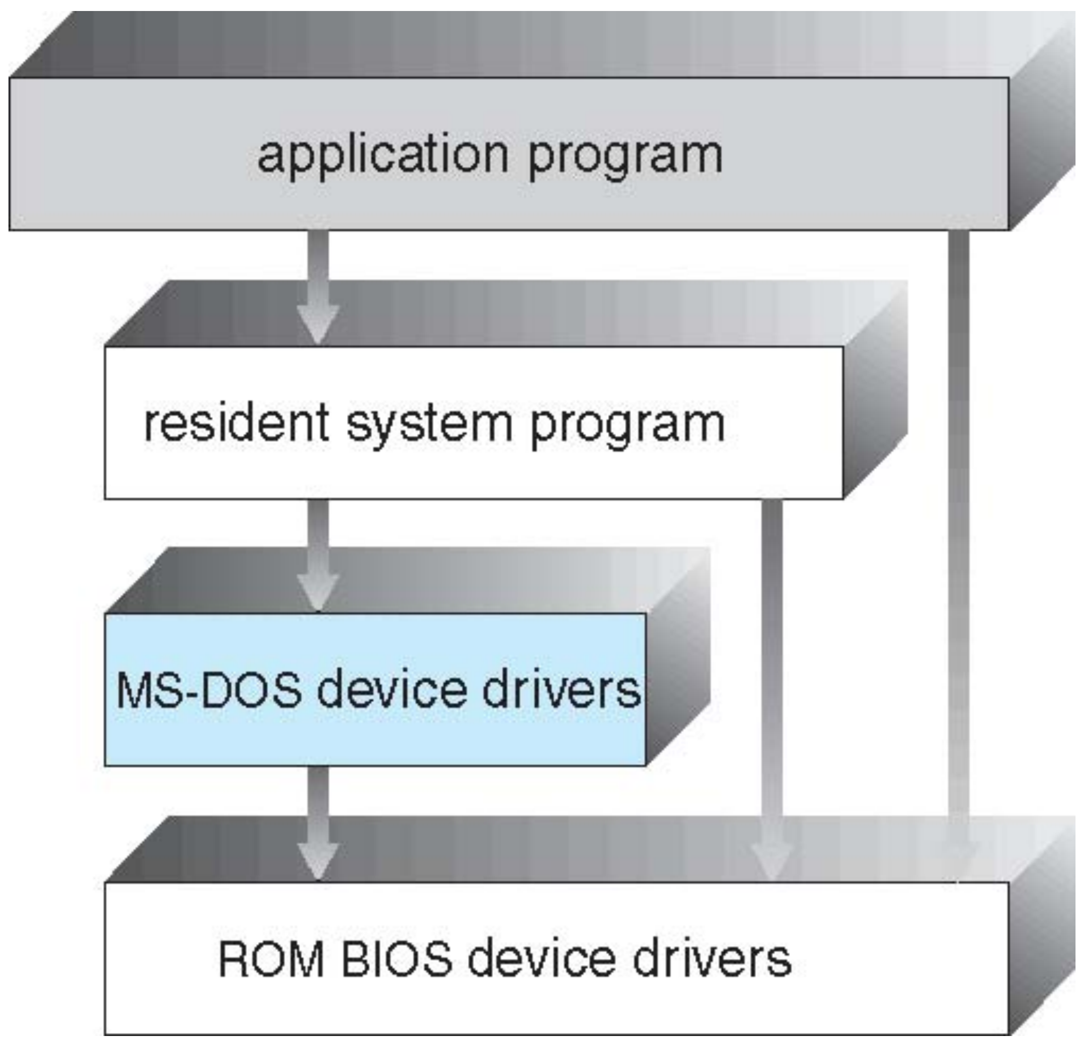
- User Goals
  - OS should be easy to use and learn
  - OS should be reliable, safe, and fast
- System Goals
  - OS should be easy to design, implement, maintain
  - OS should be reliable, error-free, and efficient

Separation of Policy and Mechanism in OS design

- Policy
  - **What** will be done? **What** is allowed? (parameterizable)
- Mechanism:
  - **How** to do it? (implementation)
- Important principle
  - it allows maximum **flexibility** if policy decisions are to be changed later (example – timer)
  - not always so separated in commercial OS but desirable as good practice of OS design

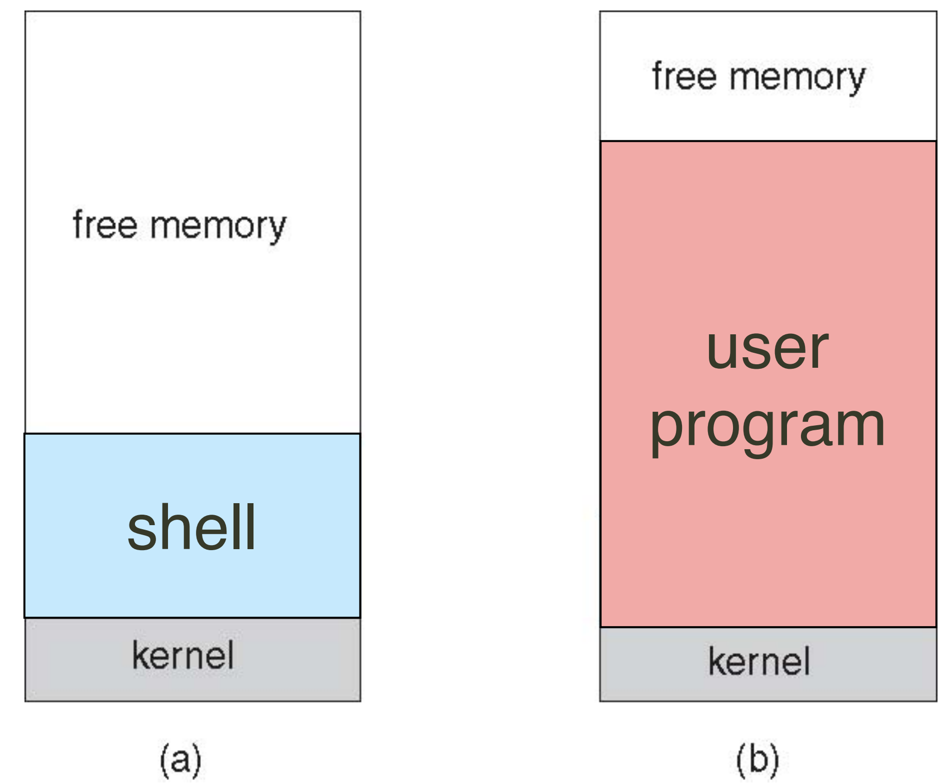
Simple Structure -- MS-DOS

- Goal:
  - Uses the least space
- Minimal structure
  - Not divided into modules
  - interfaces and levels of functionality are not well separated
- Drawbacks
  - unsafe, difficult to enhance



Example: MS-DOS

At system startup running a program



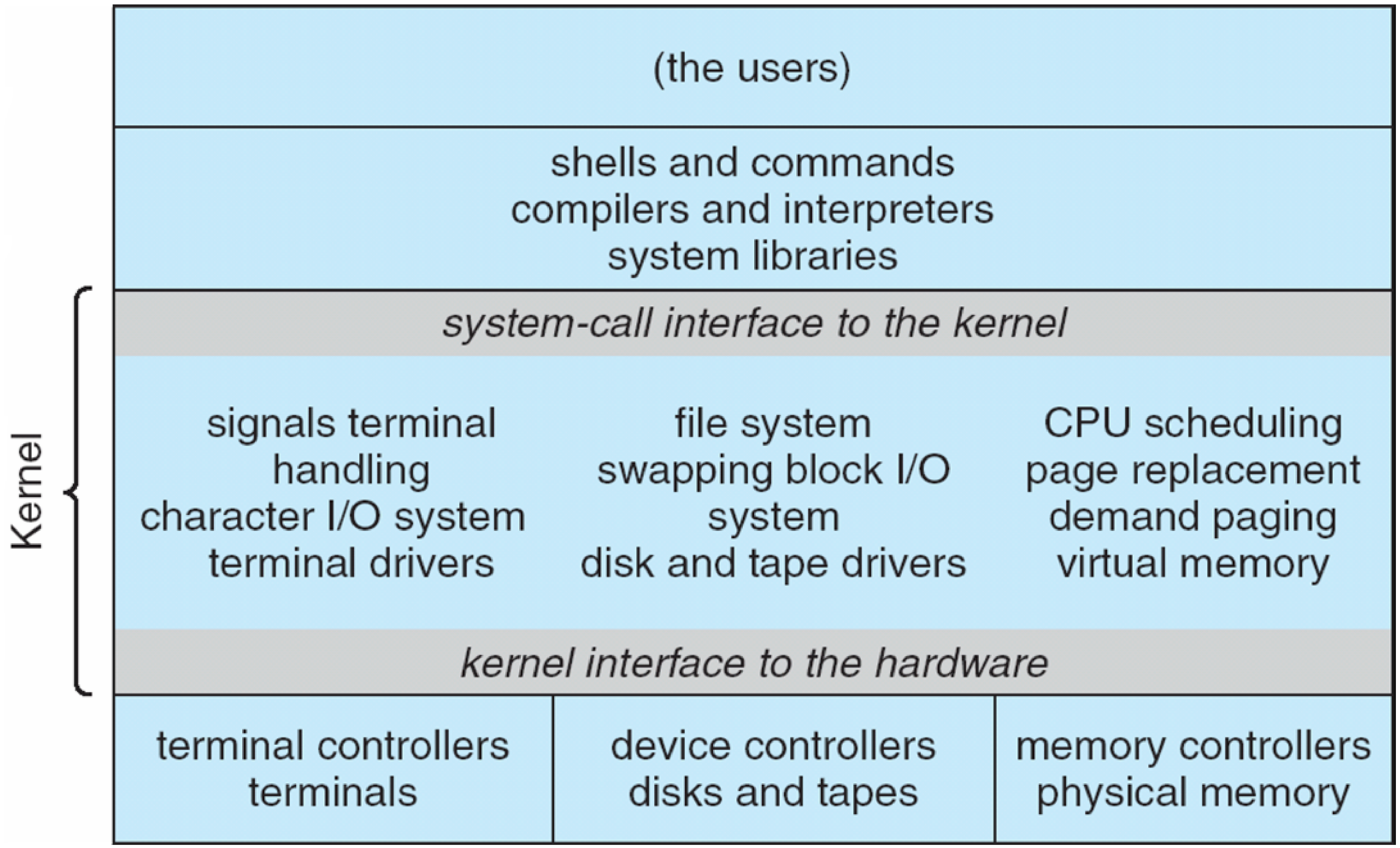
- Single-tasking
- Shell invoked on booting
- Simple way to run program
  - No process created
  - Single memory space
  - Loads program into memory, overwriting all but the kernel
- Program exit -> shell reloaded

Monolithic Structure

- Two layers: users mode vs. kernel mode.
  - the entire OS kernel runs in one address space
  - "tightly coupled" large #functions for one level
- Examples
  - traditional Unix (difficult to scale complexity)
  - Linux (monolithic for performance but modular)
  - Windows (also monolithic but got more modular)

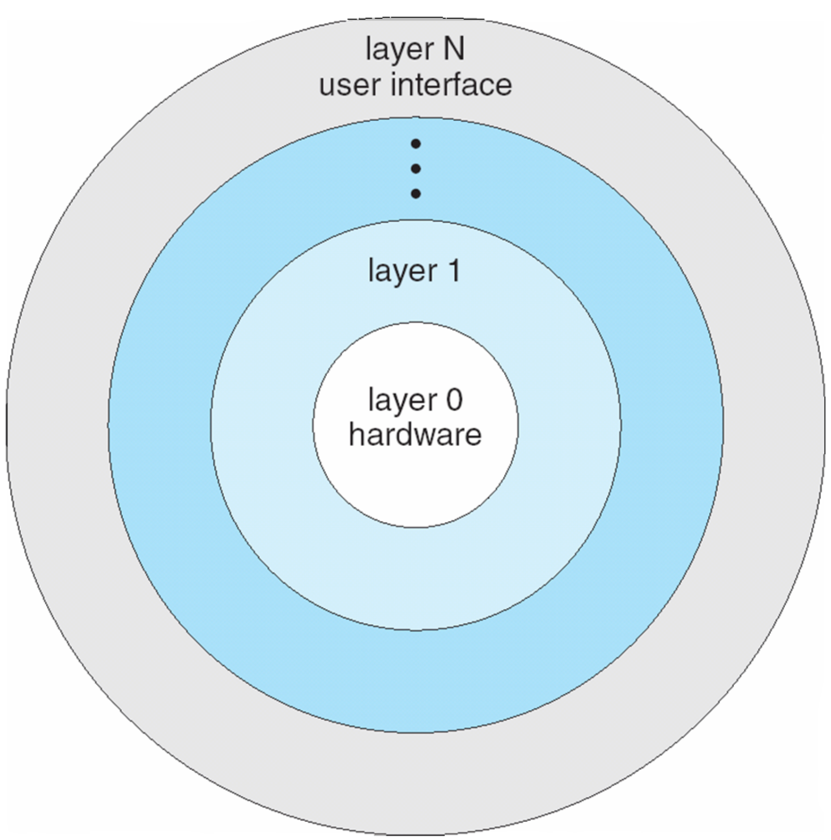
Traditional UNIX Structure

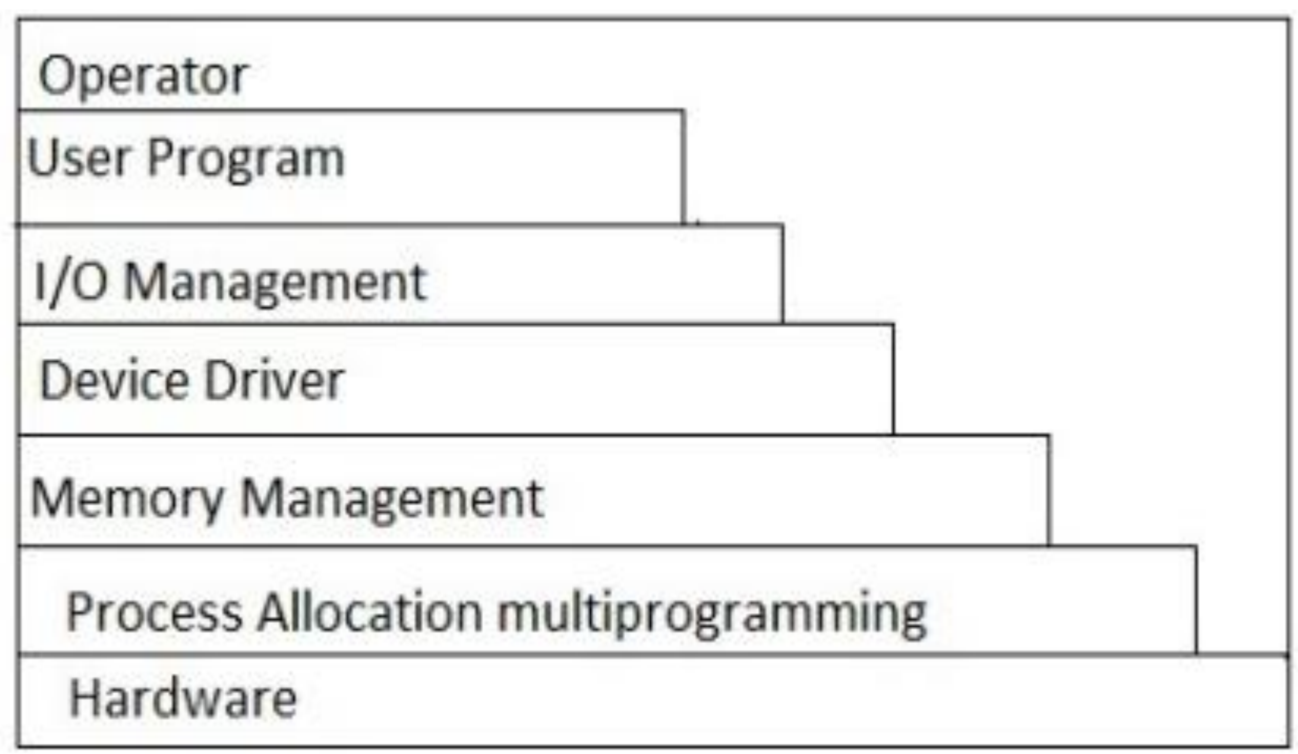
Beyond simple but not fully layered



Layered OS Architecture

- OS divided into N layers (0..N-1)
  - Layer 0 = hardware, N-1 = user interface
  - Lower layers independent of upper layers
  - Higher layer use services only of lower layers



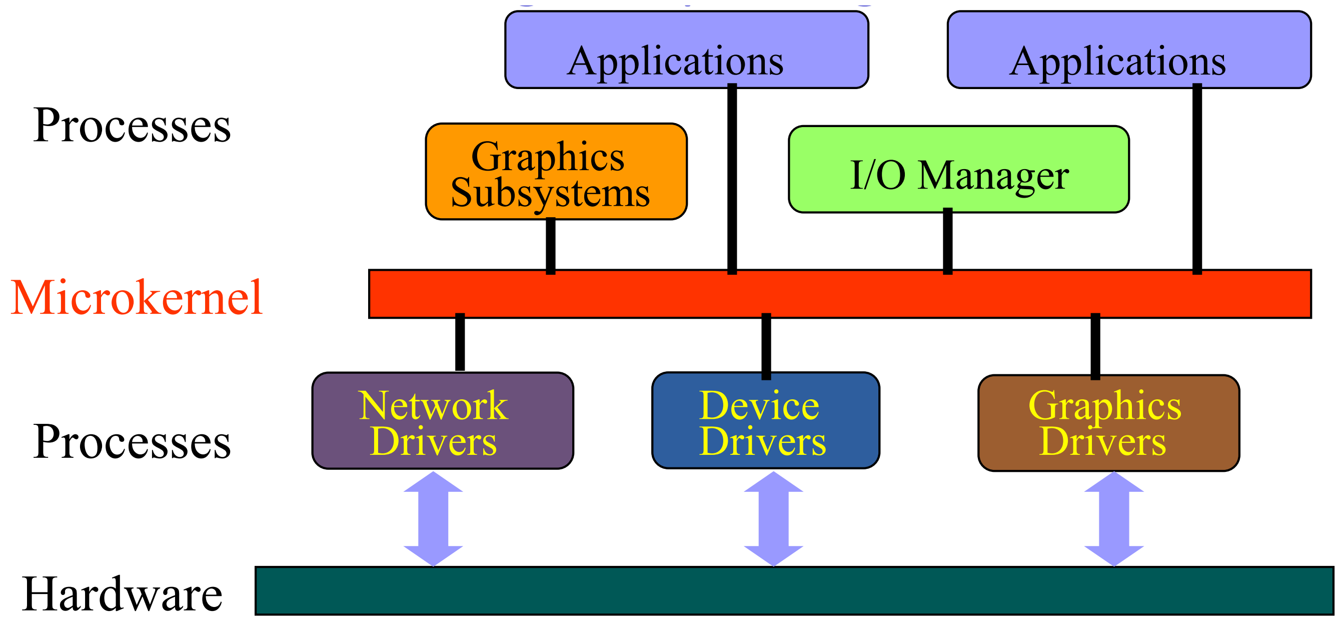


Trade-offs of Layered Approach

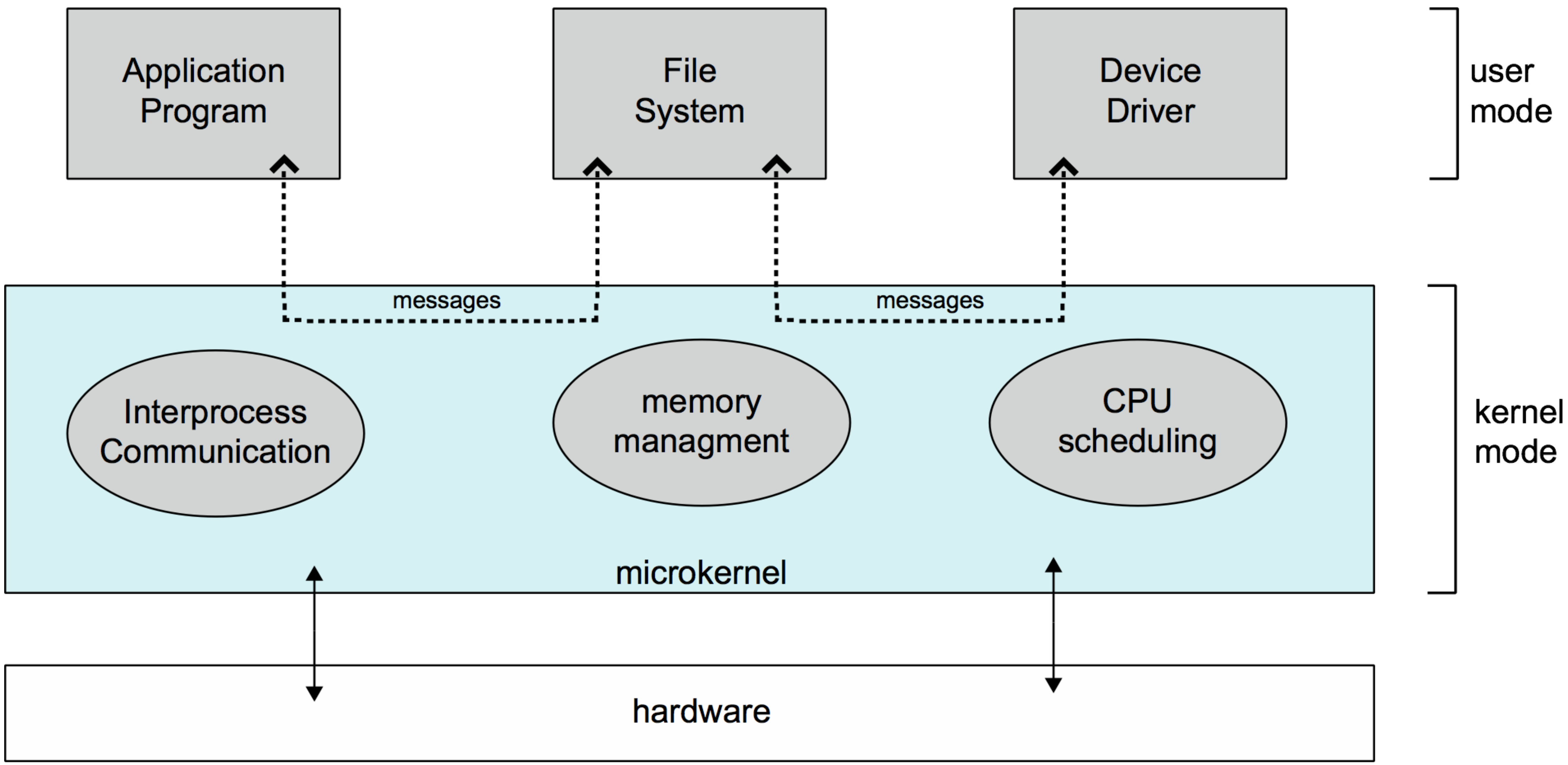
- Pro
  - easier debugging and maintenance
  - successful example: TCP/IP protocol stack
- Con
  - less efficient
  - difficult to define layers

Microkernel OS

- Approach
  - Move as much from the kernel into user space
  - Communication provided by message passing
- Example:
  - Mach, mk-Linux



Microkernel System Structure



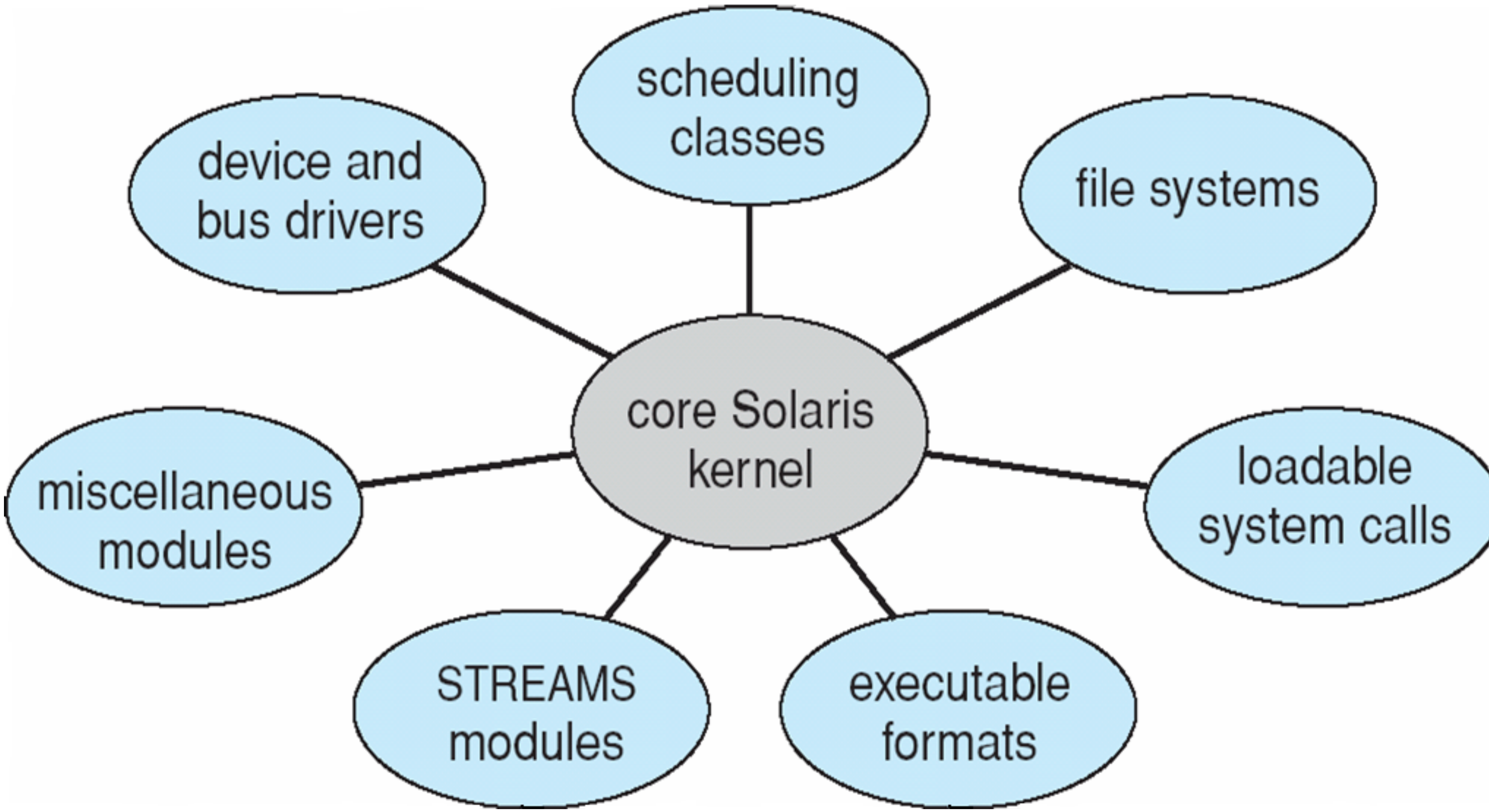
Trade-offs of Microkernel

- Pro:
  - Easier for extending and porting to new architecture
  - More reliable - less code runs in kernel mode, More secure
- Con:
  - less efficient than monolithic kernel, due to more message passing for user-to-kernel communication
- Example use:
  - Linux: monolithic for performance (but modular)
  - Windows NT started out microkernel, but XP became more monolithic for performance

Modular OS Architecture

- Supports **loadable kernel modules (LKM)**
  - Kernel = core components + LKM interfaces
  - LKM is loaded as needed, can be unloaded (e.g., USB driver)
- Combines advantages of microkernel and layered
  - load in modules as needed, no need to recompile
  - lower overhead: no need for message passing
- Similar to layers but more flexible
  - e.g., Solaris, Linux, Windows

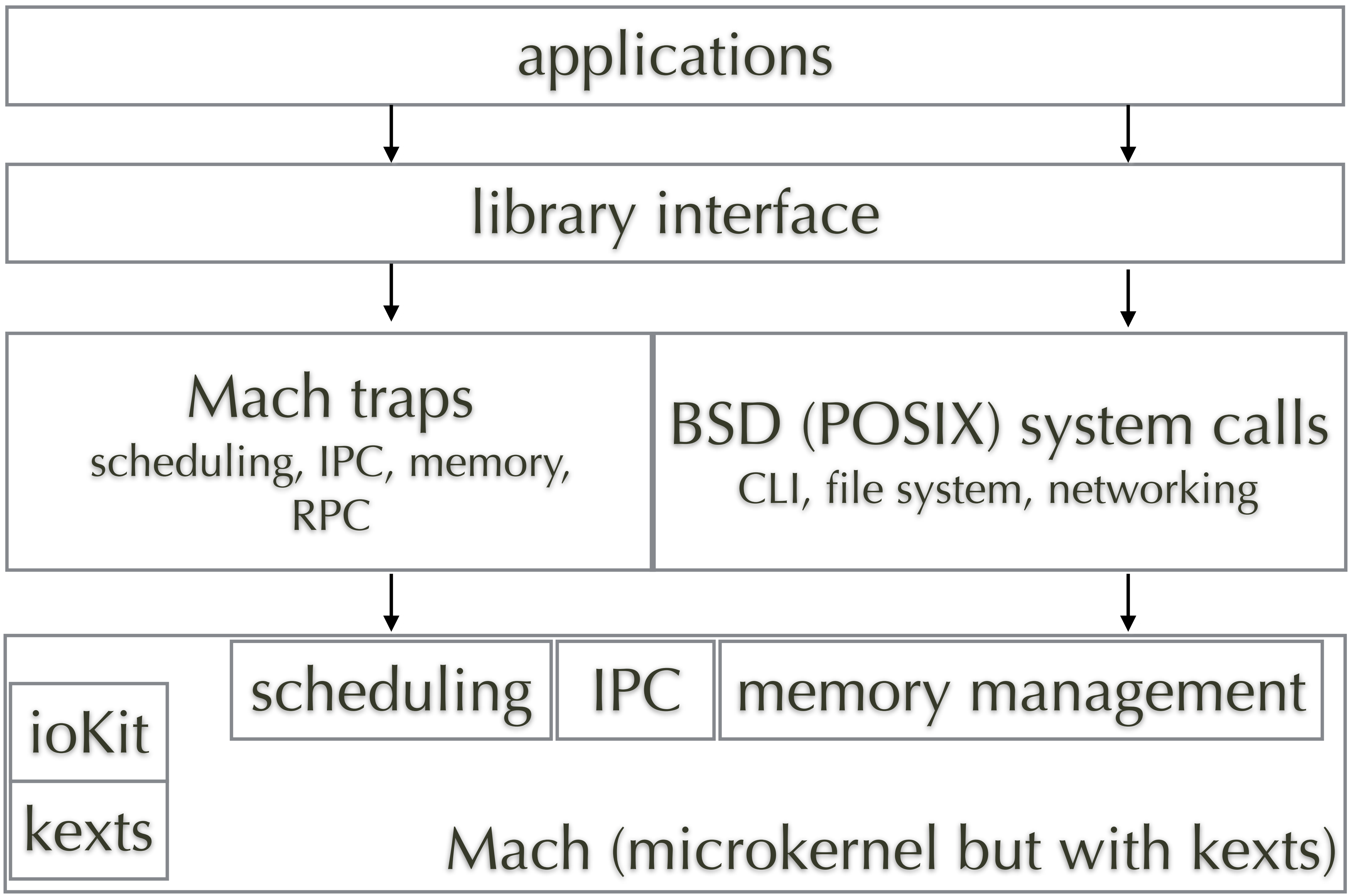
Solaris Modular Approach



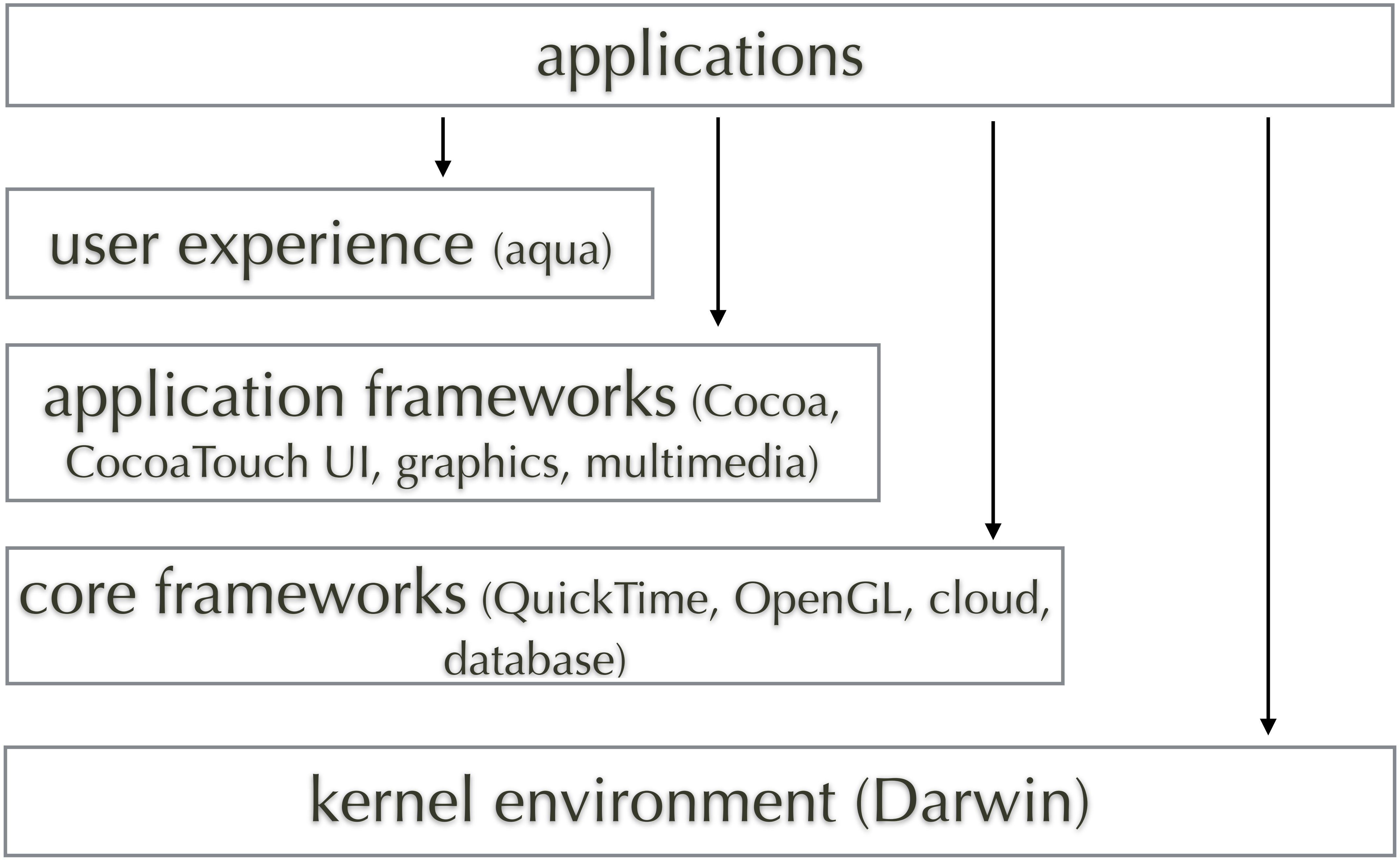
Hybrid Systems

- Most modern OSs are some mix of models
  - Hybrid to address performance, security, usability needs
- Examples of Monolithic + loadable module
  - Linux and Solaris: monolithic + modular for dynamic loading
  - Windows: monolithic + μkernel for subsystem "personalities"
- Example of Microkernel + layered + loadable
  - Darwin (macOS, iOS): Mach microkernel and BSD Unix (POSIX)
  - I/O kit and kernel extensions (i.e. dynamically loadable modules)
- Example of Monolithic + layered
  - Android: Linux kernel, somewhat layered

Darwin (kernel for macOS, iOS)



macOS and iOS



"layers" but not strictly... can bypass

Android

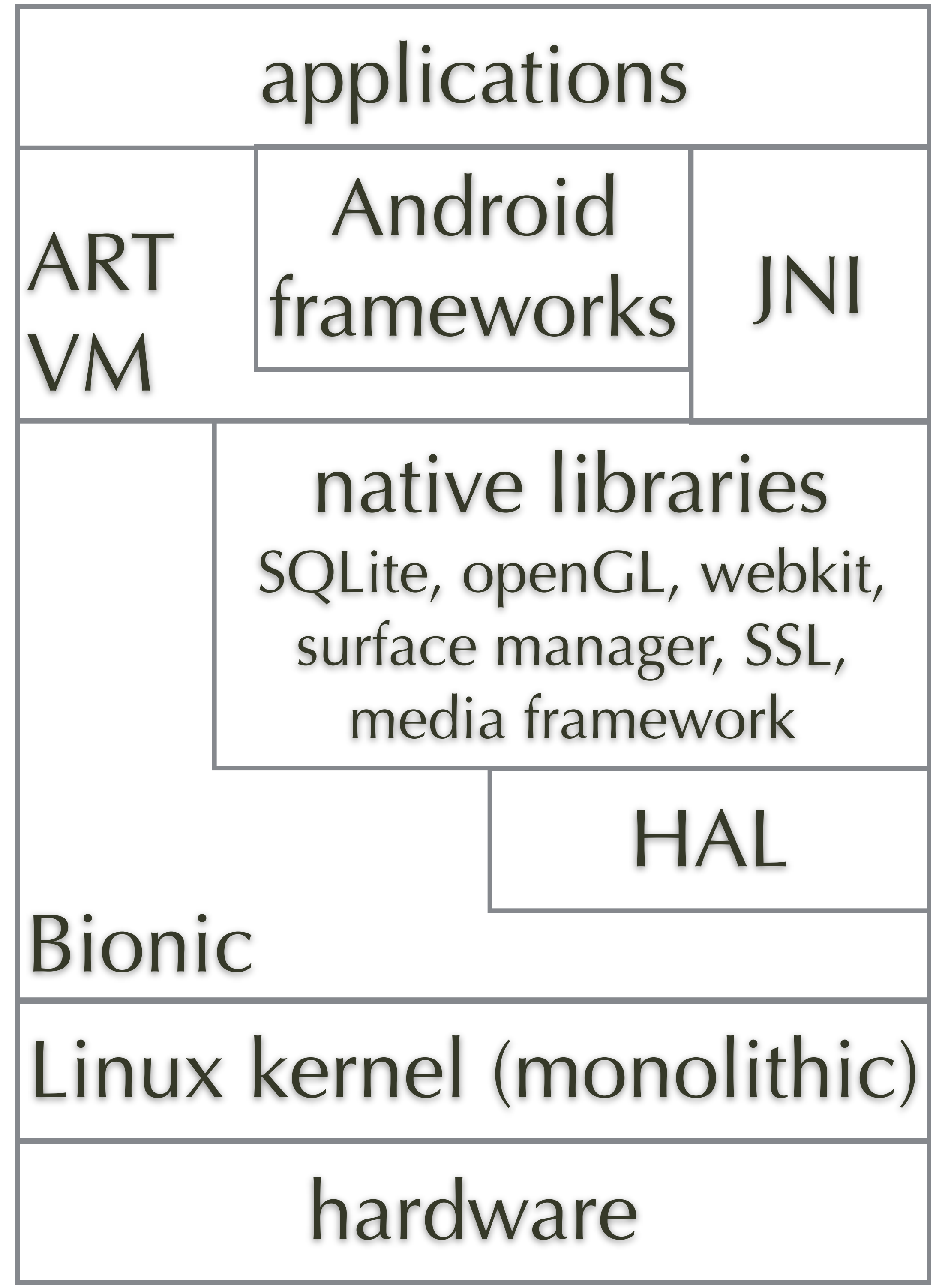
- Google acquired Android Inc. 2005
  - for mobile phone with keyboard, no touch screen
- After iPhone (2007), total redesign
  - changed to touch screen, Open Handset Alliance
  - changes to Linux kernel to support power management
  - Google Mobile Services, ported to phones, TV, watches, ...
- Mixed License
  - open source for Android core
  - proprietary (GooglePlay, Google Mobile Services)

Android

- Runtime env. includes core set of libraries and Dalvik VM
  - Apps developed in Java plus Android API
  - Java class files compiled to Java bytecode translated to executable, runs in Dalvik VM
- Libraries
  - frameworks for webkit, SQLite, multimedia, smaller libc

Android Architecture

- applications written in Java language
  - not standard Java API
- ART = Android RunTime virtual machine
  - ahead-of-time (AOT) compilation to native code
- JNI = Java native interface
- Bionic replaces standard C library
  - smaller than glibc for mobile; bypasses GPL (Gnu Public License)
- HAL = hardware abstraction layer

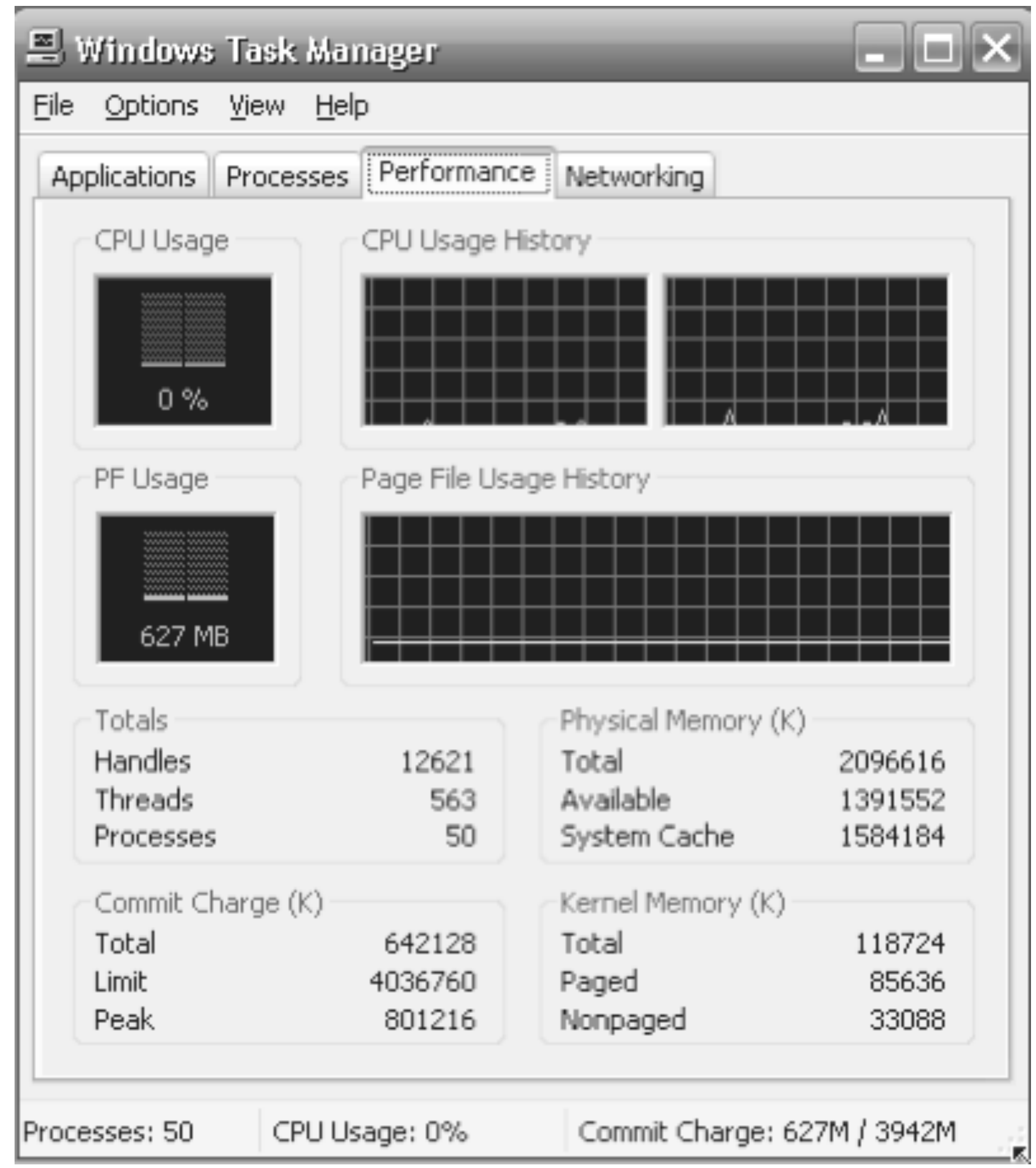


Operating-System Debugging

- Failure analysis: Finding and fixing errors
  - User program crash: (1) **log** files error info
    (2) **Core dump** file captures memory of the process
    => debugger reads it
  - OS failure: can generate **crash dump** file containing kernel memory
- Performance monitoring and tuning
  - Using trace listings of activities, recorded for analysis
  - Profiling = instruction trace for statistical trends

Performance Monitoring & Tuning

- Purposes
  - Improve performance by removing bottlenecks
- OS must provide
  - means of computing and displaying measures of system behavior
- Example:
  - "top" program or Windows Task Manager



Performance Monitor: Counters

- per-process
  - `ps`: information about individual processes
  - `top`: statistics for current processes
- system-wide
  - `vmstat`: report memory-usage statistics
  - `netstat`: statistics for network interfaces
  - `iostat`: I/O usage statistics for disks

Tracing

- Purpose:
  - observe specific events
- Per-process
  - `strace`: traces **system calls**
  - `gdb`: GNU source-level debugger
- System-wide
  - `perf`: collection of performance tools for Linux
    - subcommands: `stat`, `top`, `record`, `report`, `annotate`, `sched`, `list`
  - `tcpdump`: traces **network** packets

Summary of Chapter 2

- OS provides services for program execution
  - System call (entry into OS); API
  - System programs: linker, loader
- OS structures
  - monolithic (two level, no structure)
  - microkernel (minimal core, separate address spaces for each service)
  - modular (dynamic loadable, same address space)
  - hybrid (monolithic or microkernel, combined with modular)
- Tools
  - counters, monitors, trace