Principles of Operating Systems

CS 446/646

1. Introduction to Operating Systems

a. Role of an O/S

b. O/S History and Features

c. Types of O/S

d. Major O/S Components

e. System Calls

- ✓ System calls are the O/S API
- ✓ Sample of UNIX system calls
- ✓ Equivalent Windows system calls
- ✓ System programs

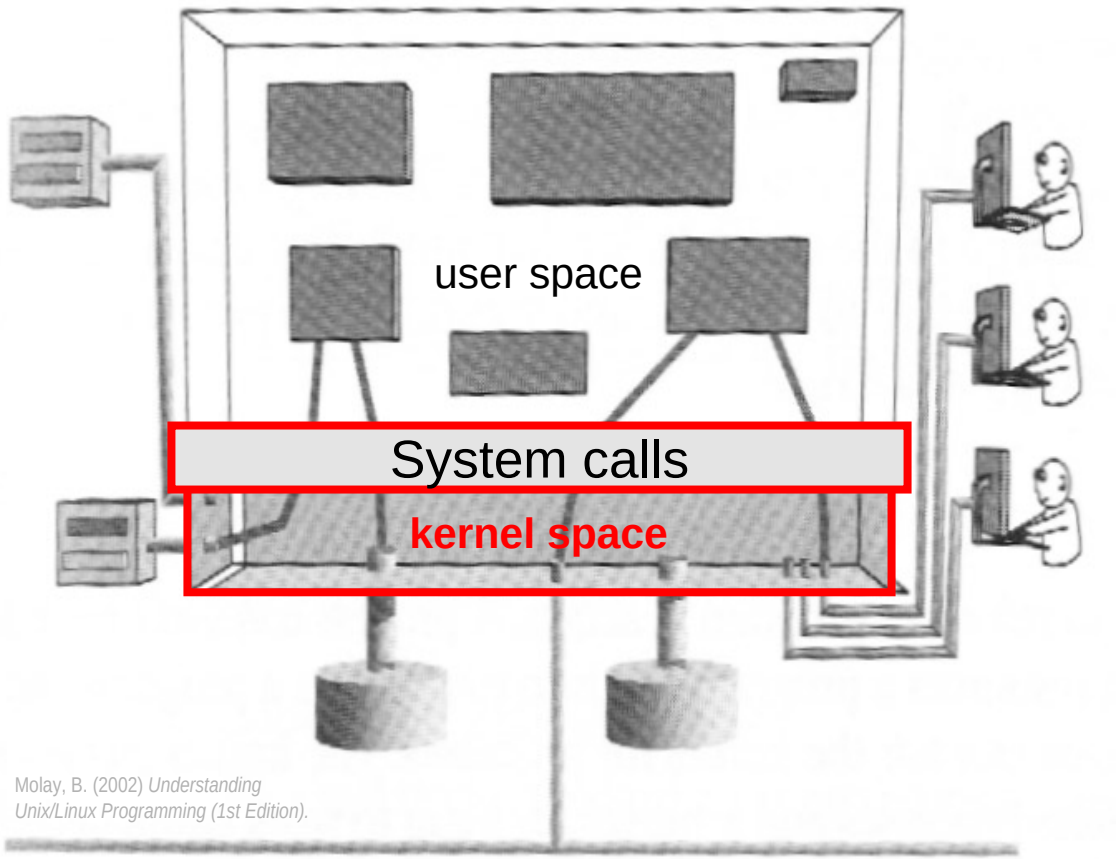
f. O/S Software Architecture

g. Examples of O/S

1.e System Calls

System calls are the O/S API

- Location of the system calls in the Molay view



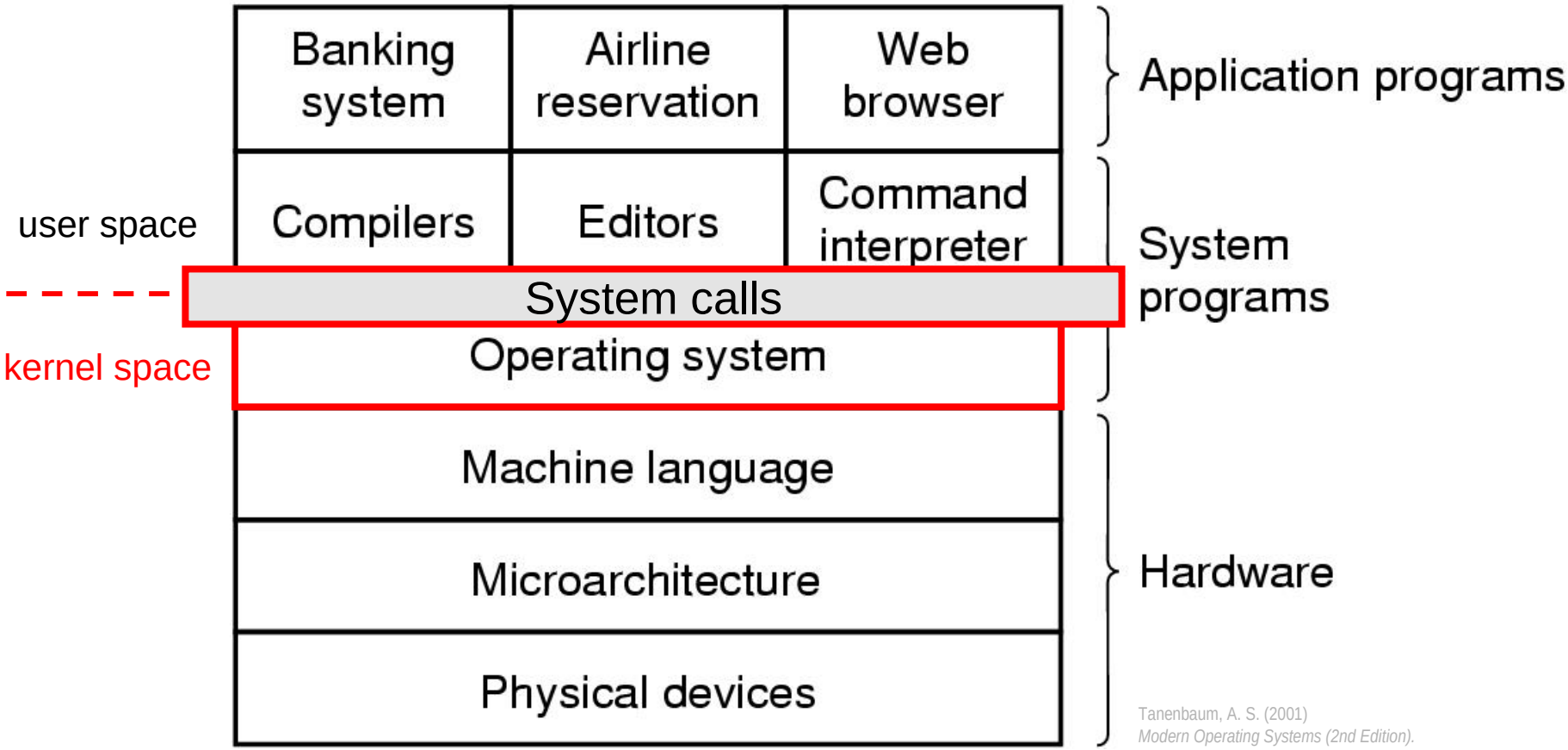
Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

The system calls are the mandatory interface between the user programs and the O/S

1.e System Calls

System calls are the O/S API

- Location of the system calls in the Tanenbaum view

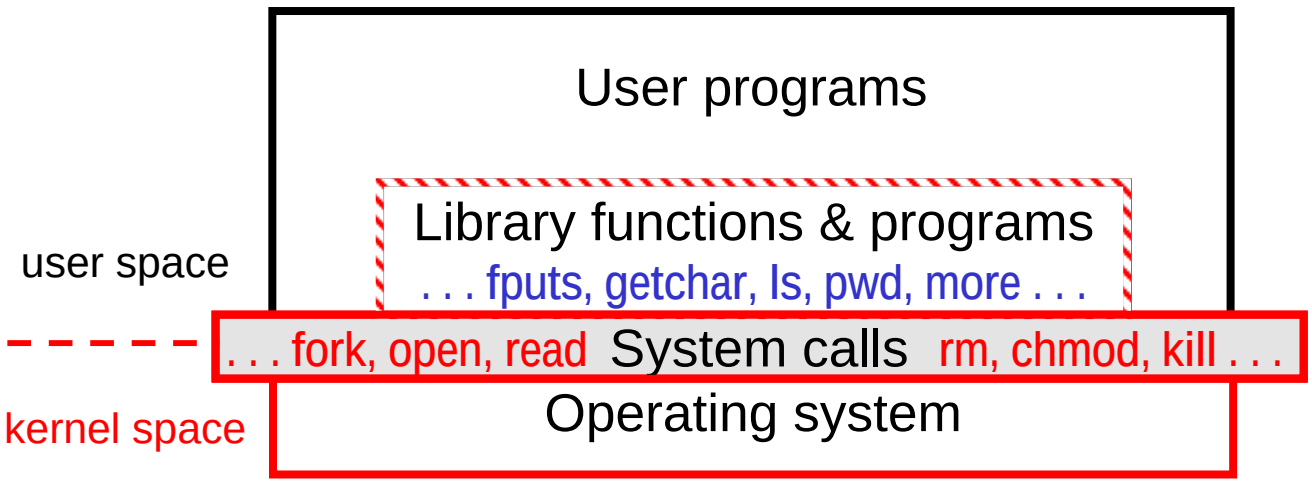


The system calls are the mandatory interface between the user programs and the O/S

1.e System Calls

System calls are the O/S API

- All programs needing resources must use system calls



the "middleman's counter"

- ✓ system calls are the only entry points into the kernel and system
- ✓ most UNIX commands are actually library functions and utility programs (e.g., shell interpreter) built on top of the system calls
- ✓ however, the distinction between library functions and system calls is not critical to the programmer, only to the O/S designer

1.e System Calls

System calls are the O/S API

```
...
int main(...)
{
    ...
    if ((pid = fork()) == 0)                          // create a process
    {
        fprintf(stdout, "Child pid: %i\n", getpid());
        err = execvp(command, arguments);             // execute child
                                                      //   process
        fprintf(stderr, "Child error: %i\n", errno);
        exit(err);
    }
    else if (pid > 0)                                 // we are in the
    {                                                 //   parent process
        fprintf(stdout, "Parent pid: %i\n", getpid());
        pid2 = waitpid(pid, &status, 0);              // wait for child
        ...                                           //   process
    }
    ...

    return 0;
}
```

Sample 1: implementing a shell command interpreter with system calls and library functions

1.e System Calls

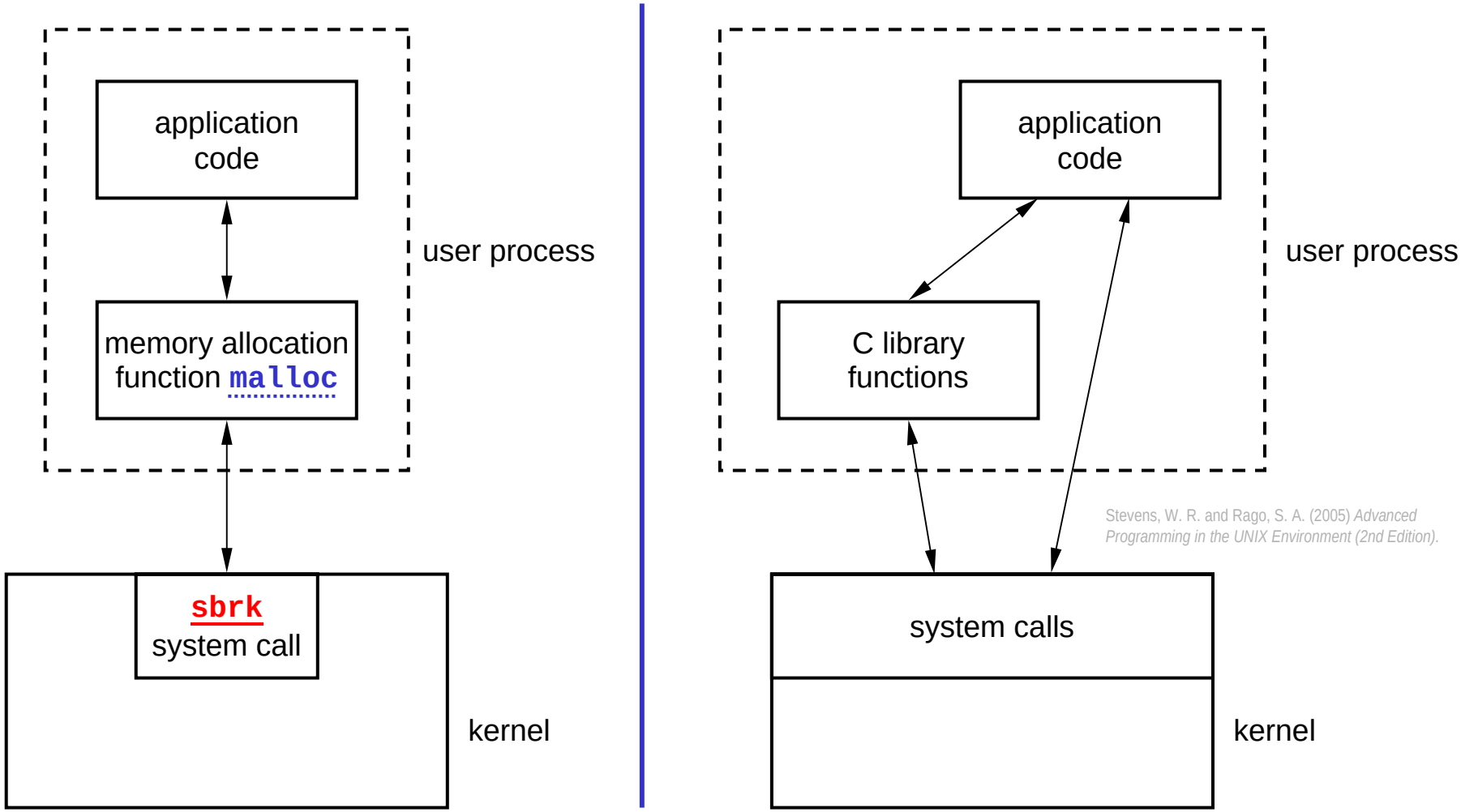
System calls are the O/S API

```
...
int main(int ac, char *av[])
{
    ...
    if ((fp = fopen(*av, "r")) != NULL) {          // open file
        while (fgets(line, 512, fp)) {             // read next line
            if (num_of_lines == 24) {              // is screen full?
                while ((c = getchar()) != EOF) {   // prompt user
                    switch (c) {
                    case 'q': ...                  // prepare to quit
                    case ' ': ...                  // prepare to show
                    ...                            //   one more screen
                    }
                }
            }
            if (fputs(line, stdout) == EOF)        // show line
                exit(1);                           //   or finish
            }
            num_of_lines++;                        // count line
        }
        fclose(fp)                                 // close file
        ...
```

Sample 2: implementing the **more** command with system calls and library functions

1.e System Calls

System calls are the O/S API



Stevens, W. R. and Rago, S. A. (2005) *Advanced Programming in the UNIX Environment (2nd Edition).*

Difference between C library functions and system calls

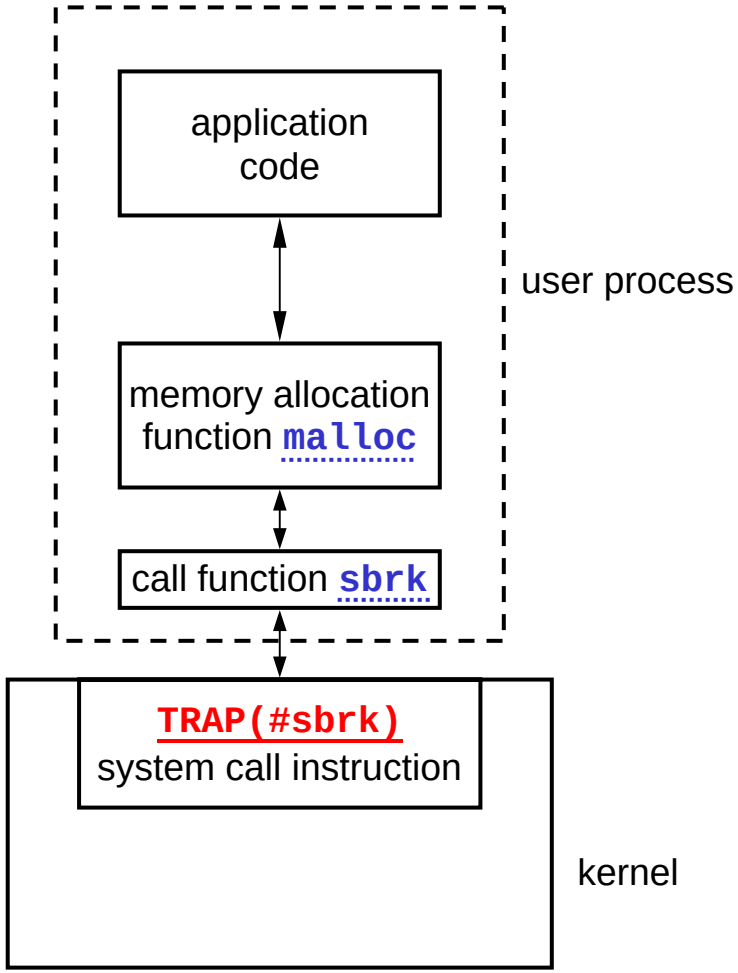
1.e System Calls

System calls are the O/S API

- System calls
  - ✓ system calls offer back-end, low-level services
  - ✓ for example: **`sbrk`** only increases or decreases a process's address space by a specified number of bytes
- Library functions
  - ✓ library functions offer front-end services that contain higher-level logic
  - ✓ for example: **`malloc`** implements a particular memory allocation strategy

1.e System Calls

System calls are the O/S API



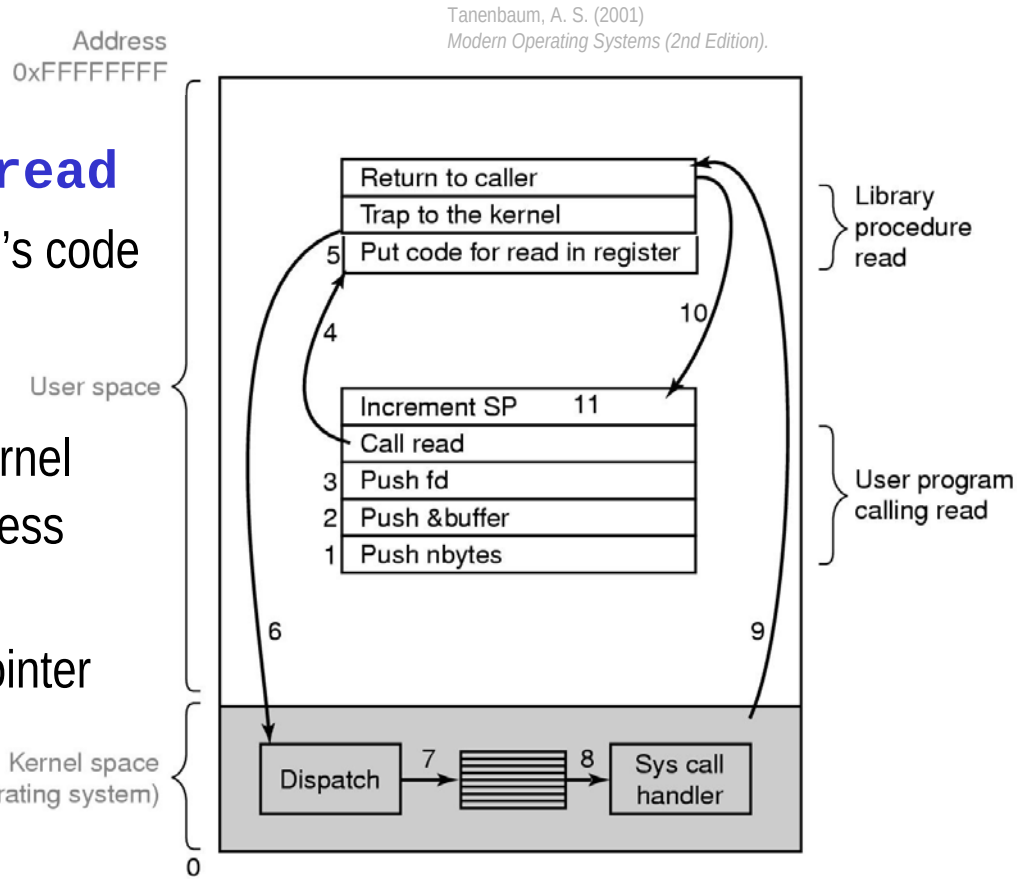
Note: In truth, the system call functions are themselves utility functions wrapping the actual system trap instructions. A system call function invokes the kernel service of the same name by (a) placing the arguments in registers and (b) executing the machine instruction that will generate a software interrupt in the kernel.

1.e System Calls

System calls are the O/S API

- Steps in making a system call

1. – 3. user program pushes parameters onto stack
4. user prog. calls library function **`read`**
5. function **`read`** puts system call's code number **`#read`** in a register
6. function executes **TRAP** instruction: this switches to kernel mode and jumps to fixed address
7. kernel program gets code # and dispatches to call handler's pointer
8. system call handler runs
9. control returns to user space
10. – 11. **`read`** returns to user program, which clears stack



11 steps in making a system call

1.e System Calls

System calls are the O/S API

Note: 3 methods to pass parameters between a process and the O/S

- ✓ pass parameters in registers
- ✓ store parameters in a table in memory, and pass table address as a parameter in a register
- ✓ parameters pushed (stored) onto the stack by the program, and popped off the stack by the O/S

1.e System Calls

System calls are the O/S API

- Main categories of system calls
  - ✓ process creation and management
  - ✓ main-memory management
  - ✓ file access
  - ✓ directory and file-system management
  - ✓ I/O handling
  - ✓ protection
  - ✓ networking
  - ✓ information maintenance (time)

1.e System Calls

Sample of UNIX system calls

- A few common system calls for process control
  - ✓ **`pid = fork()`**
    create a child process identical to the parent
  - ✓ **`err = execve(name, argv, ...)`**
    replace a process's core image
  - ✓ **`pid = waitpid(pid, ...)`**
    wait for a child process to terminate
  - ✓ **`exit(status)`**
    terminate process execution, returning status
  - ✓ **`err = kill(pid, signal)`**
    send a signal to a process

1.e System Calls

Sample of UNIX system calls

- A few common system calls for file access
  - ✓ **`fd = open(file, how, ...)`**
    open a file for reading, writing or both
  - ✓ **`err = close(fd)`**
    close an open file
  - ✓ **`n = read / write(fd, buffer, nbytes)`**
    read (write) data from a file (buffer) into a buffer (file)
  - ✓ **`err = stat(name, &buf)`**
    get a file's status information
  - ✓ **`err = chmod(name, mode)`**
    change a file's protection bits

1.e System Calls

Sample of UNIX system calls

- A few common system calls for directory management
  - ✓ **`err = mkdir(name, mode)`**
    create a new directory
  - ✓ **`err = rmdir(name)`**
    remove an empty directory
  - ✓ **`err = chdir(dirname)`**
    change the working directory
  - ✓ **`err = link(name1, name2)`**
    create a new entry, name2, pointing to name1
  - ✓ **`err = mount(name, path, how)`**
    mount a file system

1.e System Calls

Sample of UNIX system calls

Note: The detailed effect of these commands will become clearer in subsequent chapters. For now, you can find the complete documentation of all UNIX calls, library routines and commands in the man(ual) pages, by typing:

`> man` *command*

1.e System Calls

Equivalent Windows system calls

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Tanenbaum, A. S. (2001) *Modern Operating Systems (2nd Edition).*

The Win32 API calls roughly correspond to the UNIX calls

1.e System Calls

System programs

- System programs are tools that help other programs
  - ✓ not strictly part of the O/S, but provide a convenient environment for program development and execution ("utilities")
  - ✓ they range from simple library functions to full-fledged editors
  - ✓ most users' view of the operating system is defined by the system programs, not the actual low-level system calls
- Most important system programs
  - ✓ command interpreters (shells) – programs that launch programs
  - ✓ compilers, assemblers, linkers, loaders, debuggers – programs that compile programs
  - ✓ text editors – programs that develop programs

1.e System Calls

System programs

- Command-Line Interpreter (CLI) shells

```
login as: doursat
doursat@banyan.cse.unr.edu's password:
Last login: Tue Aug 30 18:06:13 2005 from loa2.cse.unr.ed
 ------------------------------------------------------------------------------
            Department of Computer Science and Engineering at UNR
 ------------------------------------------------------------------------------
          For extended message of the day: http://info.cse.unr.edu/
                   For help: http://www.cse.unr.edu/~help
 ------------------------------------------------------------------------------
                      Send problems to root@cse.unr.edu
 ------------------------------------------------------------------------------
           Please ssh to remote1-remote3 for software development
 ------------------------------------------------------------------------------
You have mail.
doursat@banyan ~ > ls
Desktop            old_login_scripts  public_html        Recherche
doursat@banyan ~ > ls -l
total 8
drwxr-xr-x   2 doursat  staff        512 Aug 26 14:09 Desktop
drwx------   2 doursat  staff        512 Jul  7 14:06 old_login_scripts
drwx-----x   4 doursat  staff       1024 Aug 28 17:54 public_html
drwx------   7 doursat  staff        512 Aug 22 14:29 Recherche
doursat@banyan ~ > ls -l public_html
```

A Bourne-Again Shell (bash) session on banyan.cse.unr.edu

1.e System Calls

System programs

- Graphical User Interface (GUI) shells

One of the many X Window graphical user interfaces available for Linux

1.e System Calls

System programs

- The “shell” is the outer envelope of the O/S
  - ✓ the shell is a program that presents a primary interface between the user and the O/S
  - ✓ most often runs as a special user program when user logs in
  - ✓ Command-Line Interface (CLI) shells
    - the user types commands that are interpreted by the shell as system calls, library functions or other programs
  - ✓ Graphical User Interface (GUI) shells
    - offer a metaphor of direct manipulation of graphical images and widgets in addition to text
    - more user-friendly but less control
    - often wrapped around a CLI core

Principles of Operating Systems

CS 446/646

1. Introduction to Operating Systems

a. Role of an O/S

b. O/S History and Features

c. Types of O/S

d. Major O/S Components

e. System Calls

- ✓ System calls are the O/S API
- ✓ Sample of UNIX system calls
- ✓ Equivalent Windows system calls
- ✓ System programs

f. O/S Software Architecture

g. Examples of O/S

Principles of Operating Systems

CS 446/646

1. Introduction to Operating Systems

a. Role of an O/S

b. O/S History and Features

c. Types of O/S

d. Major O/S Components

e. System Calls

f. O/S Software Architecture

- ✓ Why software architecture?
- ✓ Monolithic structure
- ✓ Layered structure
- ✓ Microkernel & modular structure
- ✓ Virtual machines

g. Examples of O/S

1.f Operating System Software Architecture

Why software architecture?

After our view of an O/S from the outside (the system calls interface), we take a look inside and examine the different O/S software structures that have been tried.

Note: Good software architecture principles apply to any software, not just operating systems.

1.f Operating System Software Architecture

Why software architecture?

- Operating systems have become huge complex beasts

Year	System	Code Size
1963	CTSS	32,000 words
1964	OS/360	1 million instructions
1975	Multics	20 million instructions
1990	Windows 3.1	3 million SLOC
2000	Windows NT 4.0	16 million SLOC
2002	Windows XP	40 million SLOC
2000	Red Hat Linux 6.2	17 million SLOC
2001	Red Hat Linux 7.1	30 million SLOC
2002	Debian 3.0	104 million SLOC
2005	Debian 3.1	213 million SLOC

1.f Operating System Software Architecture

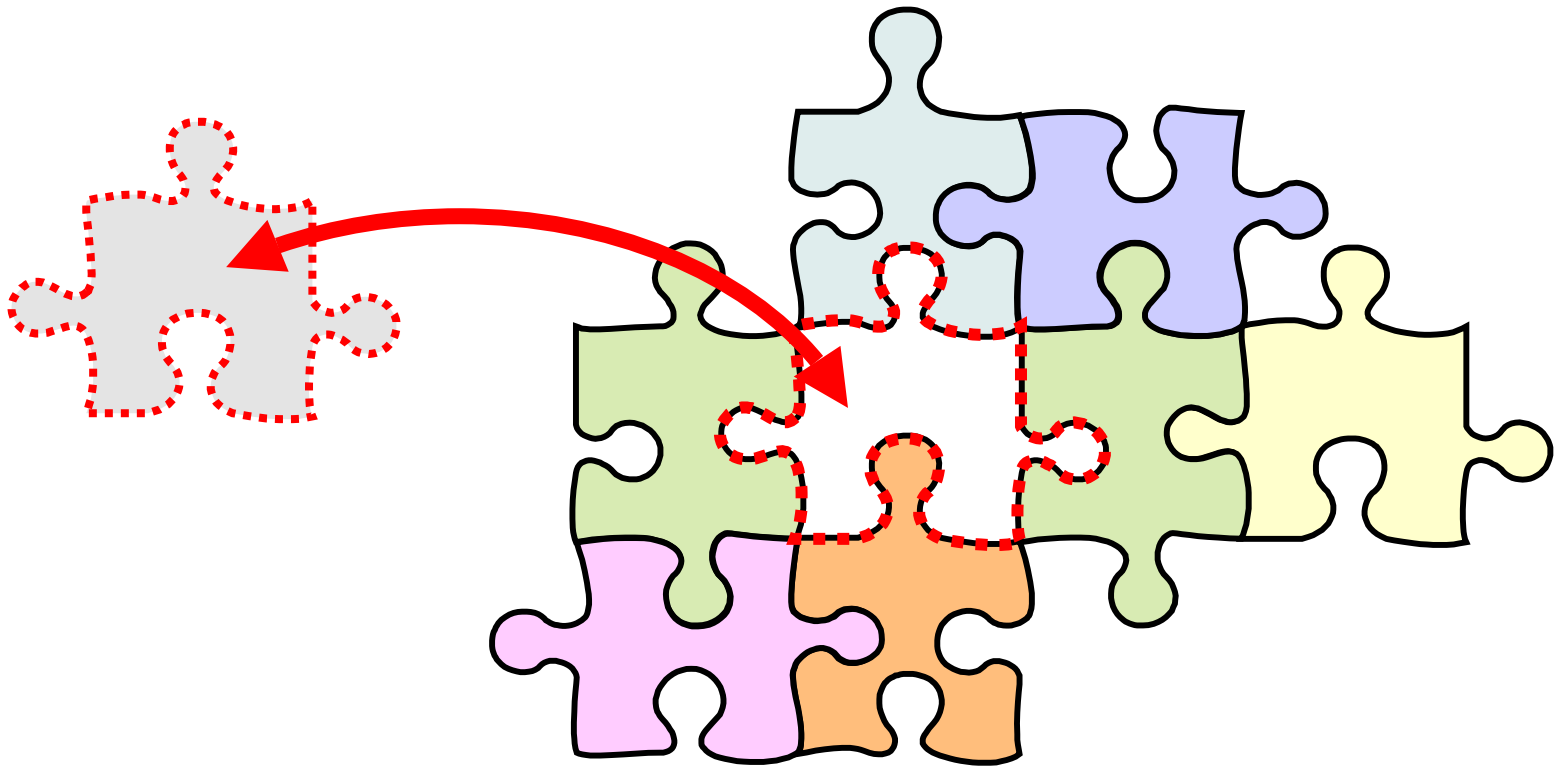
Why software architecture?

- With code size come all the problems
  - ✓ O/S are chronically late in being delivered (new or upgrades)
  - ✓ O/S have latent bugs that show up and must be fixed
  - ✓ performance is often not what was expected
  - ✓ it has proven impossible to deploy an O/S that is not vulnerable to security attacks
- Hence the critical need for a well-engineered software architecture
  - ✓ **layers** and/or **modules** with clean, minimal **interfaces**
  - ✓ the goal is that one part can be changed (fixed, upgraded, expanded) without impacting the other parts

1.f Operating System Software Architecture

Why software architecture?

- Well-defined interfaces allow part replacement without impacting the surroundings



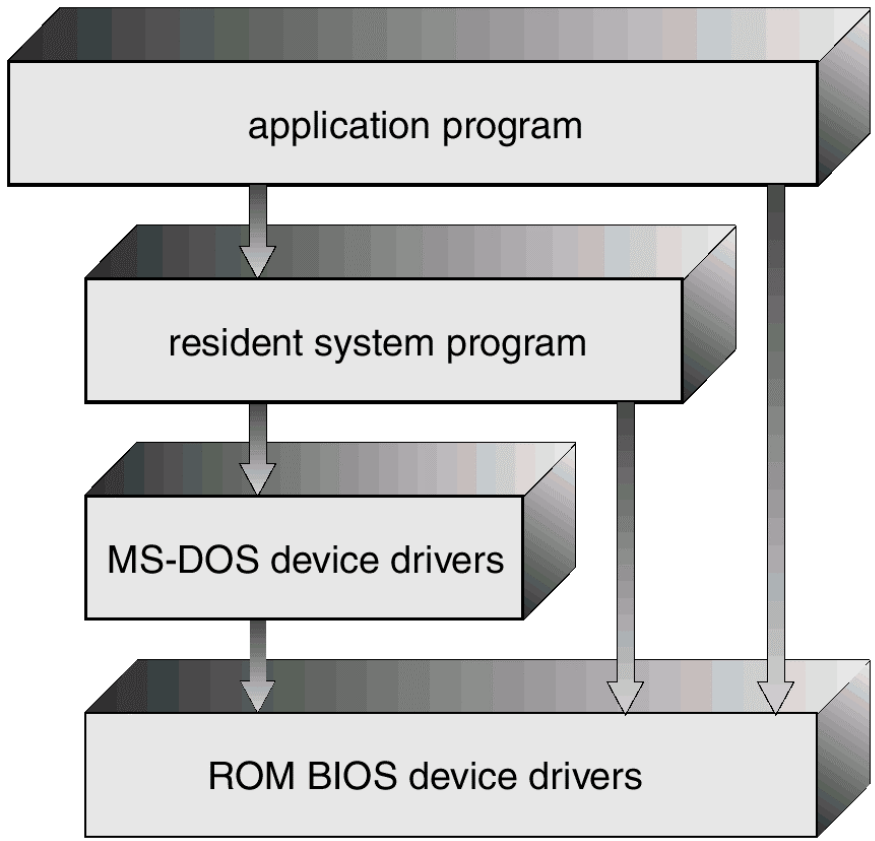
1.f Operating System Software Architecture

Monolithic structure

- A famous bad example: MS-DOS

Silberschatz, A., Galvin, P. B. and Gagne. G. (2003) *Operating Systems Concepts with Java (6th Edition).*

- ✓ initially written to provide the most functionality in the least space
- ✓ started small and grew beyond its original scope
- ✓ levels not well separated: programs could access I/O devices directly
- ✓ excuse: the hardware of that time was limited (no dual user/kernel mode)

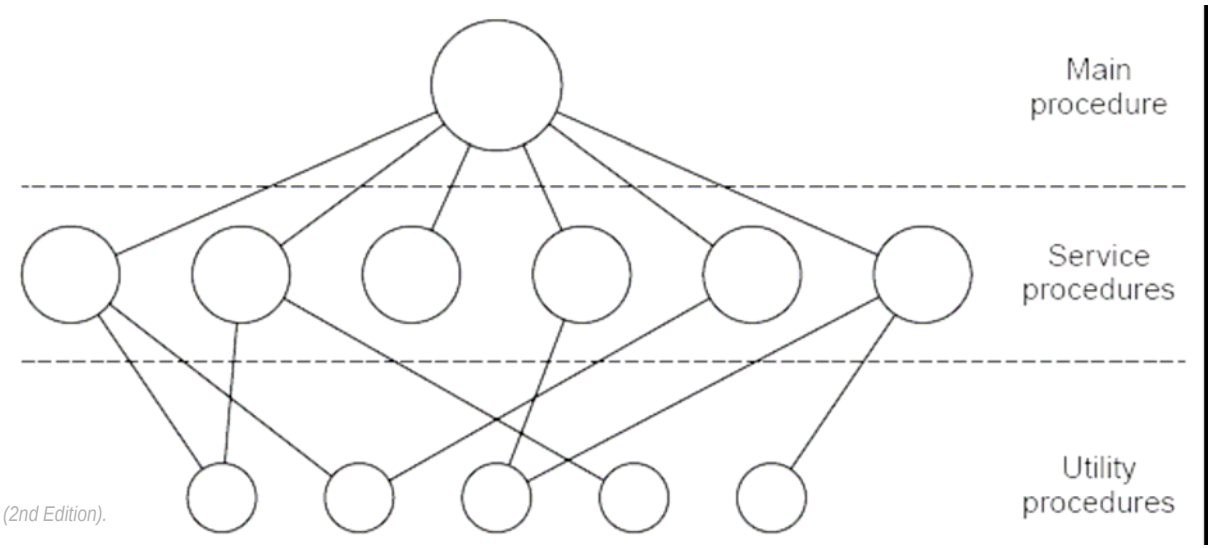


MS-DOS pseudolayer structure

1.f Operating System Software Architecture

Monolithic structure

- Another bad example: the original UNIX
  - ✓ "The Big Mess": a collection of procedures that can call any of the other procedures whenever they need to
  - ✓ no encapsulation, total visibility across the system
  - ✓ very minimal layering made of thick, monolithic layers



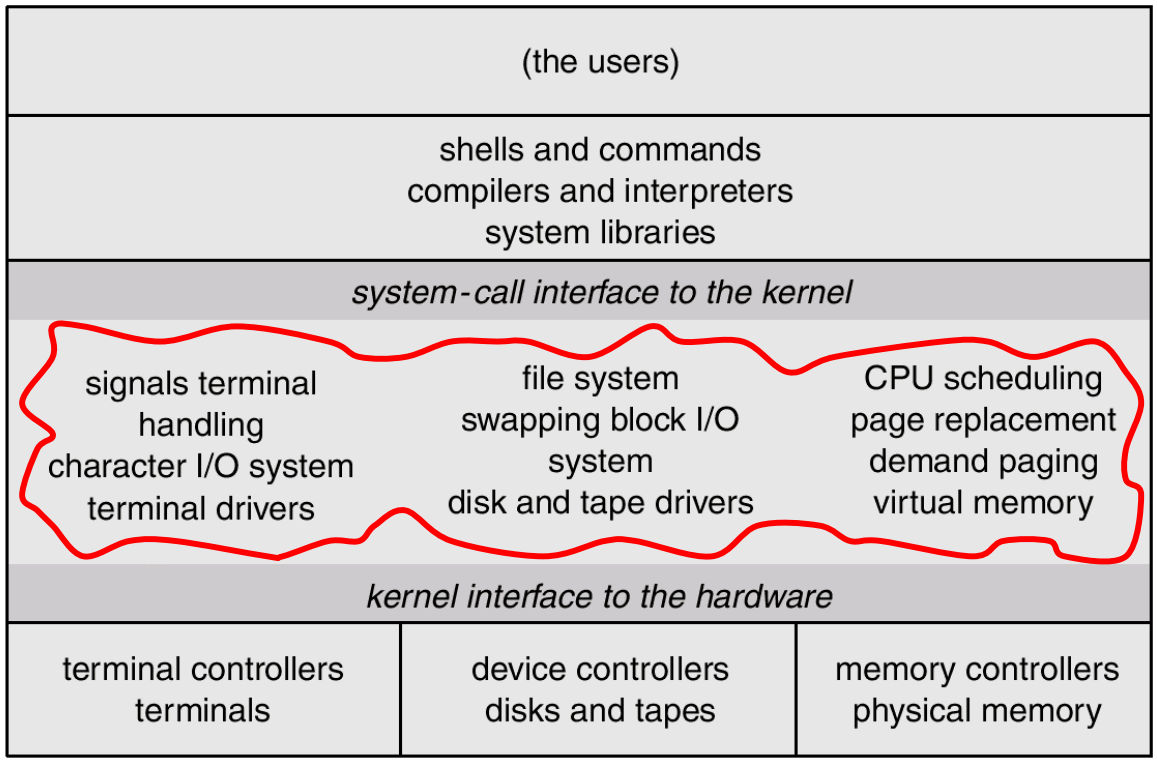
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

A simple structuring model for a monolithic system

1.f Operating System Software Architecture

Monolithic structure

- Another bad example: the original UNIX
  - ✓ enormous amount of functionality crammed into the kernel

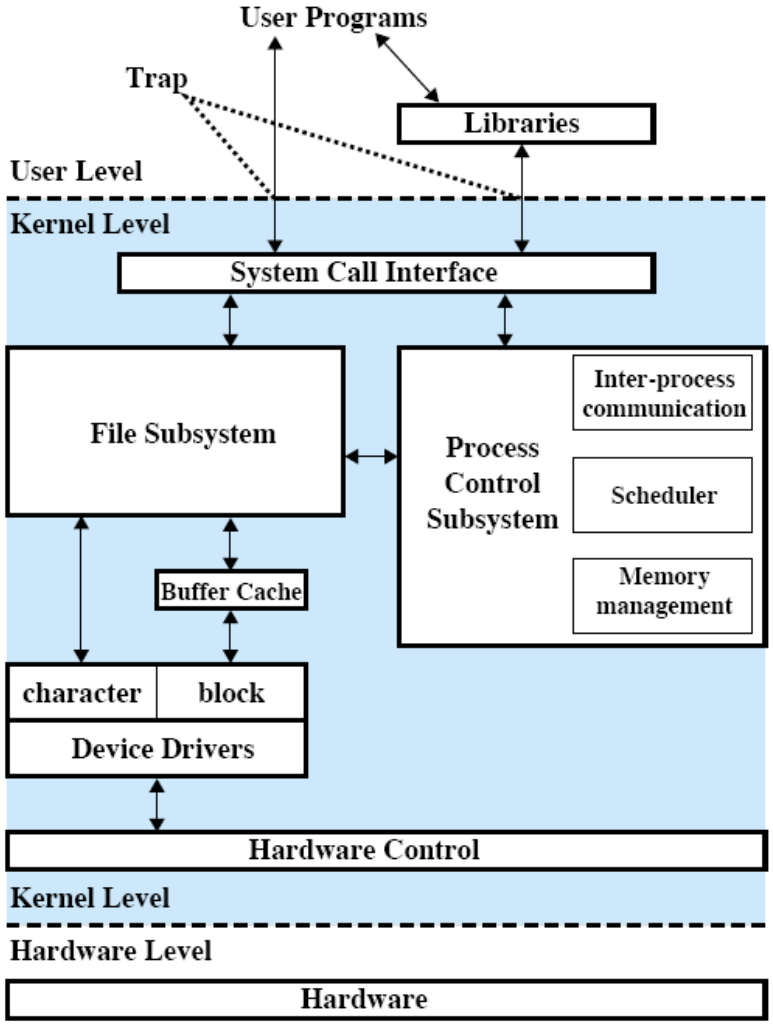


UNIX system structure

Silberschatz, A., Galvin, P. B. and Gagne. G. (2003) *Operating Systems Concepts with Java (6th Edition).*

1.f Operating System Software Architecture

Monolithic structure



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

The traditional UNIX kernel contains few “layers”

1.f Operating System Software Architecture

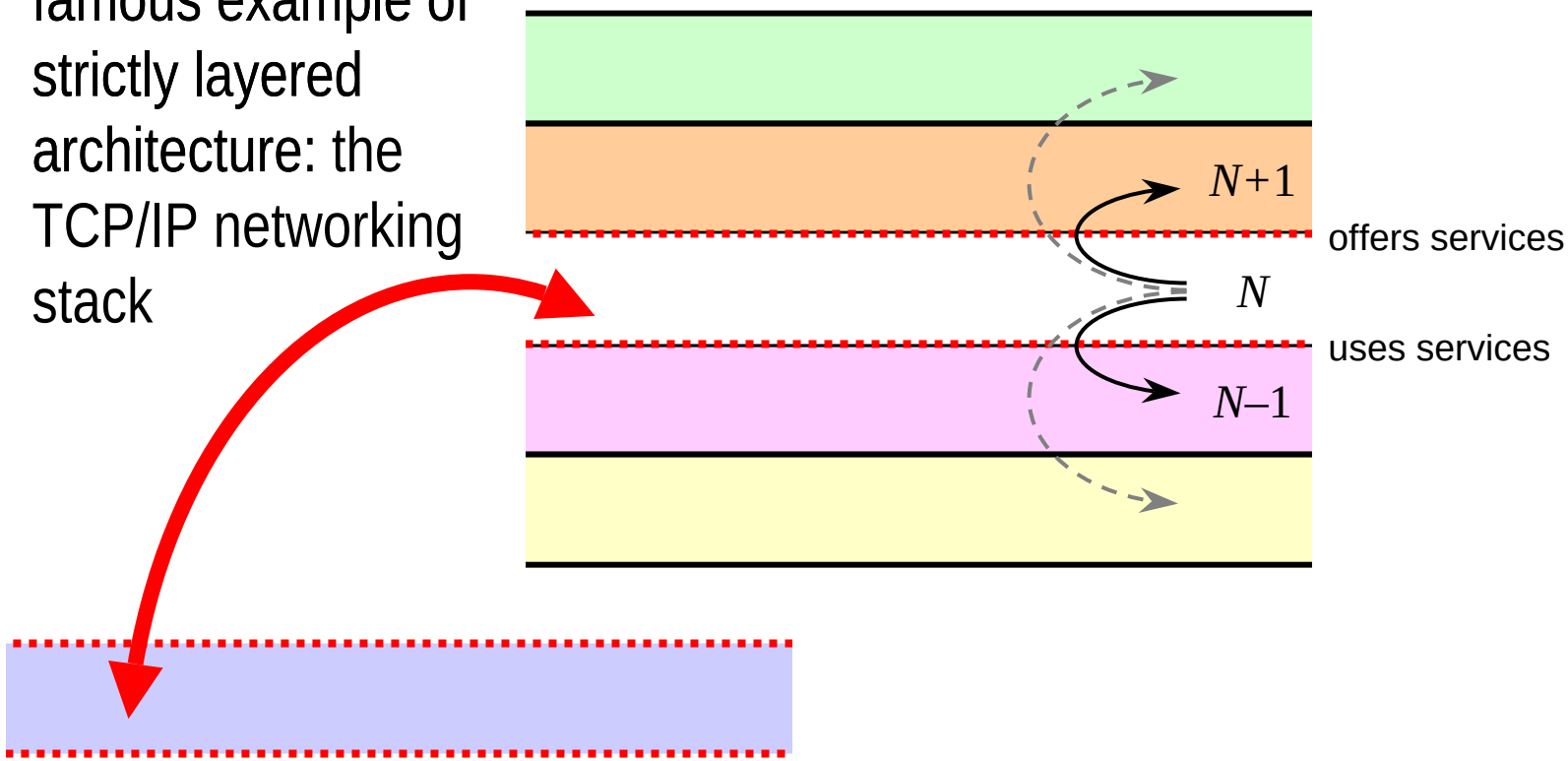
Layered structure

- Monolithic operating systems
  - ✓ no one had experience in building truly large software systems
  - ✓ the problems caused by mutual dependence and interaction were grossly underestimated
  - ✓ such lack of structure became unsustainable as O/S grew
- Enter hierarchical layers and information abstraction
  - ✓ each layer is implemented exclusively using operations provided by lower layers
  - ✓ it does not need to know how they are implemented
  - ✓ hence, lower layers hide the existence of certain data structures, private operations and hardware from upper layers

1.f Operating System Software Architecture

Layered structure

- Layers can be debugged and replaced independently without bothering the other layers above and below
  - ✓ famous example of strictly layered architecture: the TCP/IP networking stack



1.f Operating System Software Architecture

Layered structure

shell

O/S

hardware

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

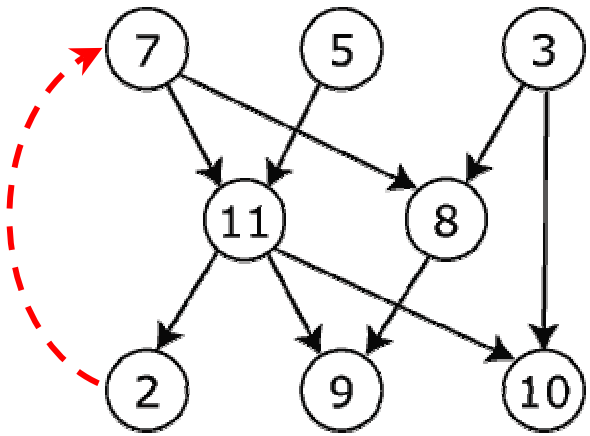
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Theoretical model of operating system design hierarchy

1.f Operating System Software Architecture

Layered structure

- Major difficulty with layering
  - ✓ . . . appropriately defining the various layers!
  - ✓ layering is only possible if all function dependencies can be sorted out into a Directed Acyclic Graph (DAG)
  - ✓ however there might be conflicts in the form of circular dependencies (“cycles”)



Circular dependency on top of a DAG

1.f Operating System Software Architecture

Layered structure

- Circular dependencies in an O/S organization
  - ✓ example: disk driver routines vs. CPU scheduler routines
    - the device driver for the backing store (disk space used by virtual memory) may need to wait for I/O, thus invoke the CPU-scheduling layer
    - the CPU scheduler may need the backing store driver for swapping in and out parts of the table of active processes
- Other difficulty: efficiency
  - ✓ the more layers, the more indirections from function to function and the bigger the overhead in function calls
  - → backlash against strict layering: return to fewer layers with more functionality

1.f Operating System Software Architecture

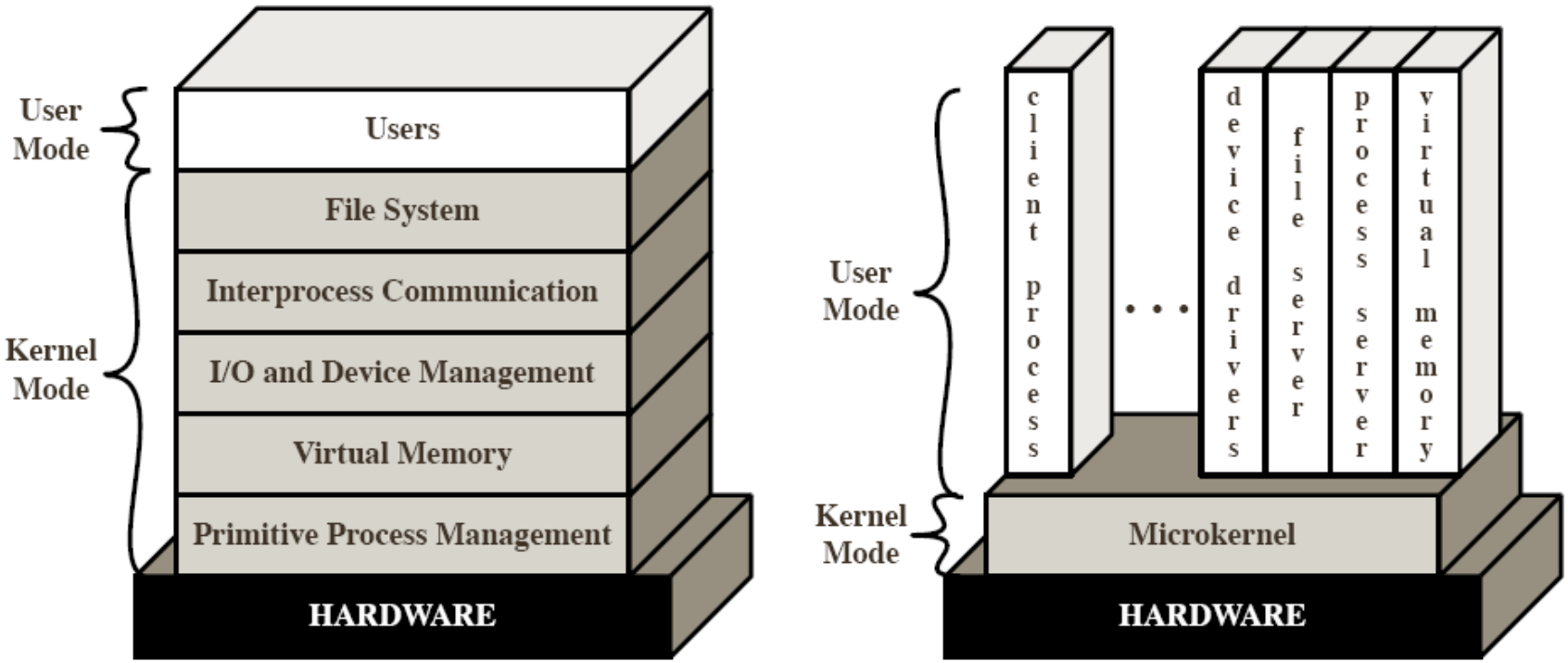
Microkernel & modular structure

- The microkernel approach
  - ✓ a microkernel is a reduced operating system core that contains only essential O/S functions
  - ✓ the idea is to “defat” the kernel by moving up as much functionality as possible from the kernel into user space
  - ✓ many services traditionally included in the O/S are now external subsystems running as user processes
    - device drivers
    - file systems
    - virtual memory manager
    - windowing system
    - security services, etc.

1.f Operating System Software Architecture

Microkernel & modular structure

- The microkernel approach



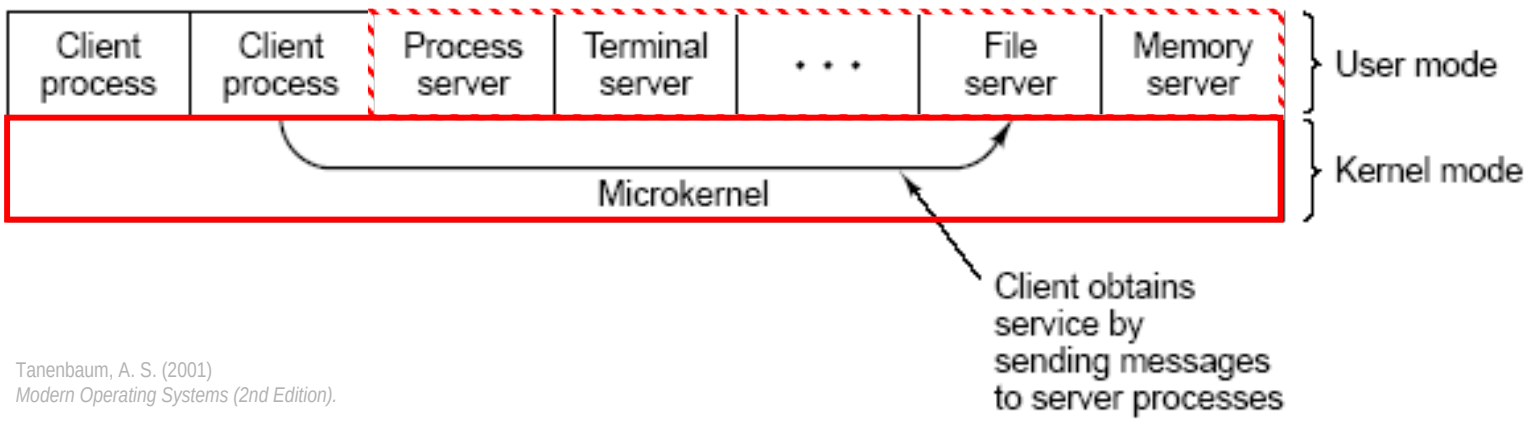
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Layered O/S architecture vs. microkernel architecture

1.f Operating System Software Architecture

Microkernel & modular structure

- User modules then communicate by message passing
  - ✓ the main function of the microkernel is to provide an interprocess communication facility between the client programs and the various services running in user space
  - ✓ the microkernel offers a uniform message-passing interface
  - ✓ MINIX (not Linux), Mach (NEXTSTEP, Mac OS X), QNX (embedded)



Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

The client-server microkernel model

1.f Operating System Software Architecture

Microkernel & modular structure

- Benefits of the microkernel approach
  - ✓ **extensibility** — it is easier to extend a microkernel-based O/S as new services are added in user space, not in the kernel
  - ✓ **portability** — it is easier to port to a new CPU, as changes are needed only in the microkernel, not in the other services
  - ✓ **reliability & security** — much less code is running in kernel mode; failures in user-space services don't affect kernel space
- Detriments of the microkernel approach
  - ✓ again, performance overhead due to communication from user space to kernel space
  - ✓ not always realistic: some functions (I/O) must remain in kernel space, forcing a separation between "policy" and "mechanism"

1.f Operating System Software Architecture

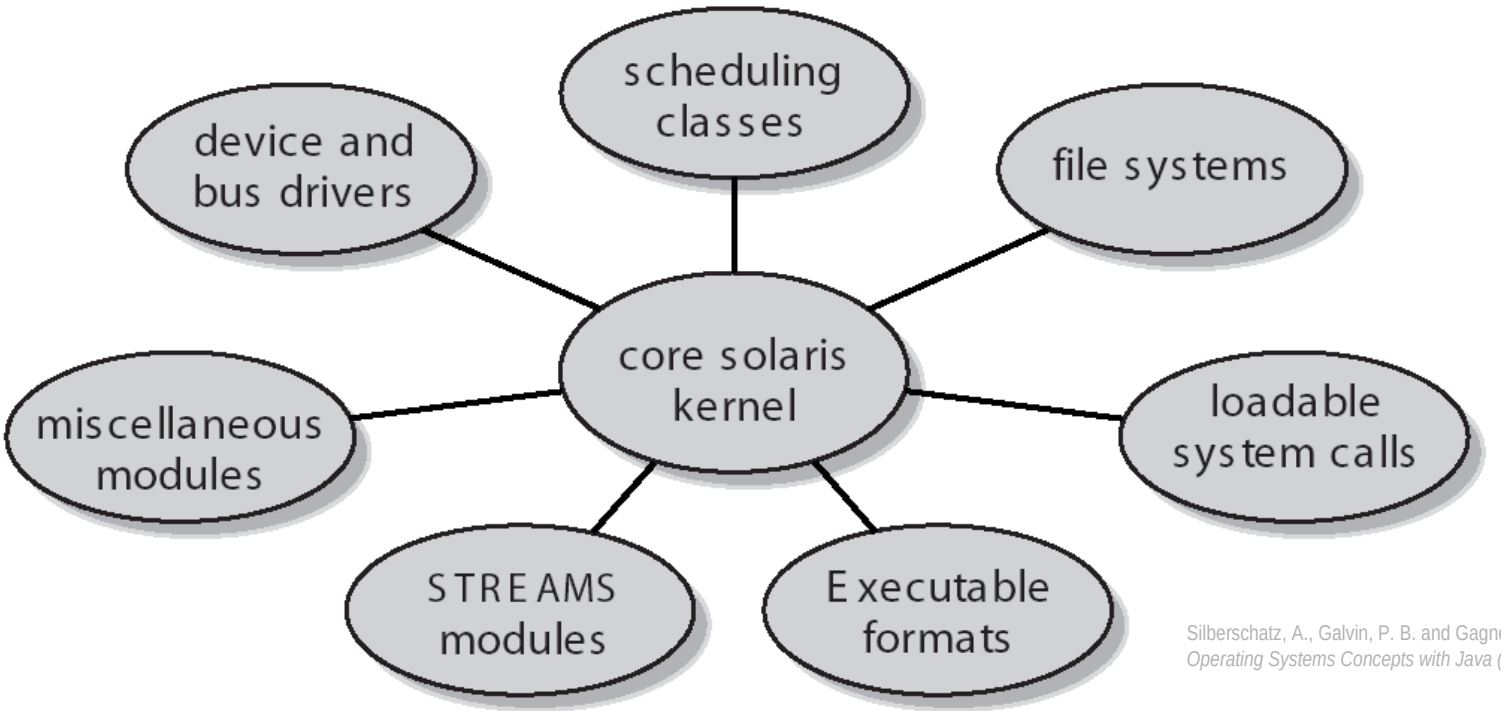
Microkernel & modular structure

- The modular approach
  - ✓ most modern operating systems implement kernel **modules**
  - ✓ this is similar to the object-oriented approach:
    - each core component is separate
    - each talks to the others over known interfaces
    - each is loadable as needed within the kernel
  - ✓ overall, modules are similar to layers but with more flexibility
  - ✓ modules are also similar to the microkernel approach, except they are inside the kernel and don't need message passing

1.f Operating System Software Architecture

Microkernel & modular structure

- Modules are used in Solaris, Linux and Mac OS X



Silberschatz, A., Galvin, P. B. and Gagne. G. (2003) *Operating Systems Concepts with Java (6th Edition).*

The Solaris loadable modules

1.f Operating System Software Architecture

Microkernel & modular structure

Note: Software development is not an exact science but a trial-and-error exploratory process. Ironically, perfectly regular machines rely upon intuitive human instructions. **Design patterns** are an attempt to bridge this gap by factoring out common practice. They provide guidelines to discipline the code, harness its complexity, and bring it closer to the ideals of modularity, scalability, flexibility and robustness.

1.f Operating System Software Architecture

Virtual machines

- A virtual machine provides an interface identical to the underlying bare hardware
  - ✓ a VM takes the layered approach to its logical conclusion: it treats hardware and the operating system kernel as though they were all hardware
  - ✓ a VM-based O/S creates the illusion that processes are each executing on their own private processor with their own private (virtual) memory
  - → a VM-based O/S does *not* provide a traditional O/S extension of the underlying hardware but an exact copy of the bare hardware, so that any traditional O/S can run on top of it
  - ✓ it can also provide a simulation of a 3rd party hardware

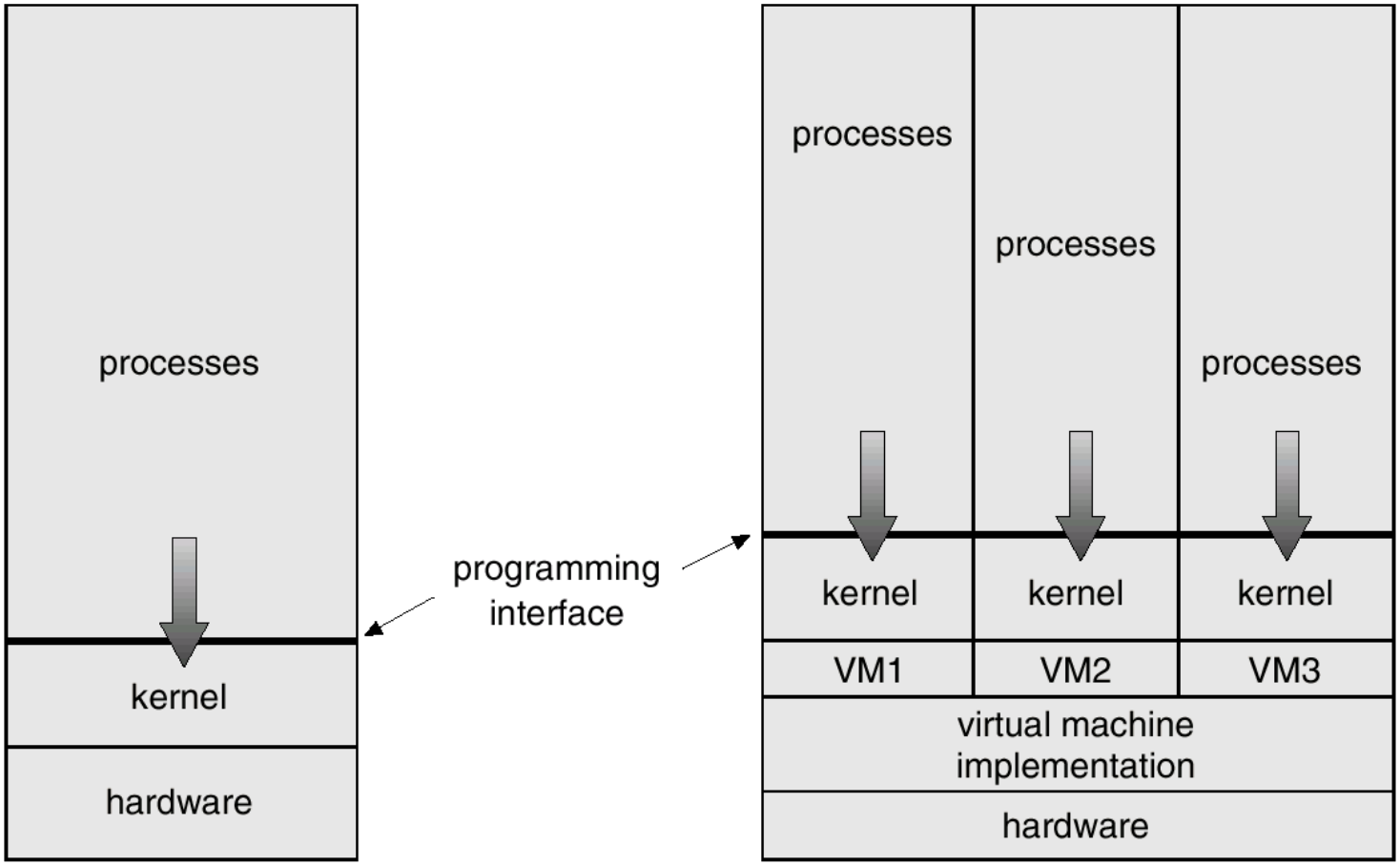
1.f Operating System Software Architecture

Virtual machines

- The resources of the physical computer are shared to create the virtual machines
  - ✓ CPU scheduling can create the appearance that users have their own processor
  - ✓ spooling and a file system can provide virtual card readers and virtual line printers
  - ✓ a normal user time-sharing terminal serves as the virtual machine operator's console
  - ✓ the VM can create more virtual disks ("minidisks") than there are physical drives

1.f Operating System Software Architecture

Virtual machines



Silberschatz, A., Galvin, P. B. and Gagne. G. (2003)
Operating Systems Concepts with Java (6th Edition).

Nonvirtual machine vs. virtual machines

1.f Operating System Software Architecture

Virtual machines

- Advantages of virtual machines
  - ✓ the VM concept provides complete protection of system resources since each virtual machine is isolated from all others
  - ✓ O/S research and development can be done on a VM without disrupting the normal system operation
  - ✓ cross-platform portability and emulation; ex: a Motorola 68000 VM on top of a PowerPC, or an Intel VM on top of a SPARC
- Disadvantages of virtual machines
  - ✓ the VM isolation permits no direct sharing of resources
  - ✓ VMs are difficult to implement due to the effort required to provide an exact duplicate to the underlying machine (four modes: physical kernel / user = { virtual kernel / user }, etc.)

Principles of Operating Systems

CS 446/646

1. Introduction to Operating Systems

a. Role of an O/S

b. O/S History and Features

c. Types of O/S

d. Major O/S Components

e. System Calls

f. O/S Software Architecture

- ✓ Why software architecture?
- ✓ Monolithic structure
- ✓ Layered structure
- ✓ Microkernel & modular structure
- ✓ Virtual machines

g. Examples of O/S

Principles of Operating Systems

CS 446/646

1. Introduction to Operating Systems

a. Role of an O/S

b. O/S History and Features

c. Types of O/S

d. Major O/S Components

e. System Calls

f. O/S Software Architecture

g. Examples of O/S

Principles of Operating Systems

CS 446/646

1. Introduction to Operating Systems

a. Role of an O/S

b. O/S History and Features

c. Types of O/S

d. Major O/S Components

e. System Calls

f. O/S Software Architecture

g. Examples of O/S

Principles of Operating Systems

CS 446/646

1. Introduction to Operating Systems

a. Role of an O/S

b. O/S History and Features

c. Types of O/S

d. Major O/S Components

e. System Calls

f. O/S Software Architecture

g. Examples of O/S

Principles of Operating Systems

CS 446/646

0. Course Presentation
1. Introduction to Operating Systems
2. **Processes**
3. **Memory Management**
4. **CPU Scheduling**
5. **Input/Output**
6. **File System**
7. **Case Studies**