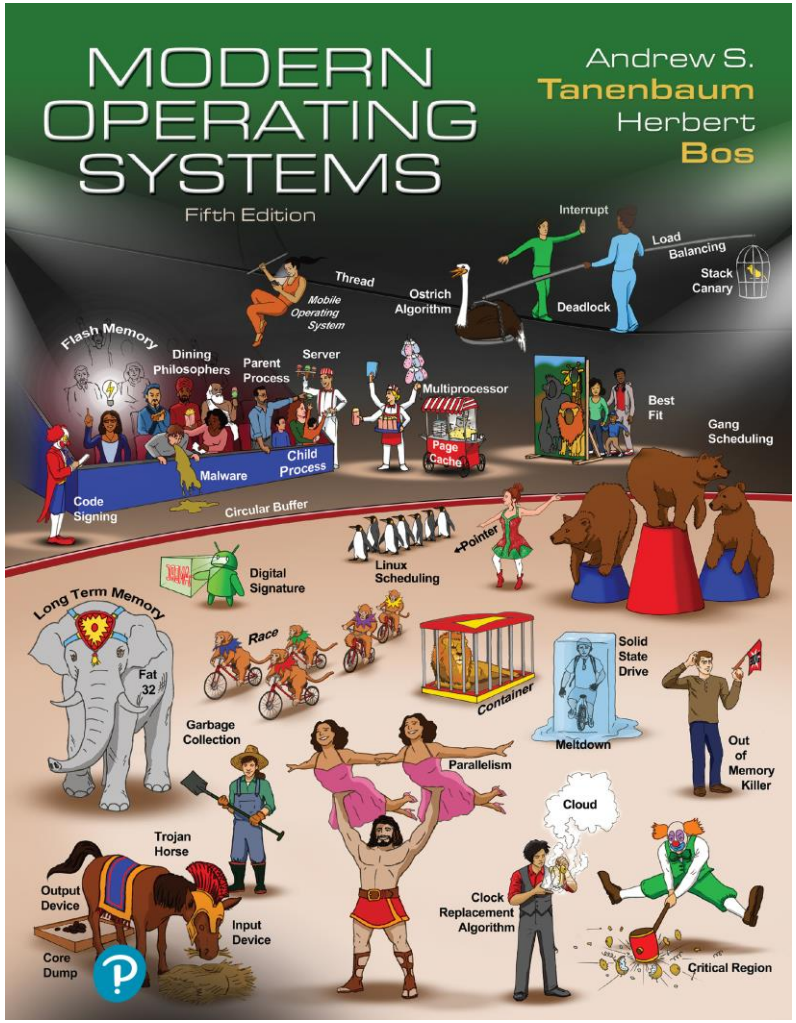
Modern Operating Systems

Fifth Edition

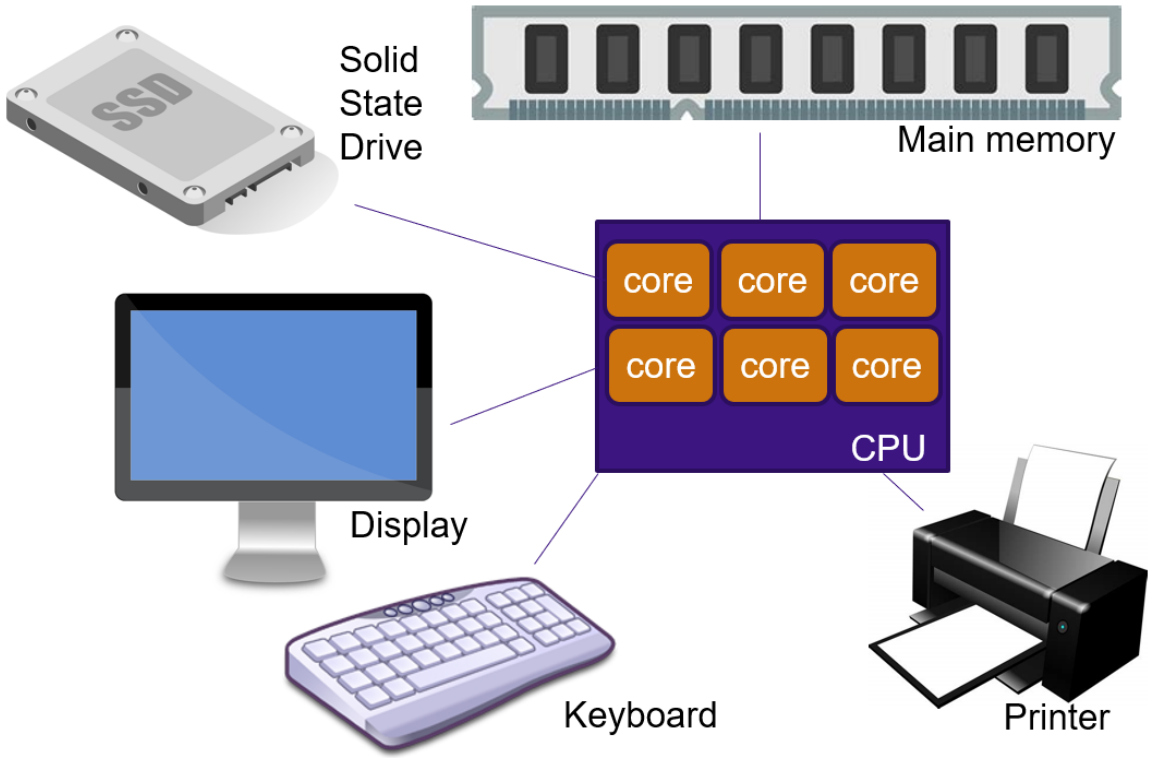


Chapter 1

Introduction

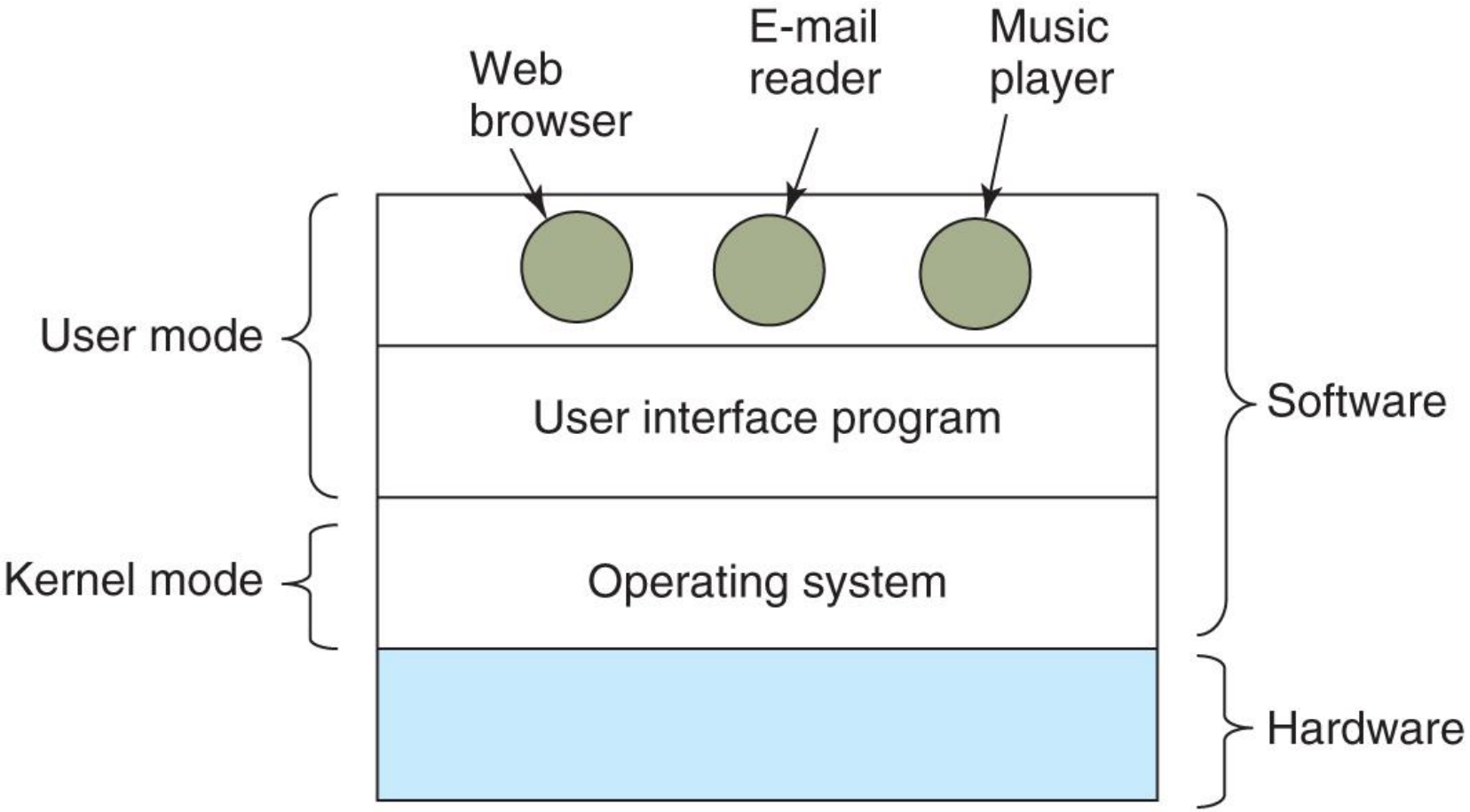
Components of a Modern Computer

- One or more processors
- Main memory
- Disks or Flash drives
- Printers
- Keyboard
- Mouse
- Display
- Network interfaces
- I/O devices



Components of a Modern Computer

Where the operating system fits in.



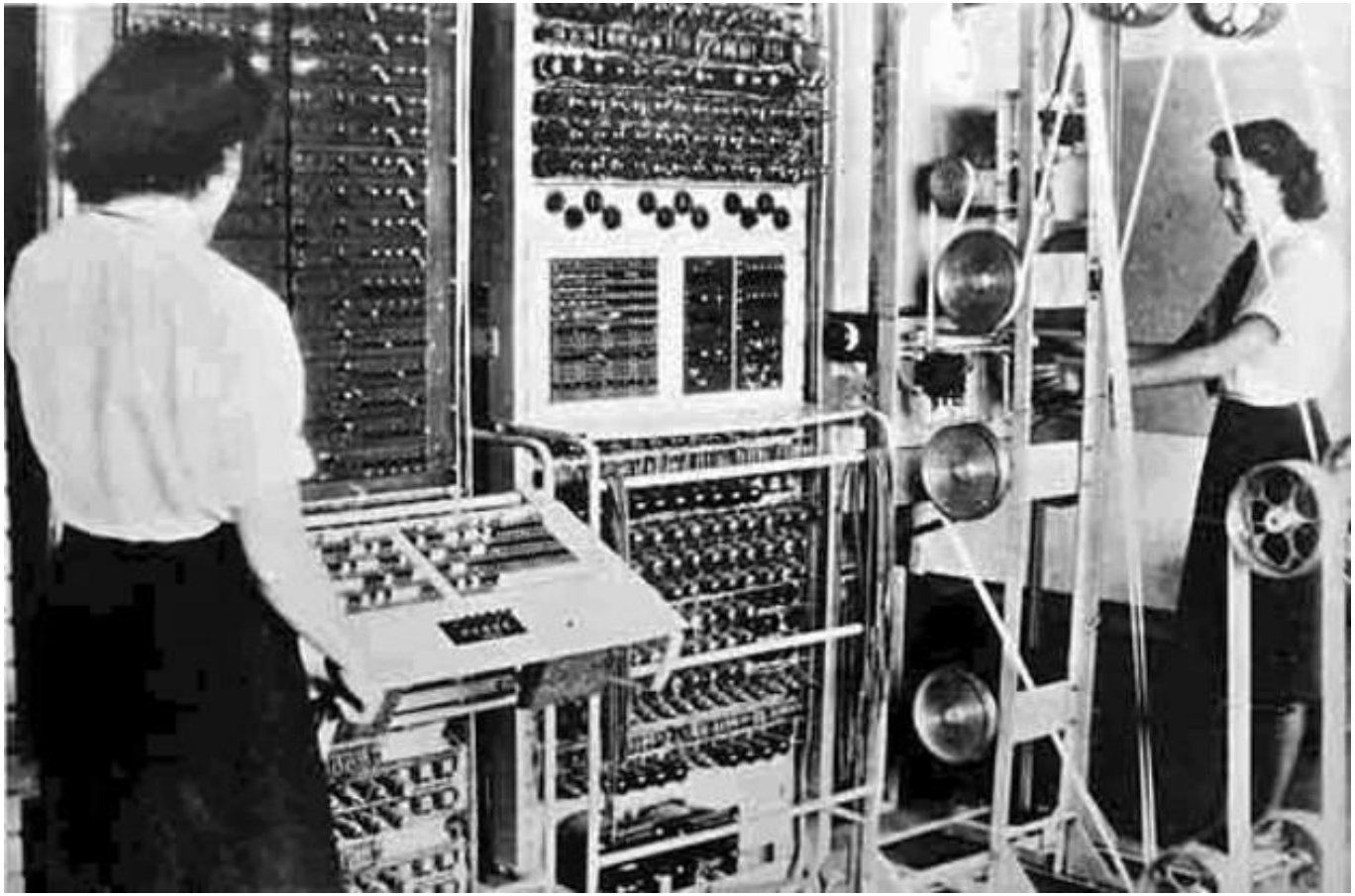
The Operating System as a Resource Manager

- Top-down view
  - Provide abstractions to application programs
- Bottom-up view
  - Manage pieces of complex system
  - Provide orderly, controlled allocation of resources

History of Operating Systems

- The first generation (1945-55): Vacuum tubes
- The second generation (1955-65): Transistors and batch systems
- The third generation (1965-1980): ICs and multiprogramming
- The fourth generation (1980-present): Personal computers
- The fifth generation (1990-present): Mobile computers

First Generation: Vacuum tubes

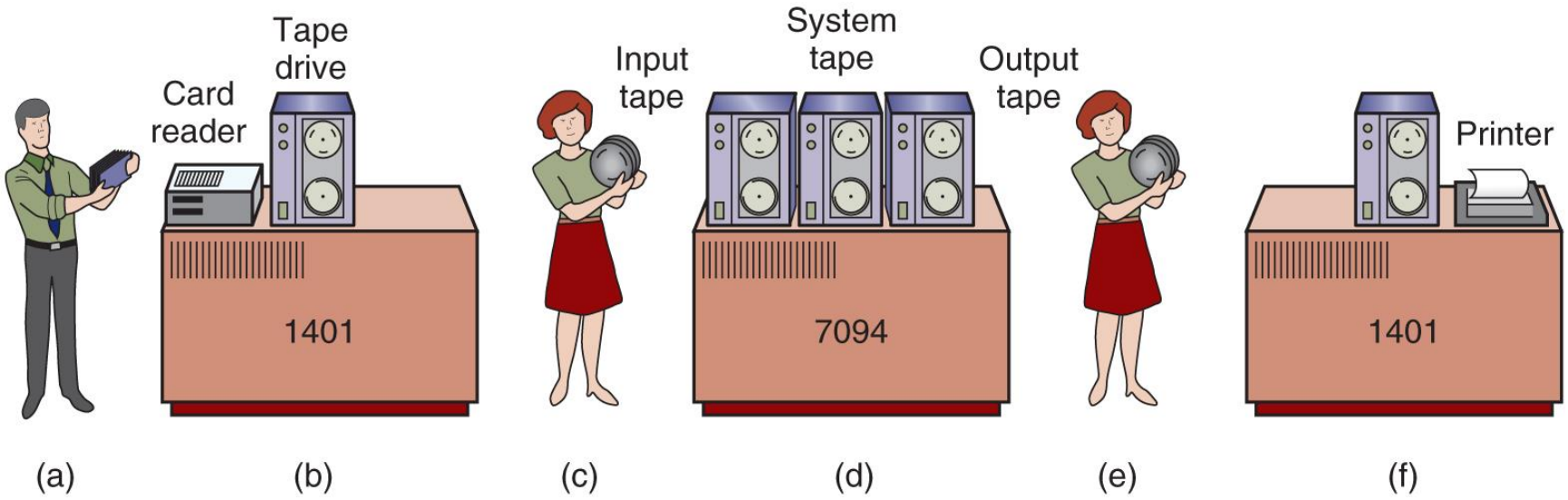


Annotated photographs of the COLOSSUS electronic digital computer/Kew/National Archives and Records Administration (NARA) [FO 850/234]

The Colossus Mark 2 computer: Cryptanalysis of the Lorenz Cipher

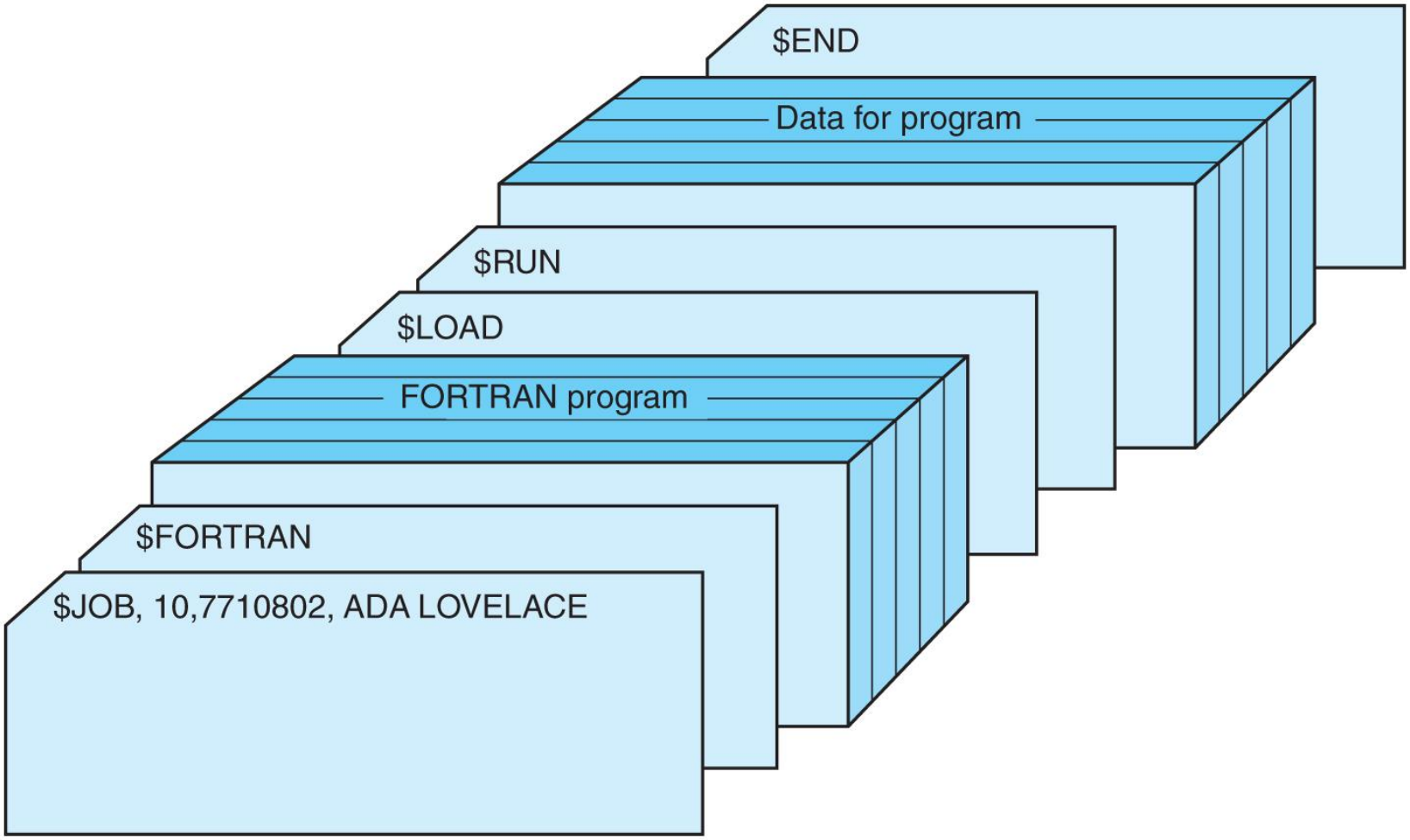
Second Generation: Transistors and Batch Systems

An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.



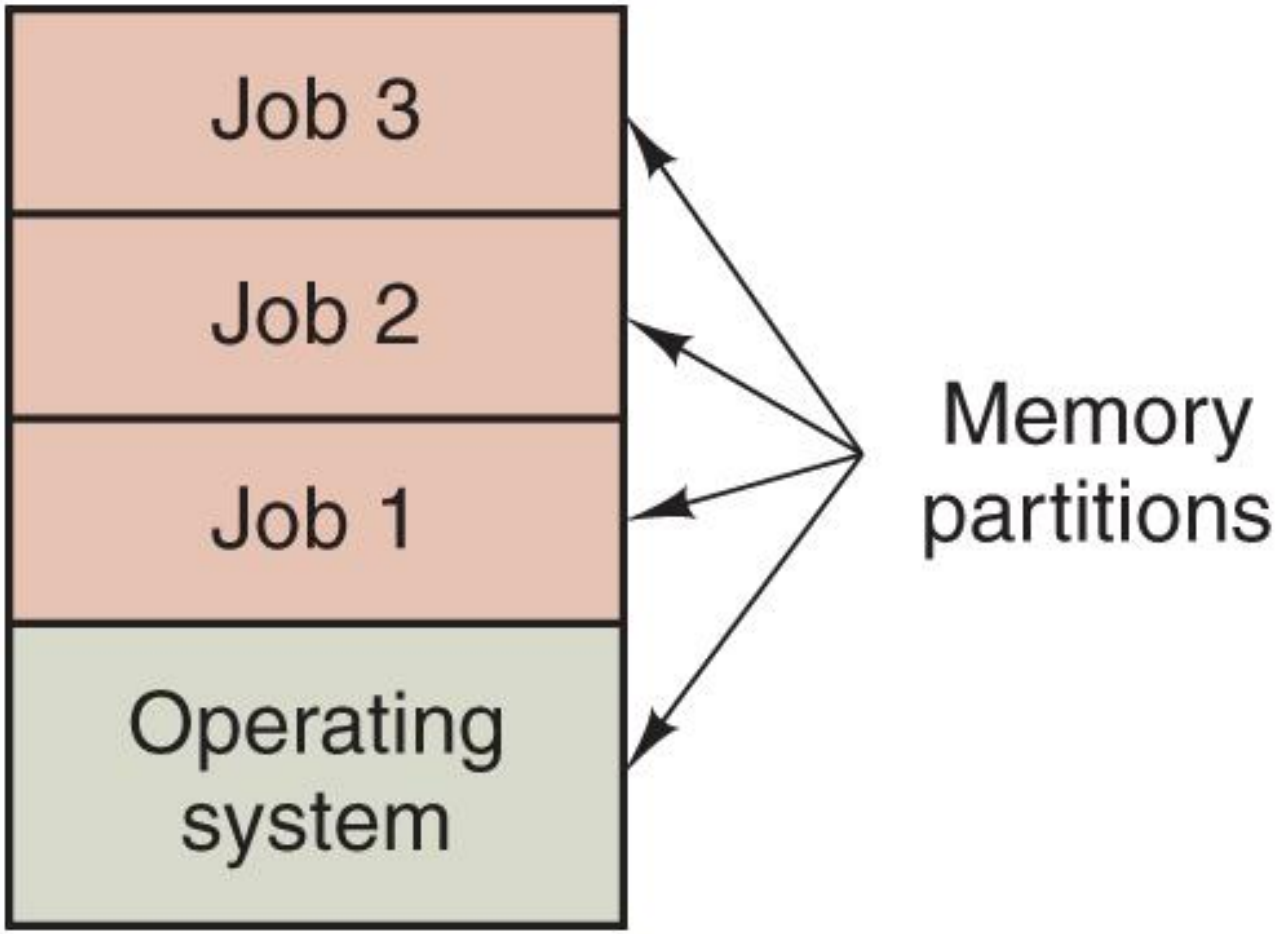
Second Generation: Transistors and Batch Systems

Structure of a typical FMS job.

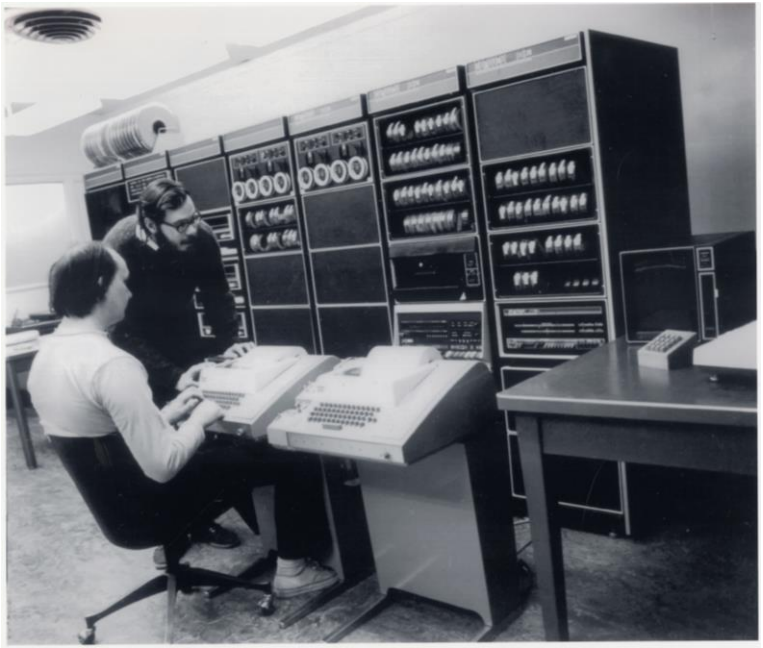


Third Generation: ICs and Multiprogramming

A multiprogramming system with three jobs in memory.



UNIX – a Simpler Operating System



Courtesy of the Computer History Museum

```
/*
 * You are not expected to understand this.
 */
```

Comment by Ken Thompson in the source code of UNIXv6, 1975. A.k.a the most famous comment ever. Interestingly, the code to which it pertained contained a bug, so perhaps the authors did not understand it either. :)

MINIX (1980s)

“Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

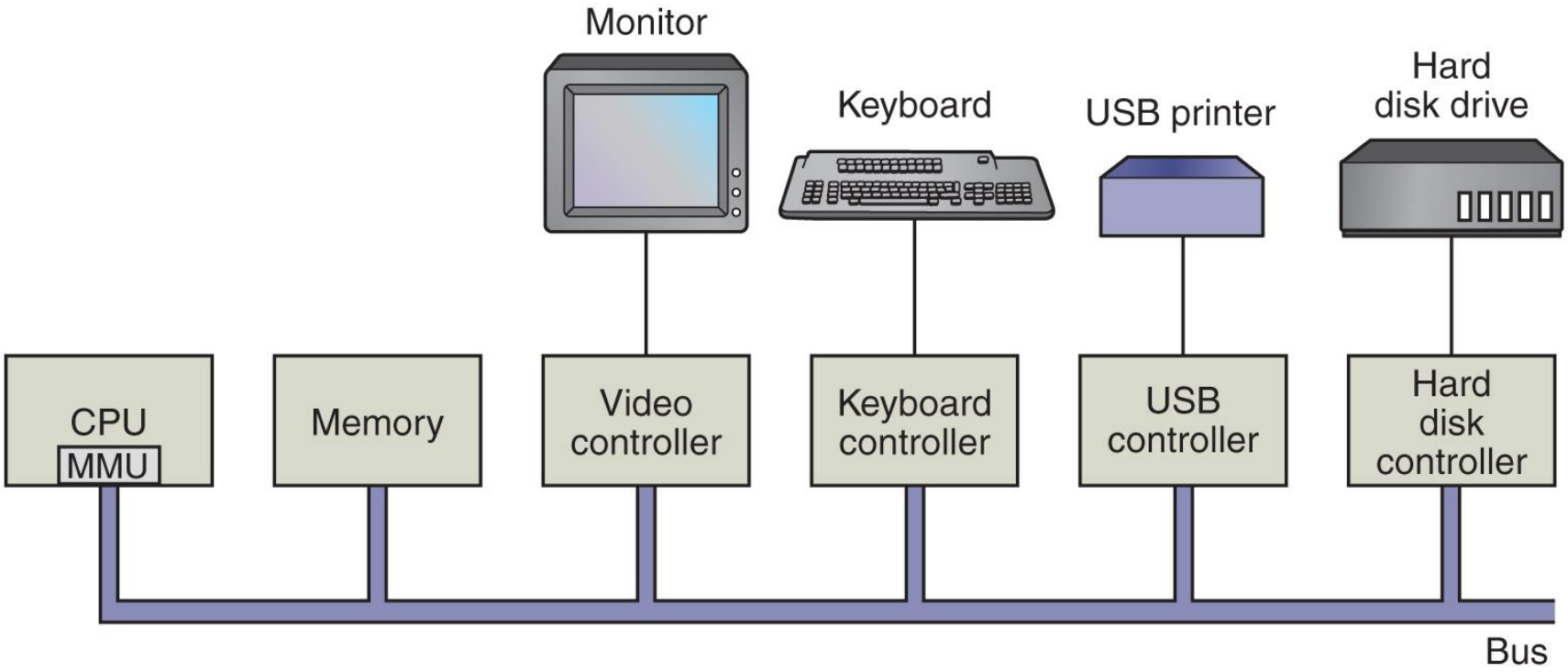
I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-hard disks, as that's all I have :-(”.

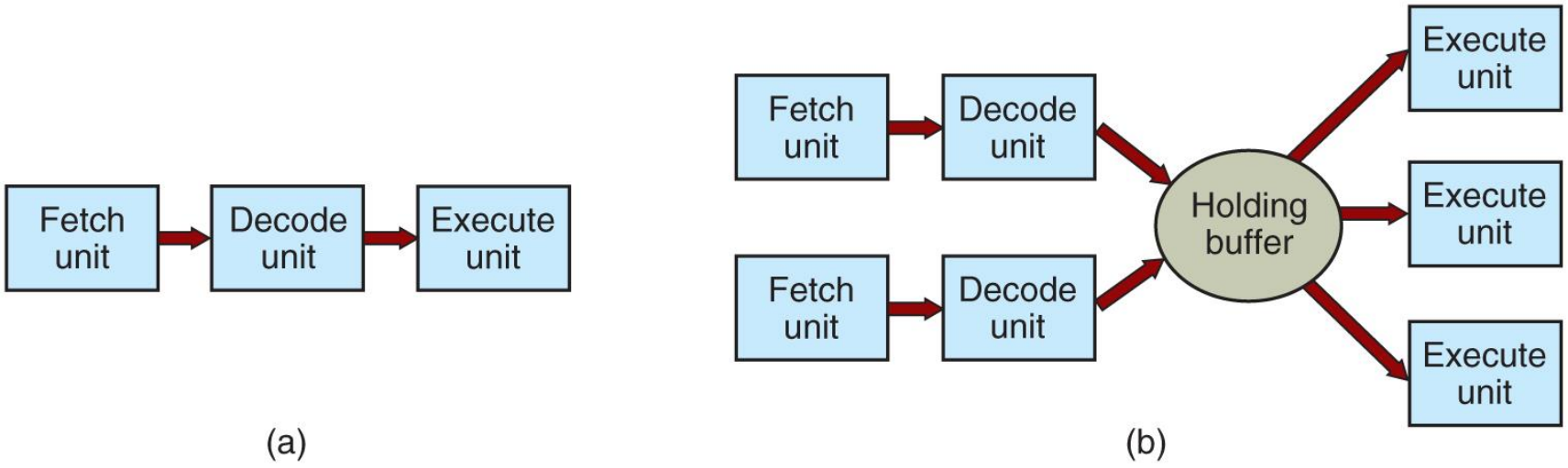
Hardware - Processors

Some of the components of a simple personal computer.



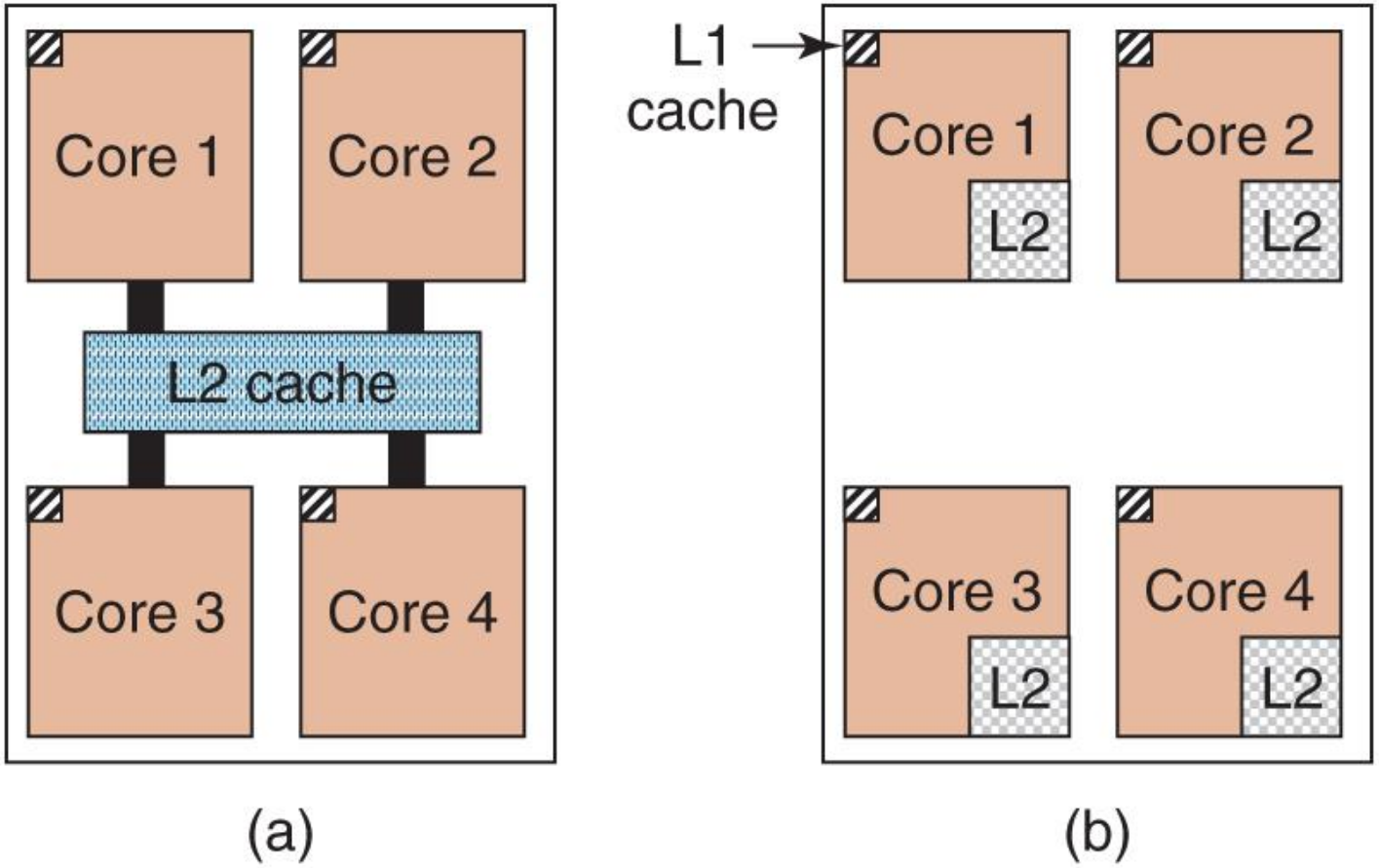
Hardware - Processors

a) A three-stage pipeline. (b) A superscalar CPU.



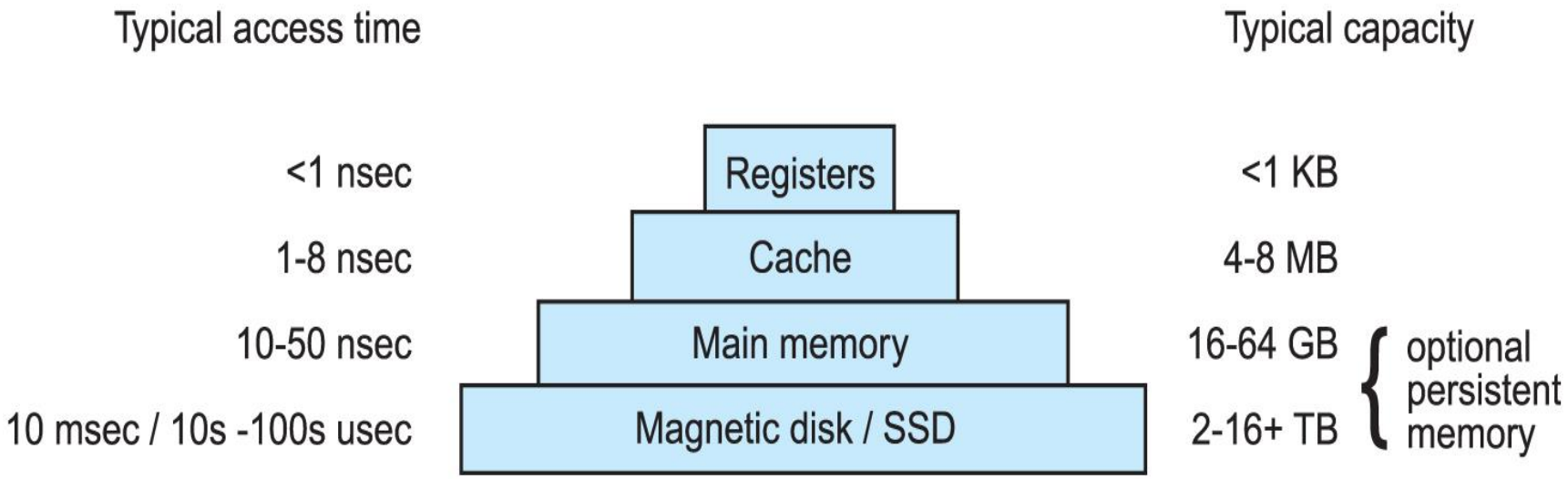
Hardware - Memory

(a) A quad-core chip with a shared L2 cache. (b) A quad-core chip with separate L2 caches.



Hardware - Memory

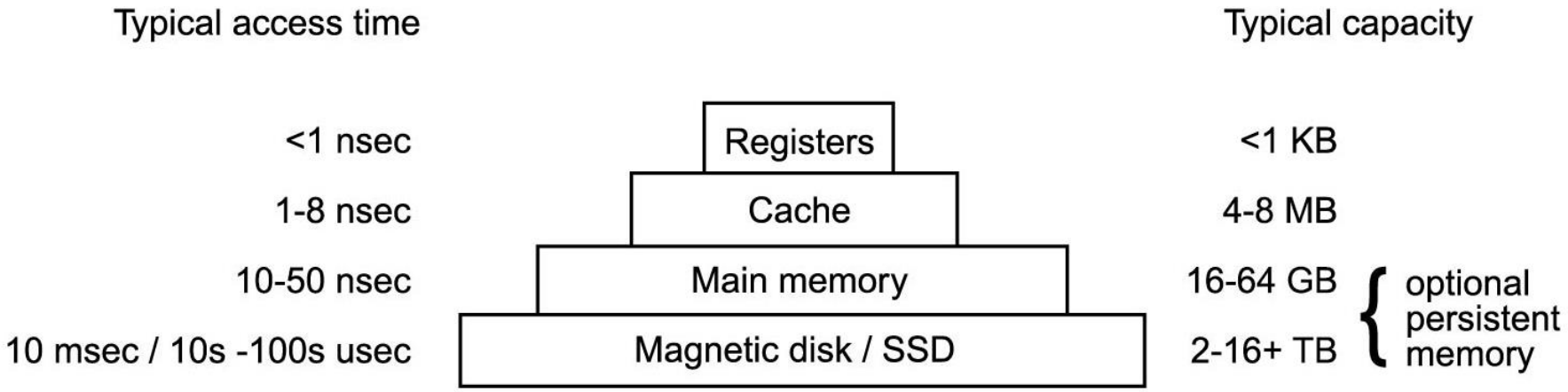
A typical memory hierarchy. The numbers are very rough approximations.



Hardware - Memory

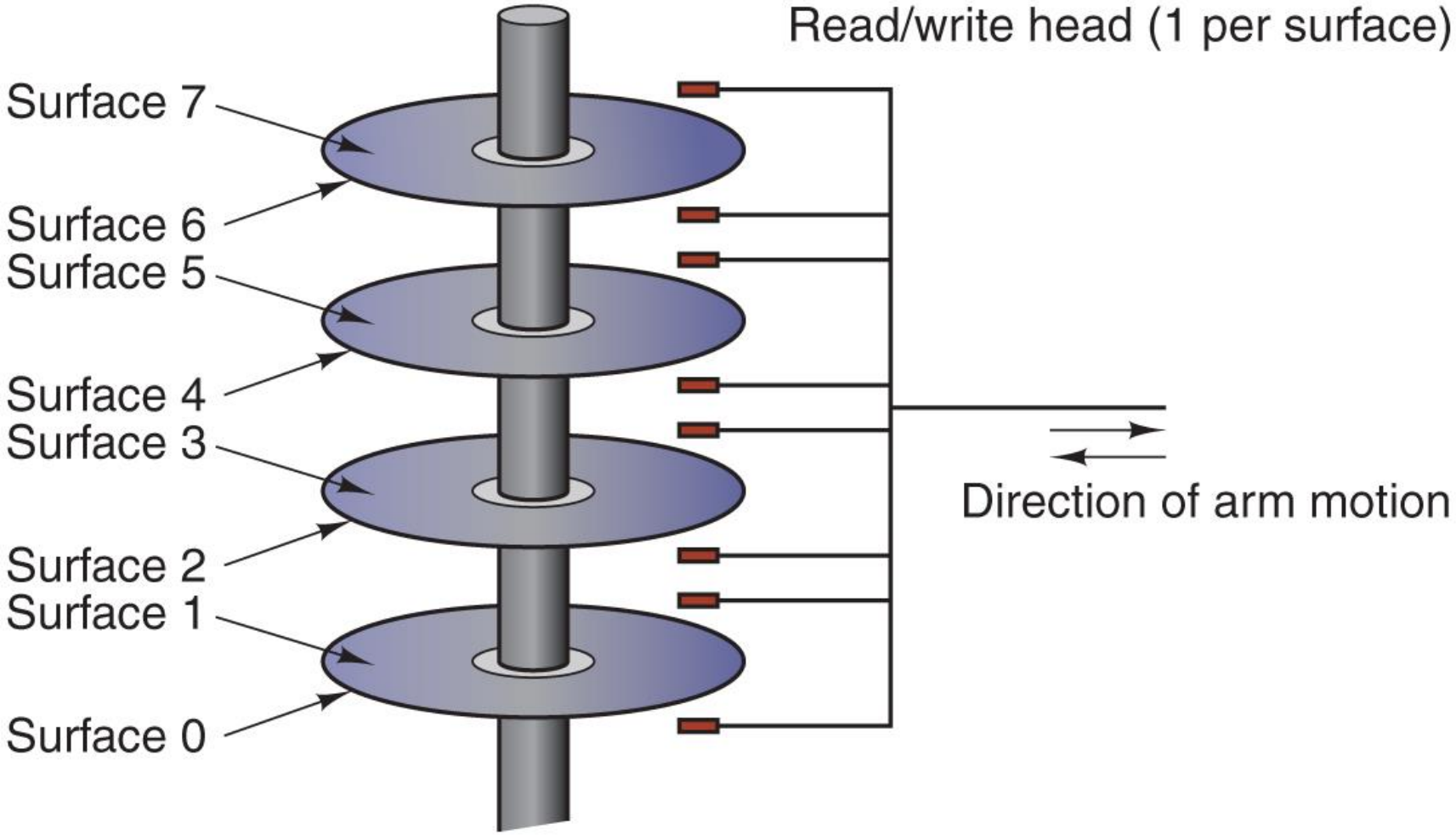
Caching system issues:

1. When to put a new item into the cache.
2. Which cache line to put the new item in.
3. Which item to remove from the cache when a slot is needed.
4. Where to put a newly evicted item in the larger memory.



Nonvolatile Storage

Structure of a disk drive.



Solid State Drives (SSD)

Often (incorrectly) referred to as disk also. No moving parts, data in electronic (flash) memory. Much faster than magnetic disks.

I/O Devices

Many types of I/O devices.

Typically consist of 2 parts:

- Controller
  - Chips that control device, receive commands from OS (e.g., to read data) Example: SATA disk controller
- Device itself
  - Generally simple interface (so SATA controller can handle any SATA disk)

Device driver: OS component that talks to controller

One for each type of device controller

I/O Devices

Device driver communicates with controller via registers

Example: disk controller may have registers for

- disk address,
- memory address
- sector count
- direction (read or write)

Device registers are either accessed using special instructions (e.g., IN/OUT). or mapped in the OS' address space (the addresses it can use).

The collection of device registers forms the **I/O Port Space**

I/O Devices

To perform I/O:

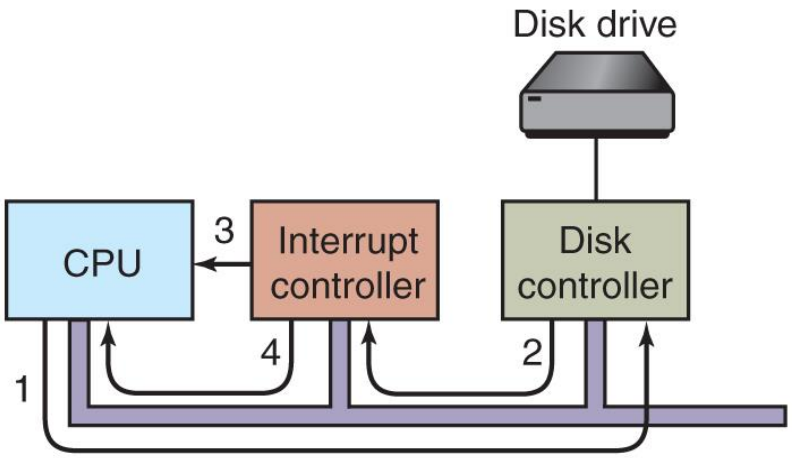
- Process executes system call
- Kernel makes a call to driver
- Driver starts I/O

Either polls device to see if it is done (busy waiting)

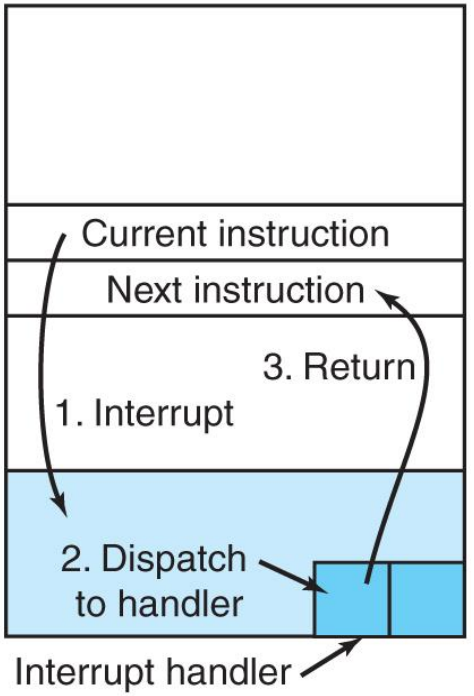
or asks device generate an interrupt when it is done
more advanced: make use of special hardware – DMA

I/O Devices

(a) The steps in starting an I/O device and getting an interrupt. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

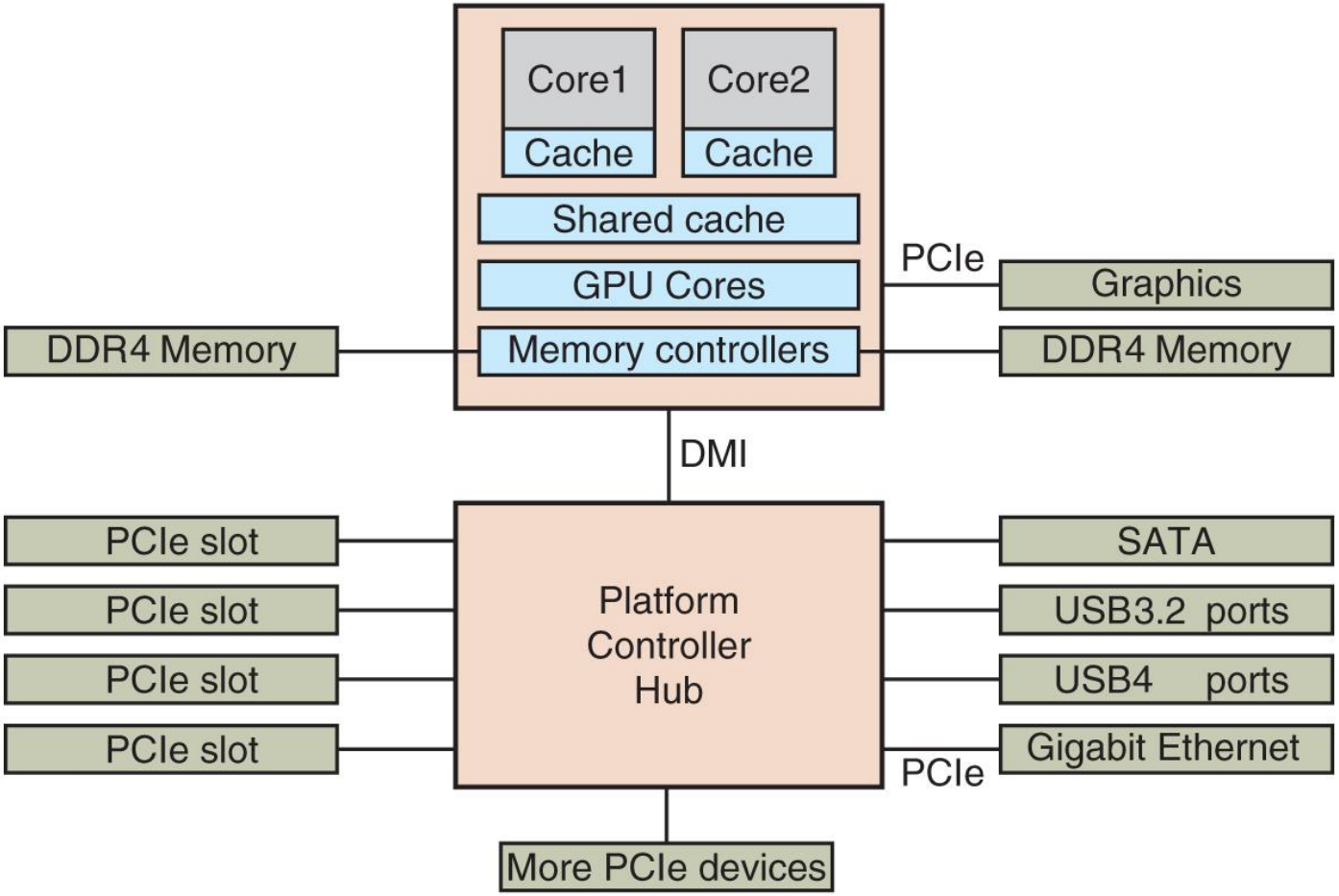


(a)



(b)

Busses



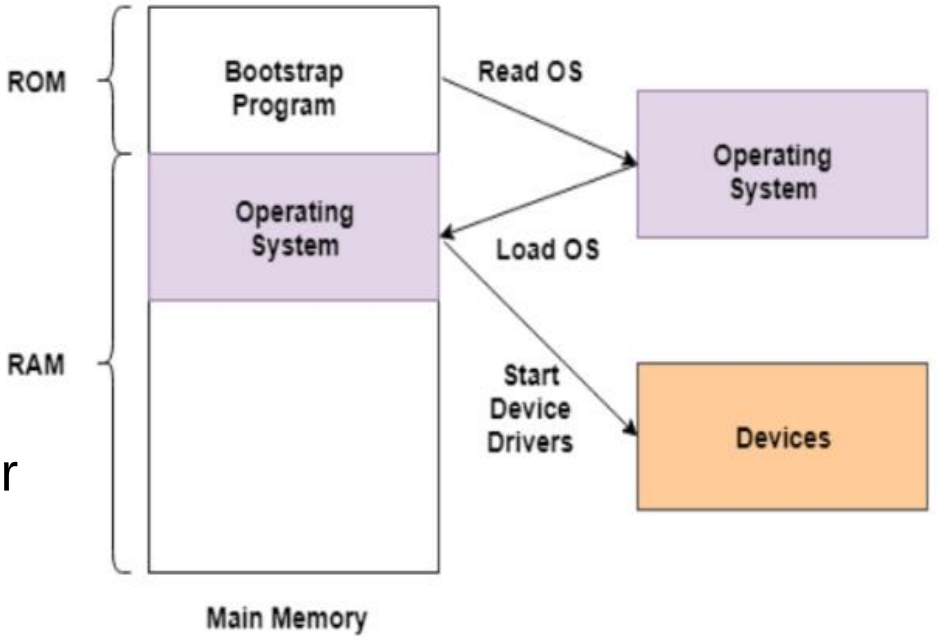
The structure of a large x86 system: many buses (e.g., cache, memory, PCIe, USB, SATA, and DMI)

Computer Startup

Flash memory on motherboard contains firmware (a.k.a. BIOS)

After pressing power button, CPU executes BIOS which

- Initializes RAM and other resources
- Scans PCI/PCIe buses and initializes devices
- Sets up the runtime firmware for critical services (e.g., low-level I/O) to be used by the system after booting



BIOS looks for location of partition table on second sector of boot device

- Contains locations of other partitions

Operating System Types

- Mainframe Operating Systems
- Server Operating Systems
- Personal Computer Operating Systems
- Smartphone and Handheld Computer Operating Systems
- The Internet of Things (IOT) and Embedded Operating Systems
- Real-Time Operating Systems
- Smart Card Operating Systems

What is an Operating System

- Extended Machine
  - Extending the hardware functionality
  - Abstraction over hardware
  - Hiding details from the programmer
- Resource Manager
  - Protects simultaneous/unsafe usage of resources
  - Fair sharing of resources
  - Resource accounting/limiting

Operating System Concepts

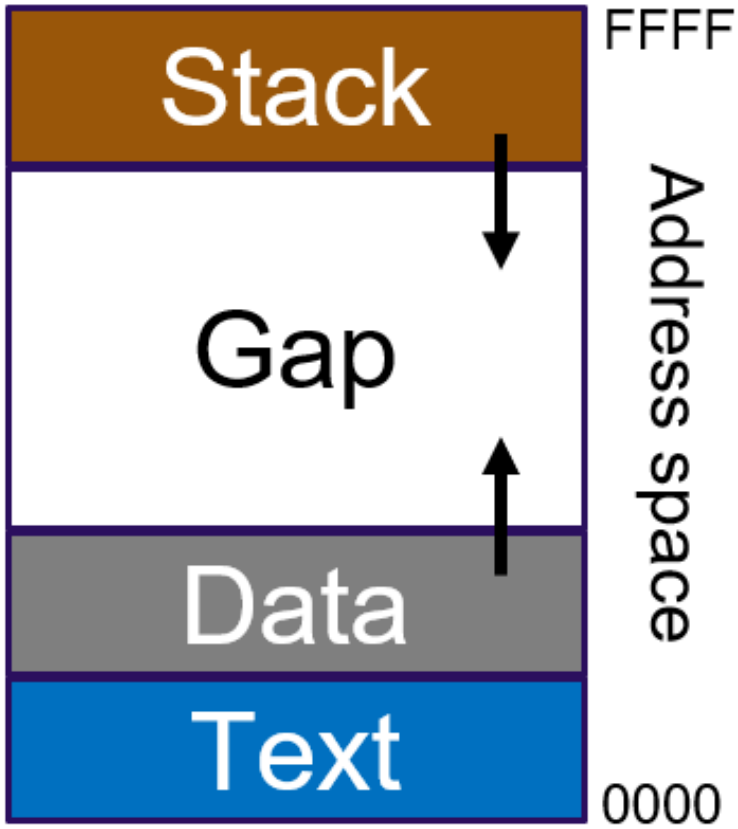
- OS offers functionality through **system calls**
- Groups of system calls implement **services**
  - File System Service
  - Process Management Service
- **Processes** are user-level abstractions to execute a program on behalf of a user
- Each process has its own **address space**
- Data involved in this processing is retrieved from/stored in **files**
- Files persist over processes

Processes

- Key concept in all operating systems
- Definition: a program in execution
- Process is associated with an address space
- Also associated with set of resources (registers, open files, alarms, etc.)
- Process can be thought of as a container
  - Holds all information needed to run program

Processes

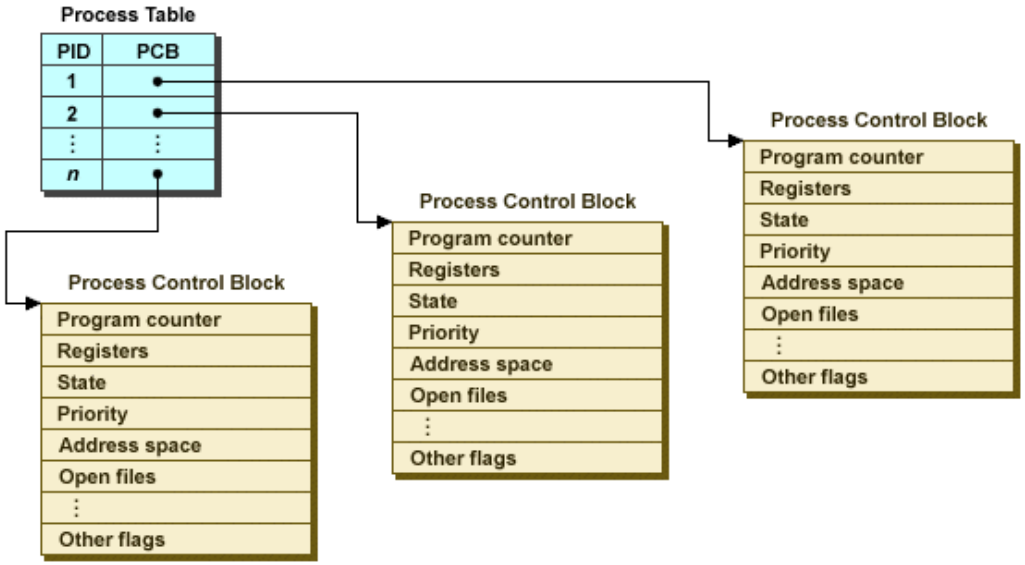
- Layout affected by:
  - Architecture
  - OS
  - Program
- **Very** basic layout:
  - Stack: Active call data
  - Data: Program variables
  - Text: Program code



Try it:
Linux : readelf –S <program name>
MAC : otool –l <program name>

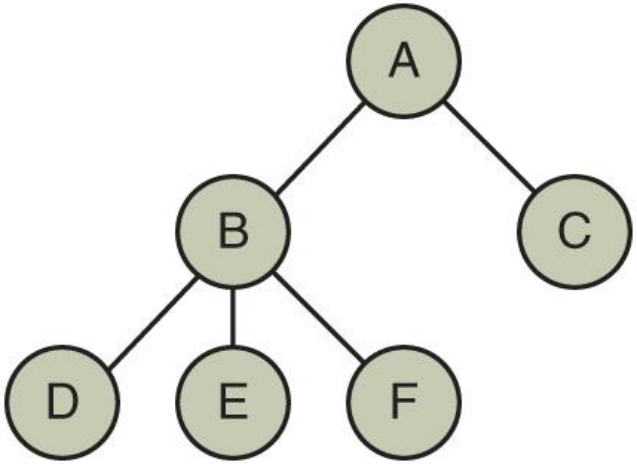
Processes

- Information about process is kept in the OS' process table
- Process Management
  - Operations such as creating, terminating, pausing and resuming a processes
- One process can create another process
  Known as a child process
  Creates a hierarchy (or "tree") of processes



Processes

A process tree. Process *A* created two child processes, *B* and *C*. Process *B* created three child processes, *D*, *E*, and *F*.



Processes are “owned” by a user, identified by a UID.

- Every process typically has the UID of the user who started it
- On UNIX, a child process has the same UID as its parent process
- Users can be members of groups, identified by a GUID

Try it:
Linux : pstree
MAC : Applications > Utilities > Activity Monitor or type “top”

Files

- File: abstraction of (possibly) real storage device (e.g., disk)
- You can read and write data from/to file by providing a position and an amount of data to transfer
- Files are maintained in **directories**
  - A directory keeps an identifier for each file it contains
  - A directory is a file by itself
  - The UNIX philosophy: *"Everything is a file"*.

Files

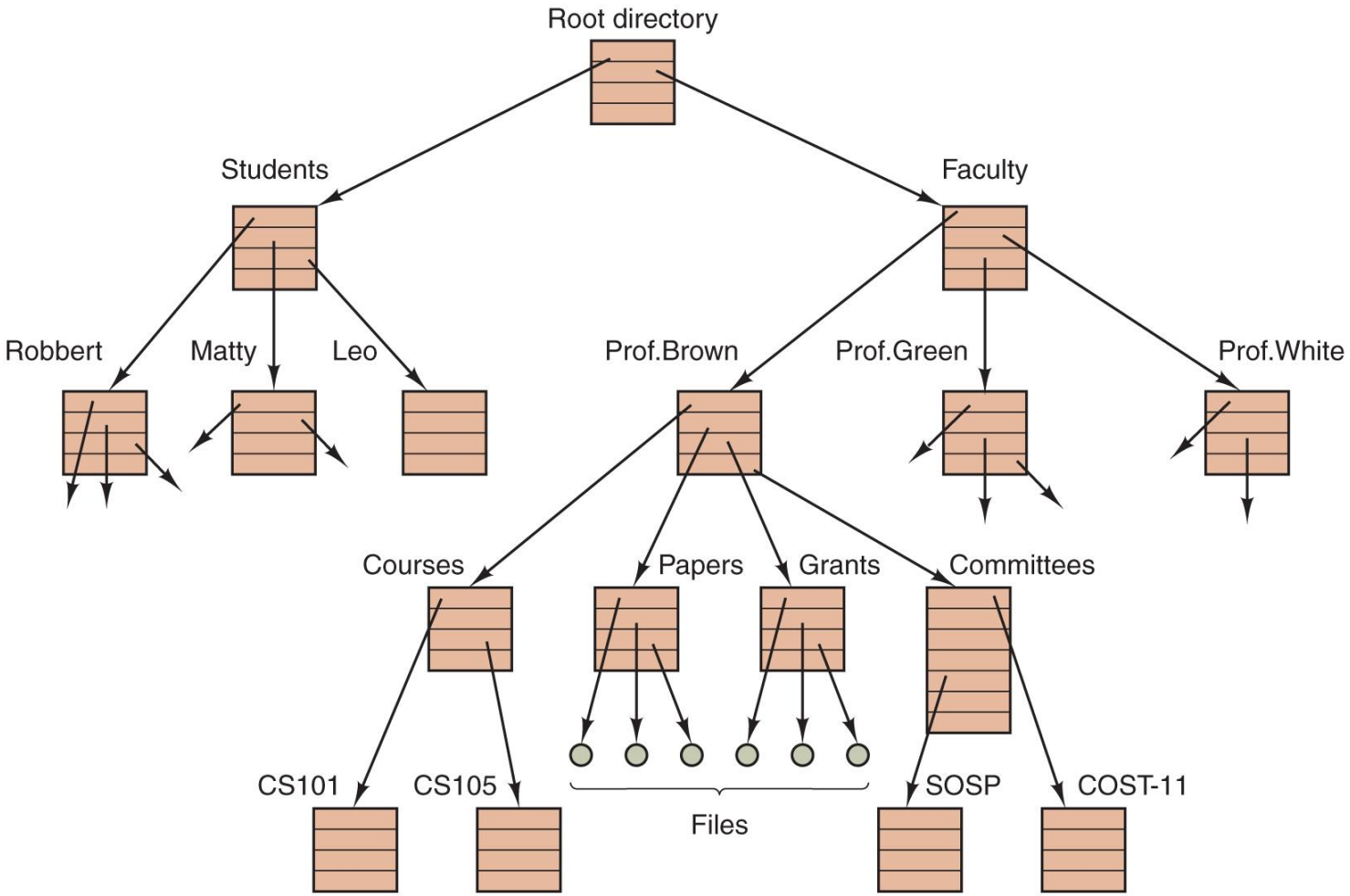
- Directories and files form a hierarchy:
  - Hierarchy starts at **root directory:**
    - `/`
  - Files can be accessed through **absolute paths:**
    - `/home/ast/todo-list`
  - … or relative paths starting from the **current working directory:**
    - `../courses/slides1.pdf`
  - Other filesystems can be **mounted** into the root:
    - `/mnt/windows`
  - Filesystems mounting on Windows?

Files

- Files are “protected” by three bit tuples for **owner**, **group** and **other** users
- Tuples contain a **(r)**ead, **(w)**rite and an e**(x)**ecute bit (but more bits are available)
  - Example:
    - `-rwxr-x--x myuser my group 14492 Dec 4 18:04 my file`
  - Owner is allowed to execute, modify, read the file
  - Group is allowed to read and execute the file
  - Other users are only allowed to execute the file
- **x** bit for directories?

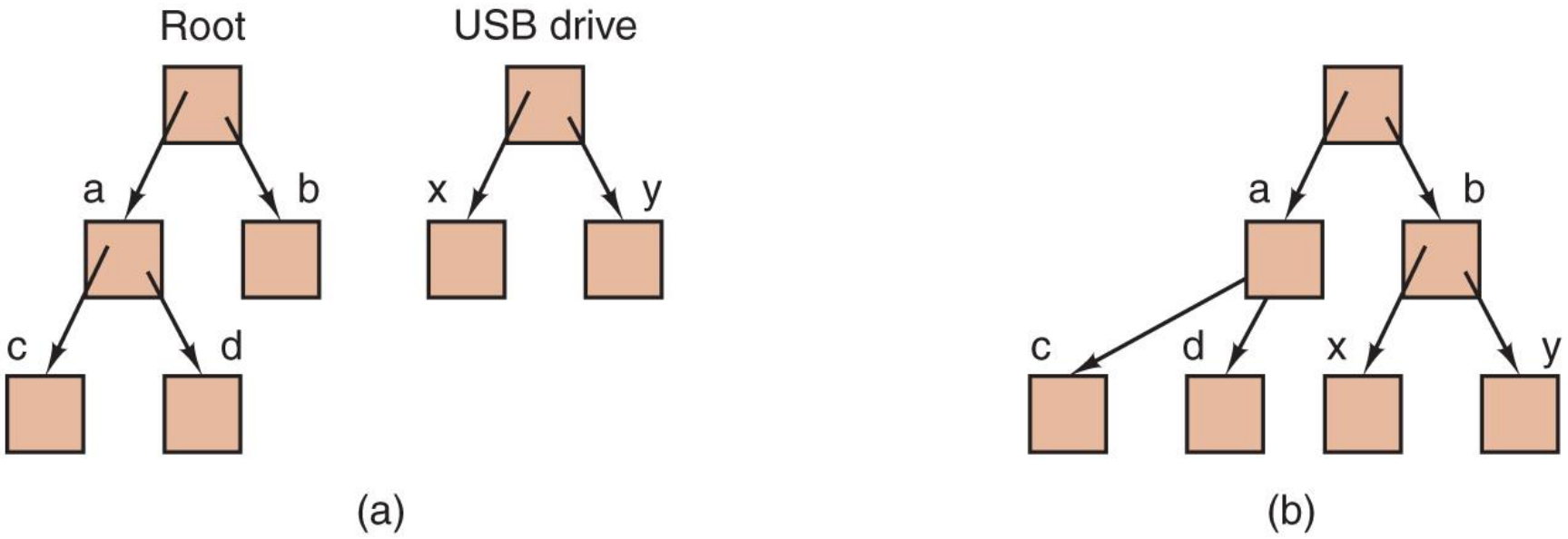
Files

A file system for a university department.



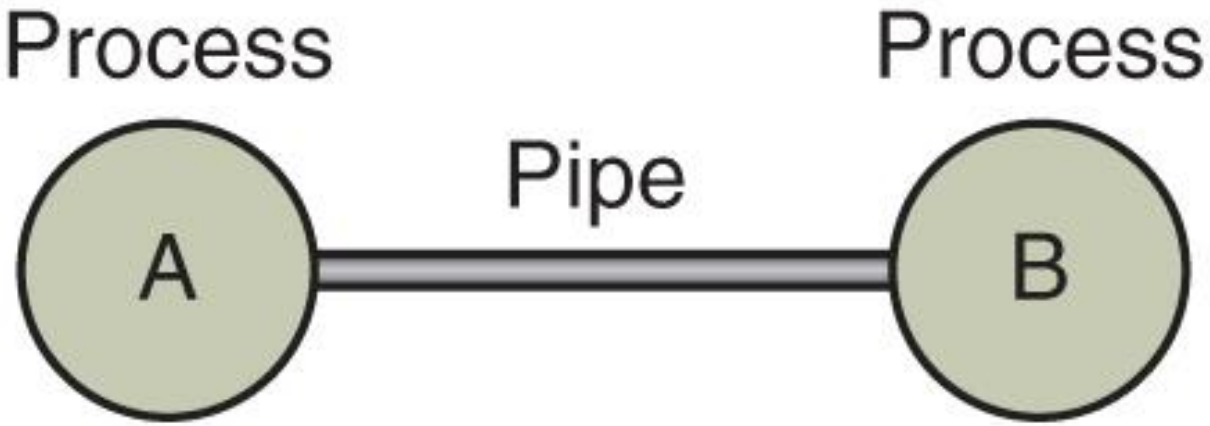
File System Mounting

(a) Before mounting, the files on the USB drive are not accessible. (b) After mounting, they are part of the file hierarchy.



Files (Pipes)

Two processes connected by a pipe.



- Pipes: pseudo files allowing for processes to communicate over a FIFO channel
- Has to be set up in advance
- Looks like a "*normal*" file to `read` and `write` from/to running processes

System Calls

System calls are the interface the OS offers to applications to request services

Problem:

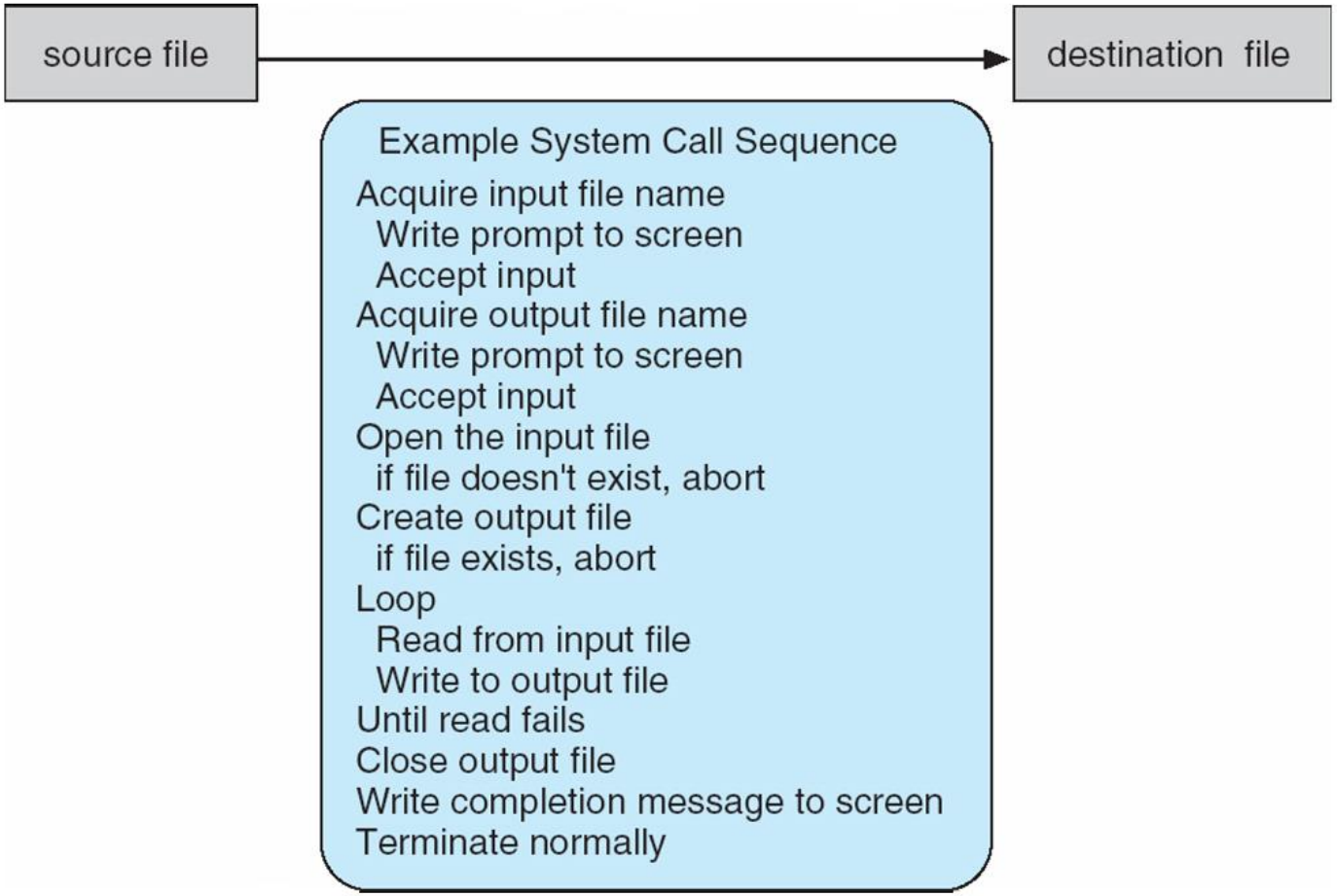
- System call mechanism is highly operating system and hardware specific
- The need for efficiency exacerbates this problem

Solution:

- Encapsulate system calls in the C library (**`libc`**)
- **Typically** exports 1 library call for each system call
- UNIX **`libc`** based on the C POSIX library
- Note that many UNIX C libraries exist…

System Calls

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



System Calls

Some of the major POSIX system calls. The return code *s* is −1 if an error has occurred. The return codes are as follows: *pid* is a process id, *fd* is a file descriptor, *n* is a byte count, *position* is an offset within the file, and *seconds* is the elapsed time. The parameters are explained in the text.

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory- and file-system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

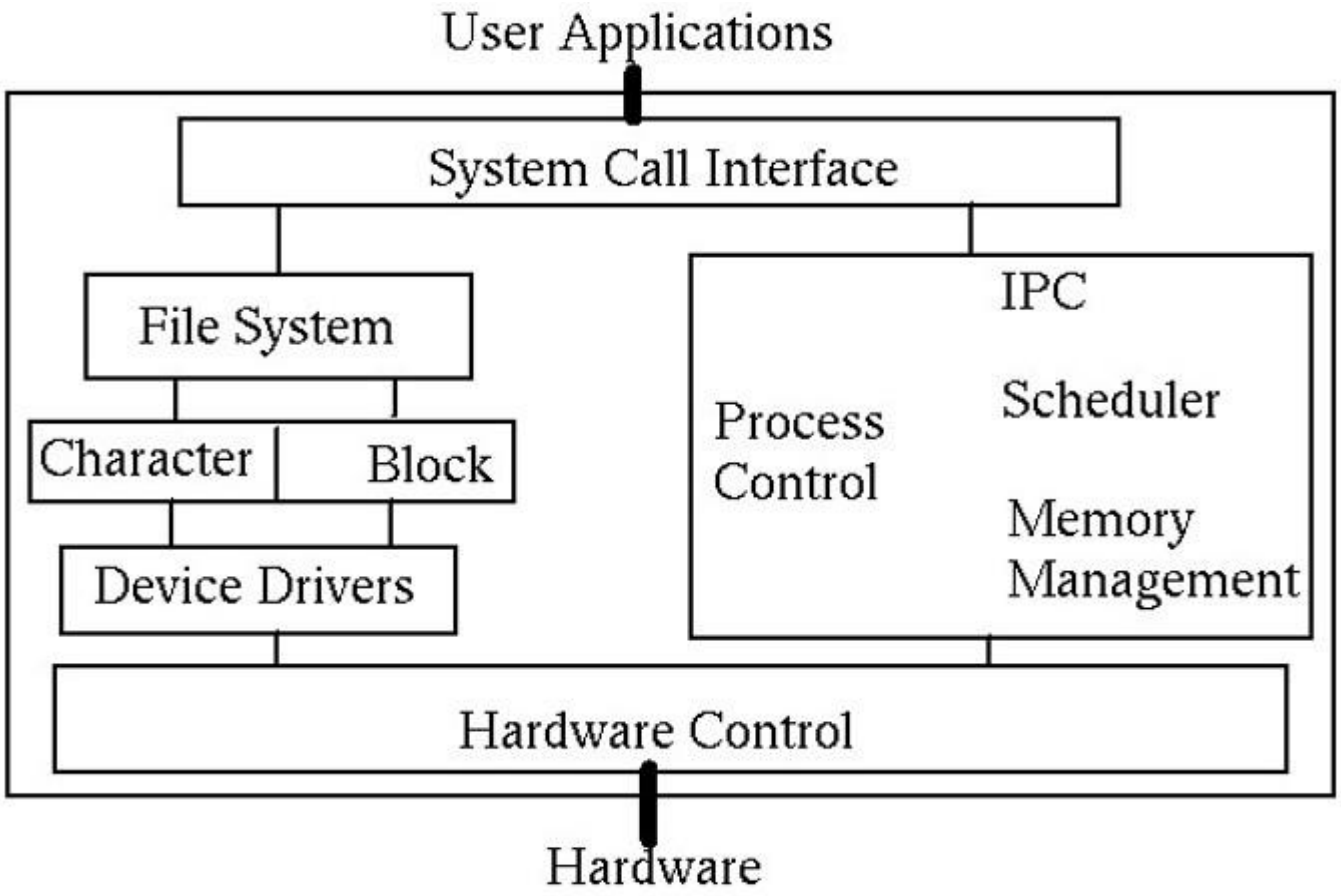
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Windows System Call API

The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1.18. It is worth emphasizing that Windows has a very large number of other system calls, most of which do not correspond to anything in UNIX.

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, so no umount
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

OS Structure: Monolithic

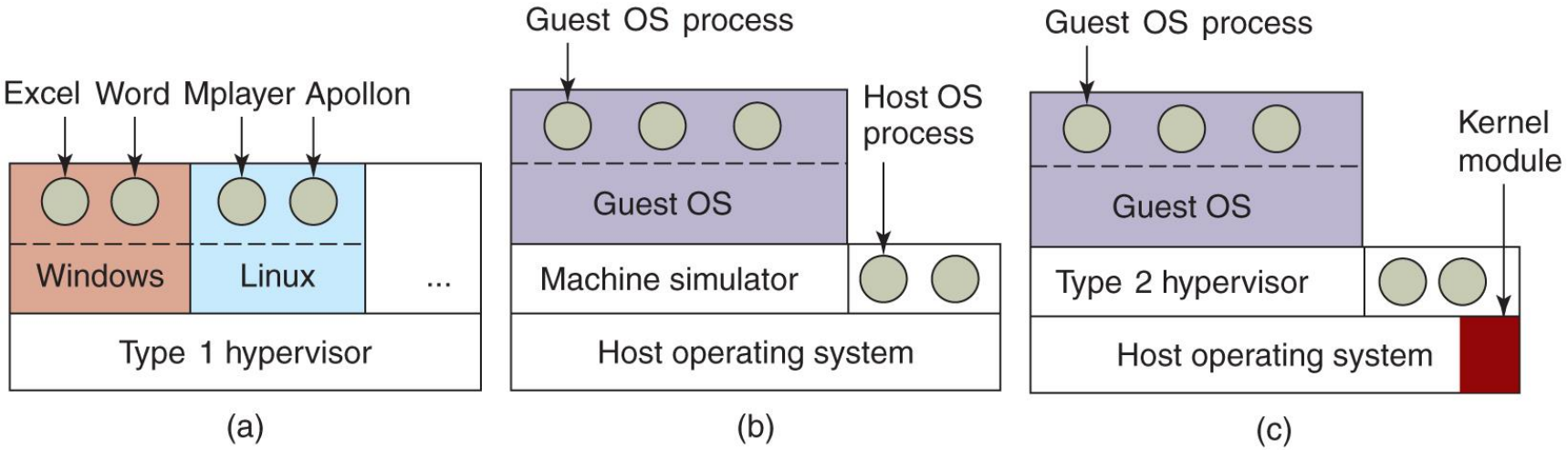


Typical UNIX structure

OS structure: Virtualization

Virtual machine monitor (VMM) or Hypervisor emulates hardware

- **Type 1:** VMM runs on bare metal (e.g., Xen)
- **Type 2:** VMM hosted in the OS (e.g., QEMU)
- **Hybrid:** VMM inside the OS (e.g., KVM)



(a) A type 1 hypervisor. (b) A pure type 2 hypervisor. (c) A practical type 2 hypervisor

Containers

- Containers can run multiple instances of an OS on a single machine
- Each container shares the host OS kernel and the binaries and libraries
  - Container does not contain full OS and therefore can be lightweight
- Downsides to containers
  - Cannot run a container with a completely different OS than the host
  - Unlike with virtual machines no strict resource partitioning
  - Containers are process-level isolated
    - If a container alters the stability of the underlying kernel this may affect other containers

The World According to C

- C was created by Dennis Ritchie in 1972 to develop UNIX programs
- Some of the UNIX roots still visible
- “Everything is a file”

- UNIX/Linux – Everything is a file
  - Sockets
  - Devices
  - Hard drives
  - Printers
  - Pipes

The World According to C

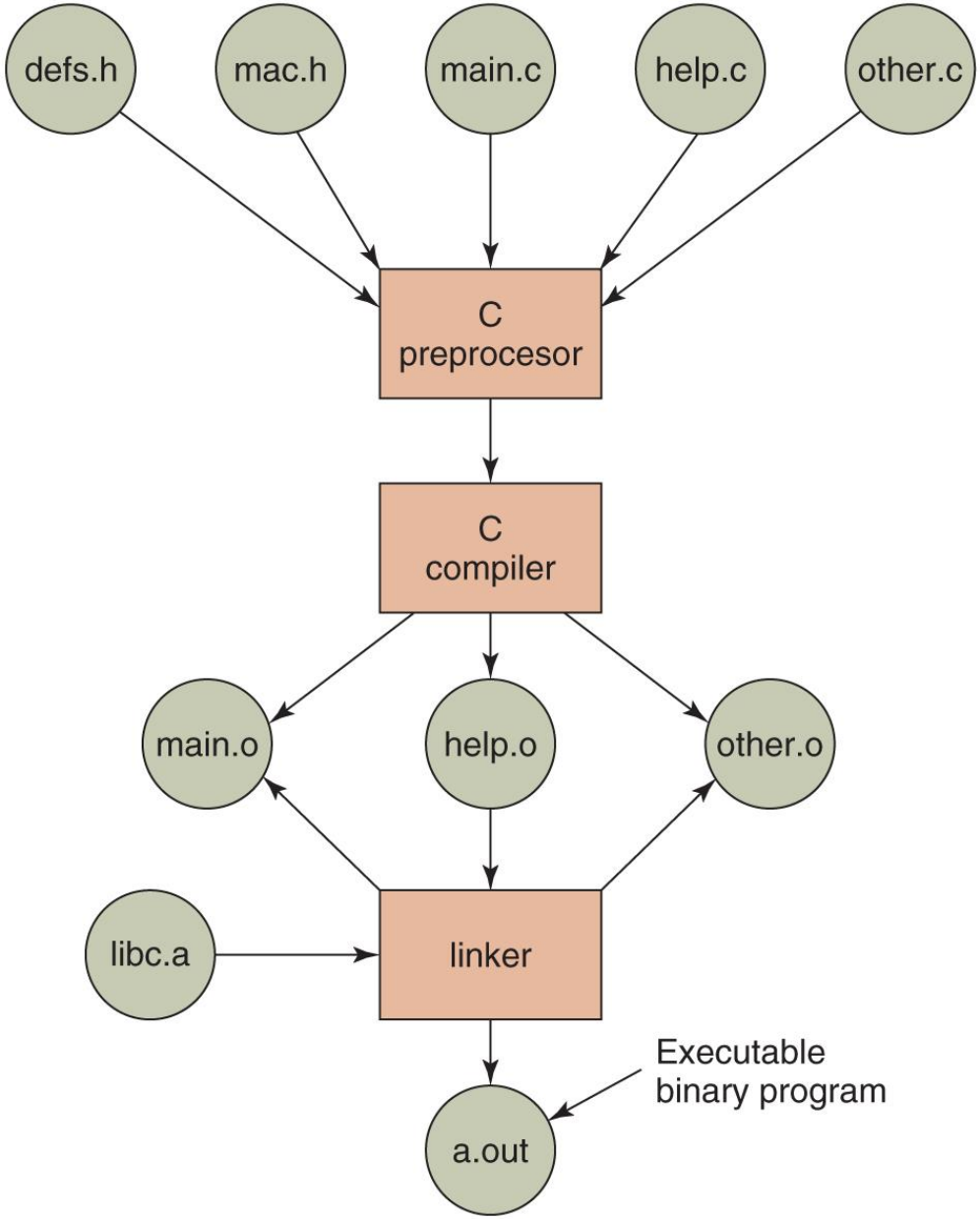
What do we have to do to print “Hello World” on the console (Standard output)?

- **printf**(char *str, …);

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```

Build Process

The process of compiling C and header files to make an executable.



Hello World Example

What do we have to do to print “Hello World” on the console (Standard output)?

- `printf(char *str, …);`
- `int write(int fd, char *buf, size_t len);`

Recall: Everything Is a File!

Descriptive Name	Short Name	File Number	Description
Standard In	stdin	0	Input from the keyboard
Standard Out	stdout	1	Output to the console
Standard Error	stderr	2	Error output to the console

Default every process starts with 3 “files” opened

Hello World Example

```
#include <unistd.h>
#define STDOUT 1

int main(int argc, char **argv)
{
    char msg[] = "Hello World!\n";
    write(STDOUT, msg, sizeof(msg));
    return 0;
}
```

Hello World Example

What do we have to do to print “Hello World” on the console (Standard output)?

- `printf(char *str, …);`
- `int write(int fd, char *buf, size_t len);`
- `int syscall(int number, ...);`

Hello World Example

```
#define _GNU_SOURCE
#include <sys/syscall.h>
#define STDOUT 1

int main(int argc, char **argv)
{
    char msg[] = "Hello World!\n";
    int nr = SYS_write;
    syscall(nr, STDOUT, msg, sizeof(msg));
    return 0;
}
```

Standard C Library

- Libc provides useful wrappers around syscalls
  - E.g., `write`, `read`, `exit`
- Need to call the `syscall` or `int 0x80` instruction
  - Done in assembly (inline, or separate object)
- `syscall(int nr, …)`

Process Management System Calls

Consider a minimal shell:

- Waits for user to type in a command
- Starts a process to execute the command
- Waits until the process has finished

(fork, wait, execv)

Process Creation

- `pid_t fork()`
  - Duplicates the current process
  - Returns child pid in caller (parent)
  - Returns 0 in new (child) process
- `pid_t wait(int *wstatus)`
  - Waits for child processes to change state
  - Writes status to `wstatus`
  - E.g., due to exit or signal

Process Creation (fork, wait)

```
void main(void)
{
     int pid, child_status;
     if (fork() == 0) {
          do_something_in_child();
     } else {
          wait(&child_status); // Wait for child
     }
}
```

Process Creation (execv)

- `int execv(const char *path, char *constargv[ ]);`
  - Loads a new binary (path) in the current process, removing all other memory mappings.
  - `constargv` contains the program arguments
  - Last argument is NULL
  - E.g., `constargv = {“/bin/ls”, “-a”, NULL}`
  - Different exec(v)(p) variants (check man pages)

Process Creation (fork, wait, execv)

```
void  main(void)
{
     int    pid, child_status;
     char   *args[] = {“/bin/ls”, “-l”, NULL};
     if (fork() == 0) {       // fork creates child process
      execv(args[0], args); // in child: load+execute program
     } else {
         wait(&child_status);        // Wait for child
     }
}
```

Minimal Shell

```
while (1) {
    char cmd[256], *args[256];
    int status;
    pid_t pid;
    read_command(cmd, args); /* reads command and arguments
from command line */

    pid = fork();

    if (pid == 0) {
        execv(cmd, args);
        exit(1);
    } else {
        wait(&status);
    }
}
```

How to Exit Programs?

- Ctrl+C, but how does this work??
- Answer: **signals**

- Processes sometimes need to be interrupted during their execution
- A **signal** is sent to the process that needs to be interrupted
- Interrupted process can catch the signal by installing a **signal handler**
- What happens when a terminal user hits CTRL+C or CTRL+Z

(signal, alarm, kill)

Signals

- `sighandler_t signal(int signum, sighandler_t handler)`
  - Registers a signal handler for signal signum
- `unsigned int alarm(unsigned int seconds)`
  - Deliver SIGALRM in specified number of seconds
- `int kill(pid_t pid, int sig)`
  - Deliver signal sig to process `pid`

Alarm Example

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void alarm_handler(int signal)
{
    printf(“In signal handler: caught signal %d!\n",signal);
    exit(0);
}
int main(int argc, char **argv)
{
    signal(SIGALRM, alarm_handler);
    alarm(1); // alarm will send signal after 1 sec

    while (1) {
        printf("I am running!\n");
    }
    return 0;
}
```

Pipe Example

What happens if we execute the following command?

```
$ cat names.txt | sort
```

And how about the following commands?

```
$ mkfifo named.pipe
$ echo “Hello World!” > named.pipe
$ cat named.pipe
```

(open, close, pipe, dup)

Pipe Example (open, close, pipe, dup)

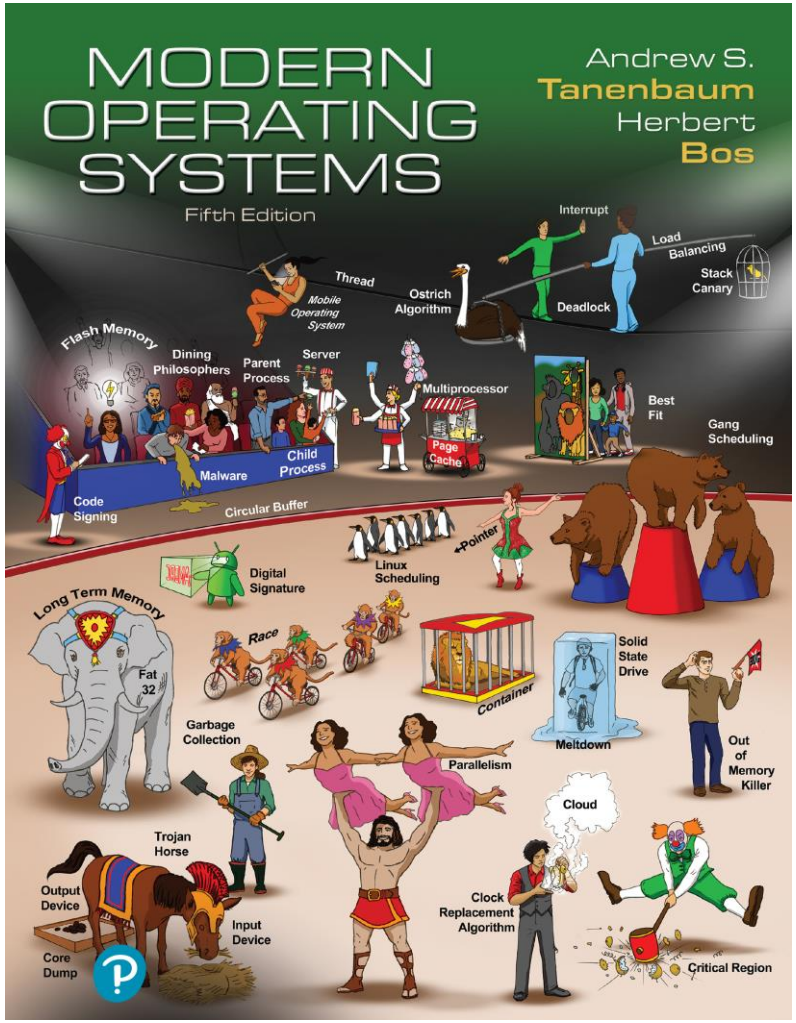
- `int open(const char *pathname, int flags)`
  - Opens the file specified by `pathname`
- `int close(int fd)`
  - Closes the specified file descriptor `fd`
- `int pipe(int pipefd[2])`
  - Creates a pipe with two `fds` for the ends of the pipe
- `int dup(int oldfd)`
  - Creates a copy of the `oldfd` file descriptor using the lowest-numbered unused file descriptor for the copy

Pipe Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define STDIN  0
#define STDOUT 1

#define PIPE_RD 0
#define PIPE_WR 1

int main(int argc, char** argv)
{
    pid_t cat_pid, sort_pid;
    int fd[2];

    pipe(fd);

    cat_pid = fork();
    if ( cat_pid == 0 ) {
        close(fd[PIPE_RD]);
        close(STDOUT);
        dup(fd[PIPE_WR]);
        execl("/bin/cat", "cat", "names.txt" , NULL);

    }

    sort_pid = fork();
    if ( sort_pid == 0 ) {
        close(fd[PIPE_WR]);
        close(STDIN);
        dup(fd[PIPE_RD]);
        execl("/usr/bin/sort", "sort", NULL);
    }

    close(fd[PIPE_RD]);
    close(fd[PIPE_WR]);


    /* wait for children to finish */
    waitpid(cat_pid, NULL, 0);
    waitpid(sort_pid, NULL, 0);

    return 0;
}
```

Modern Operating Systems

Fifth Edition



End of Introduction