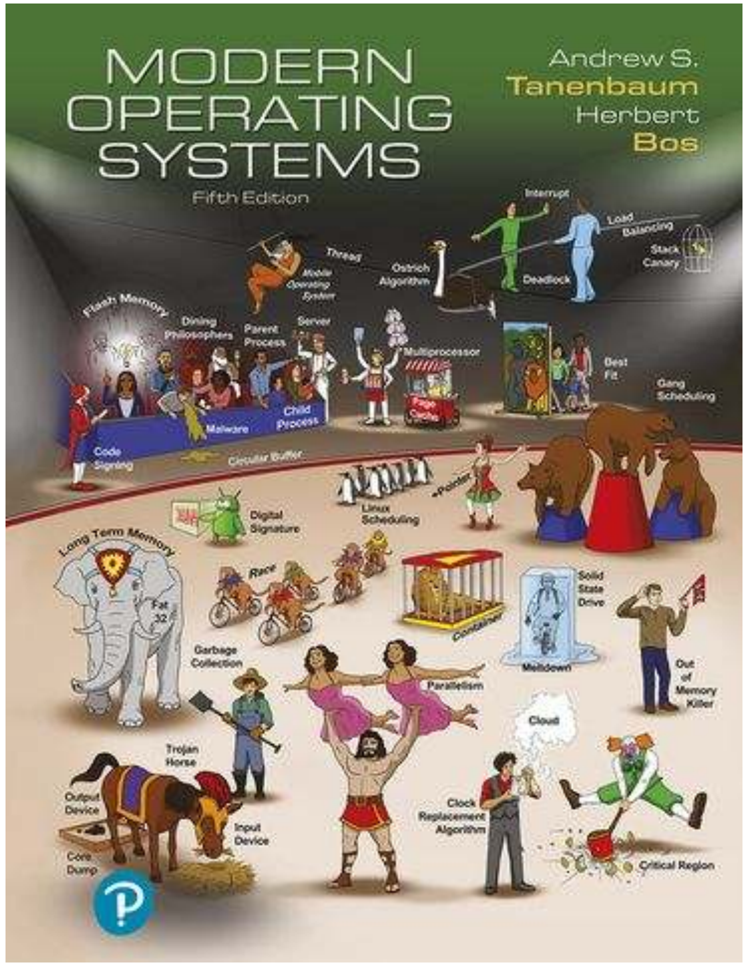
Modern Operating Systems

Fifth Edition

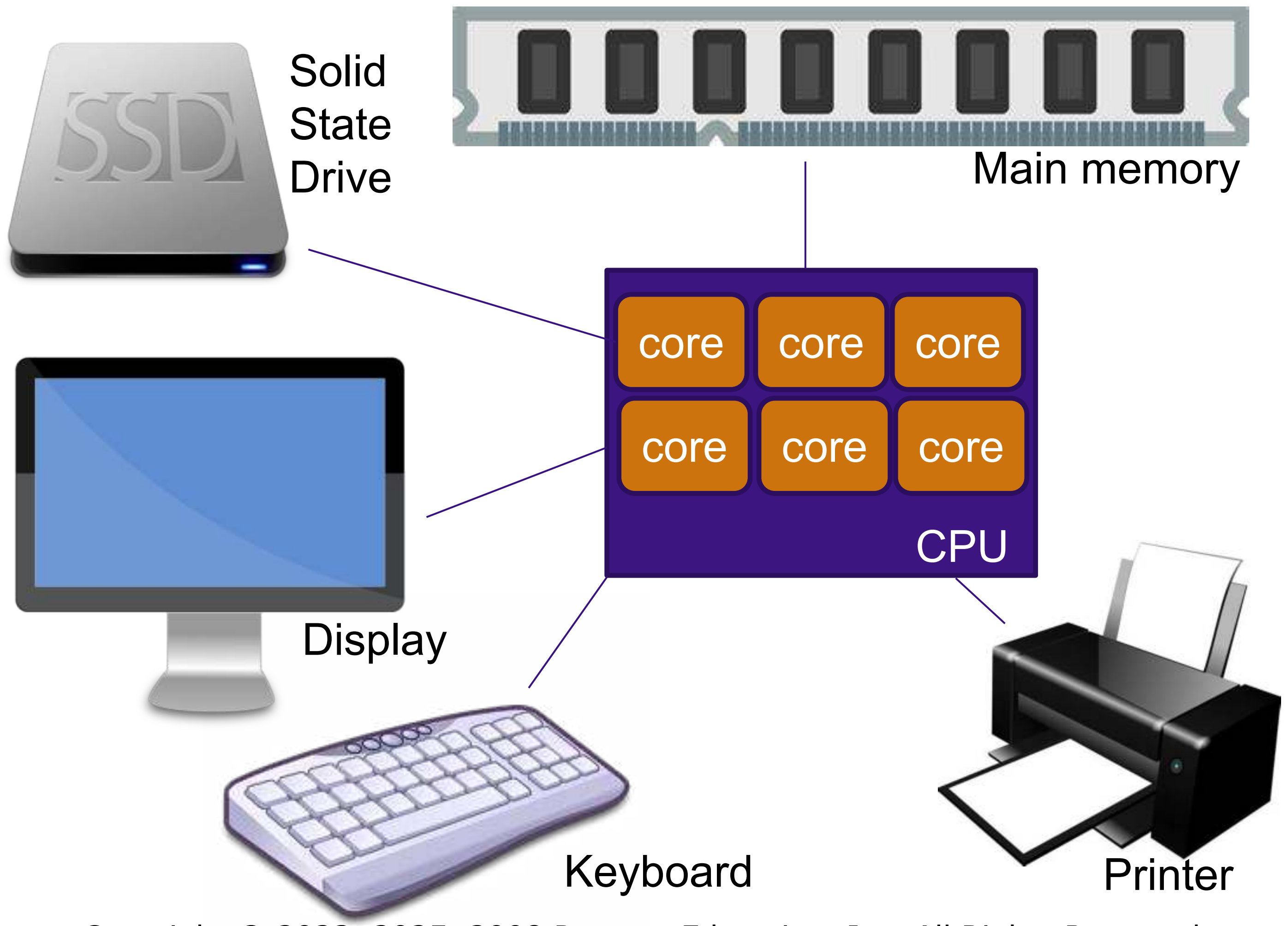


Chapter 1

Introduction

Components of a Modern Computer (1 of 2)

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



Components of a Modern Computer (2 of 2)

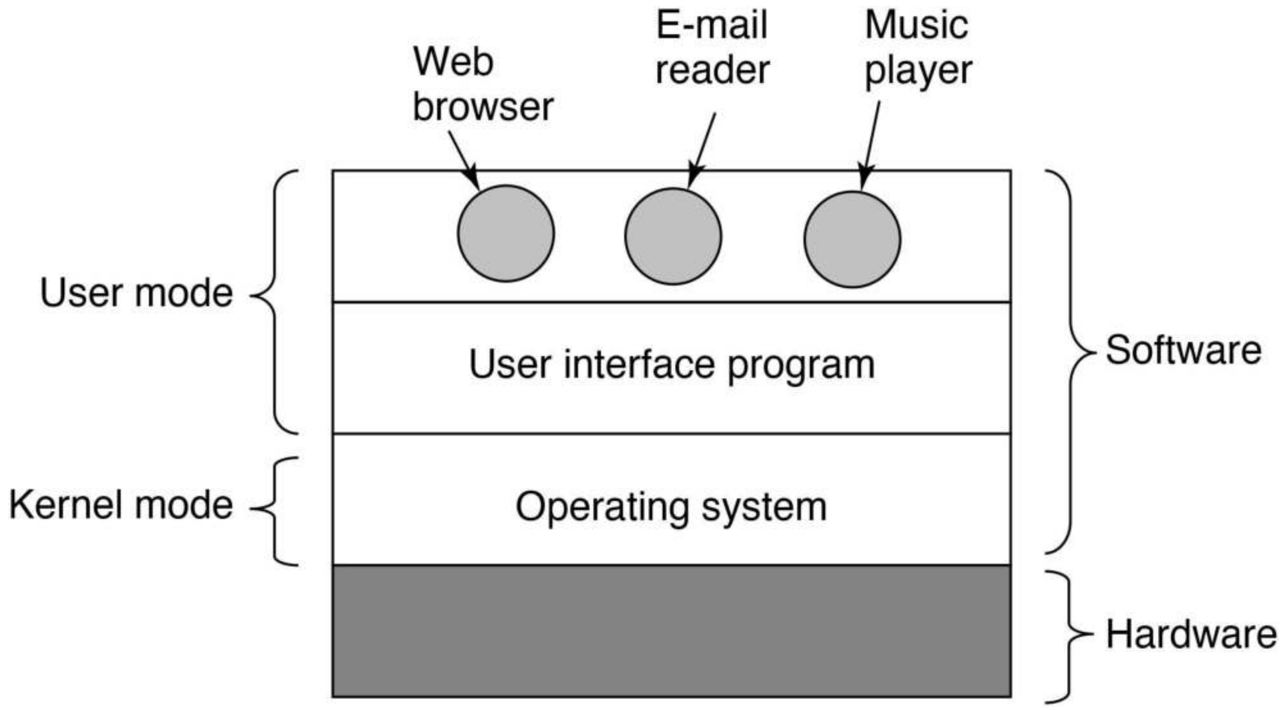


Figure 1-1. Where the operating system fits in.

The Operating System as an Extended Machine

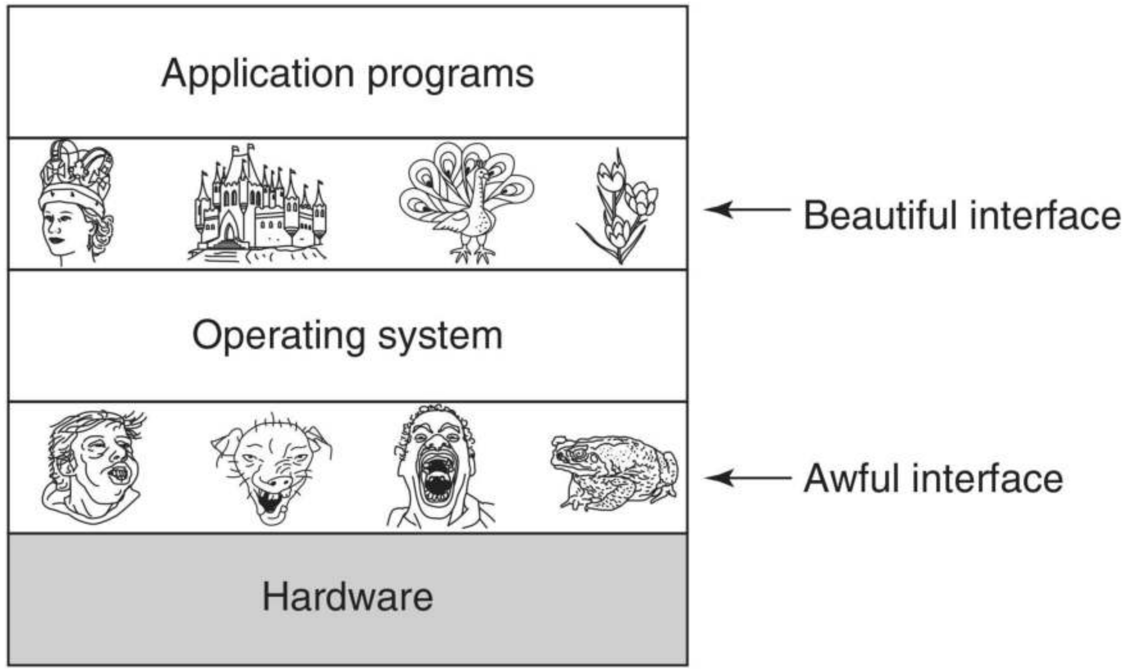


Figure 1-2. Operating systems turn awful hardware into beautiful abstractions.

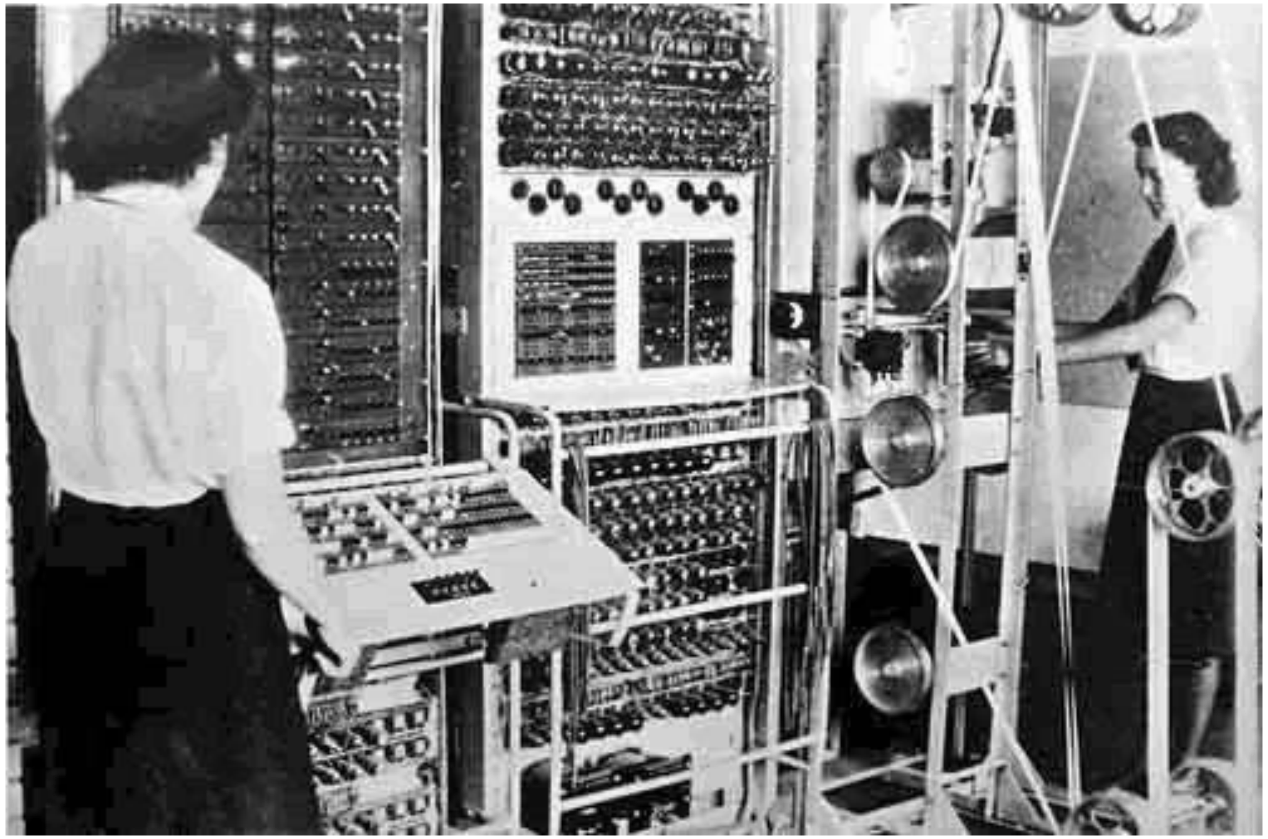
The Operating System as a Resource Manager

- 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



The Colossus Mark 2 computer: Cryptanalysis of the Lorenz Cipher

Second Generation: Transistors and Batch Systems
(1 of 3)

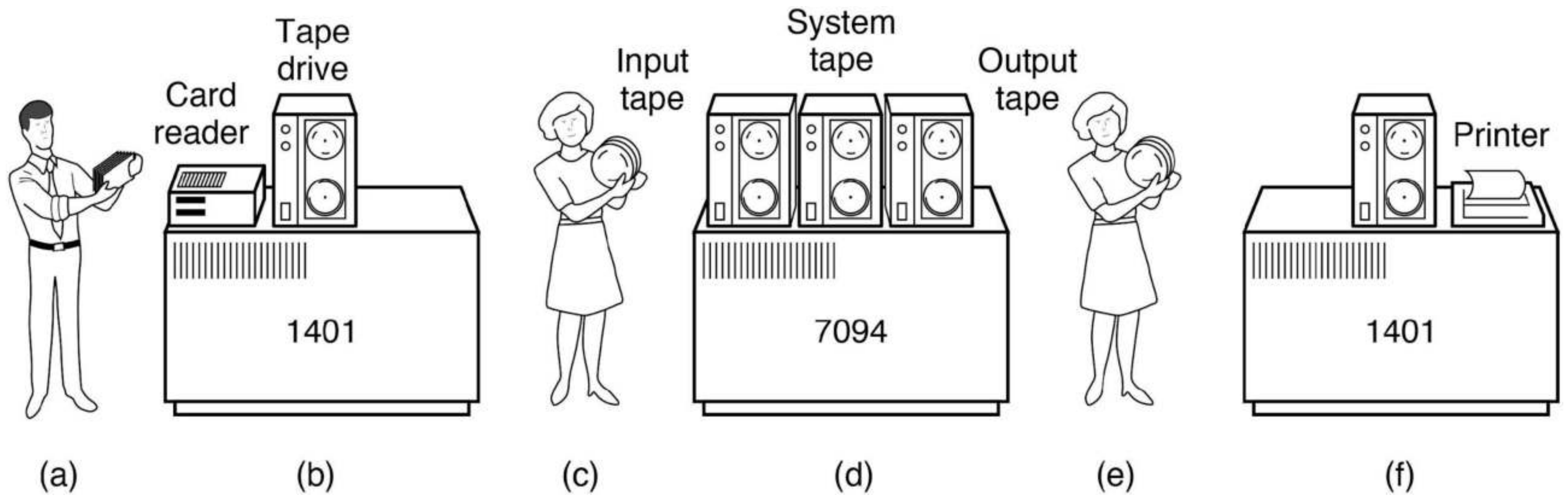


Figure 1-3. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape…

Second Generation: Transistors and Batch Systems
(1 of 3)

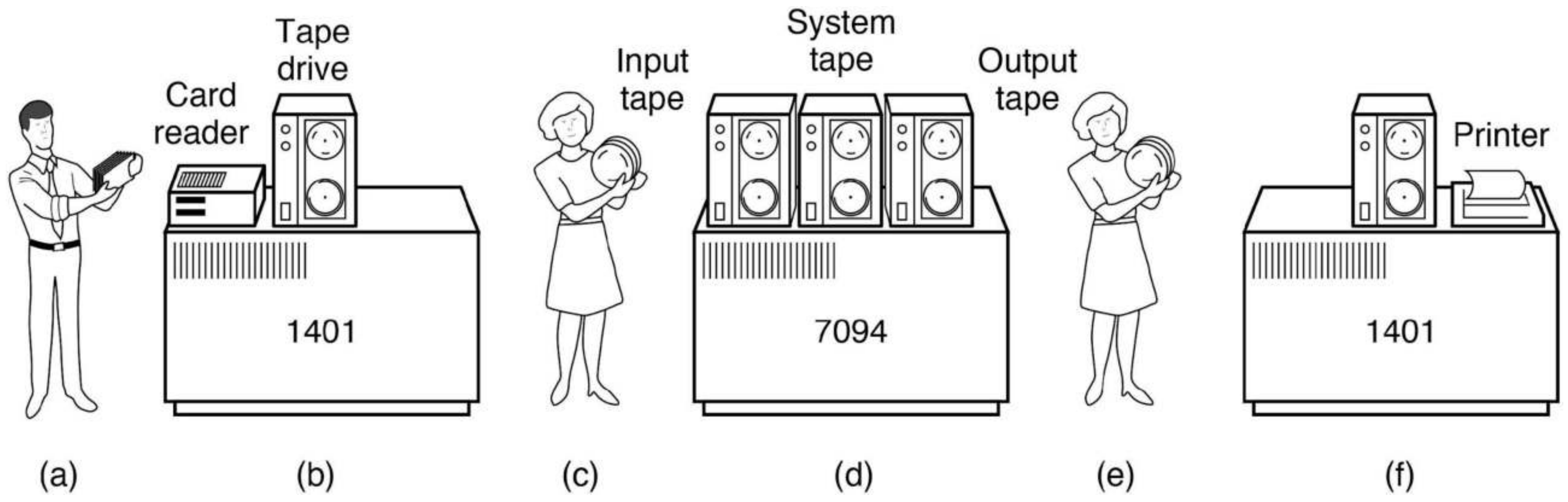


Figure 1-3. An early batch system. … (c) Operator carries input tape to 7094. (d)7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

Second Generation: Transistors and Batch Systems
(3 of 3)

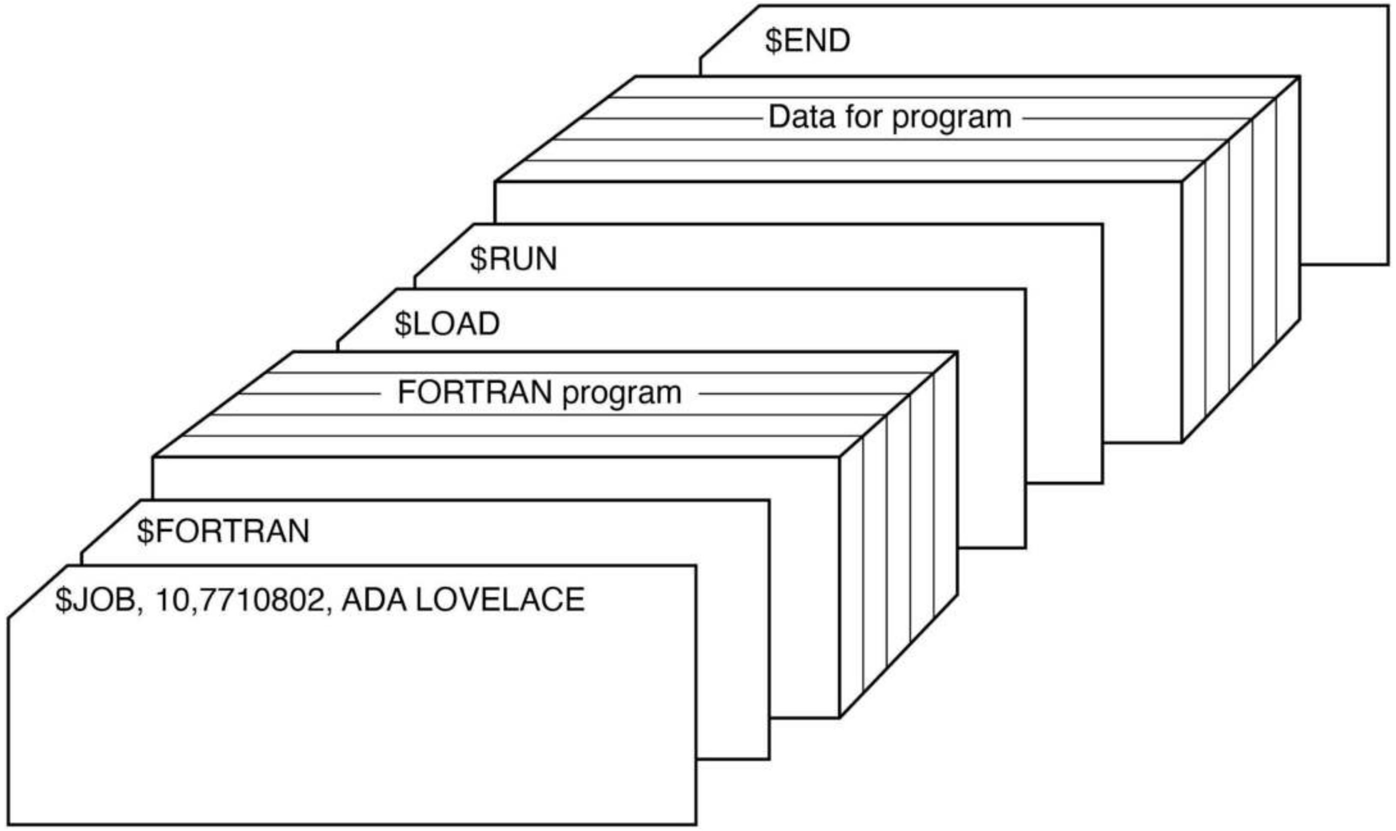


Figure 1-4. Structure of a typical FMS job.

Third Generation: ICs and Multiprogramming

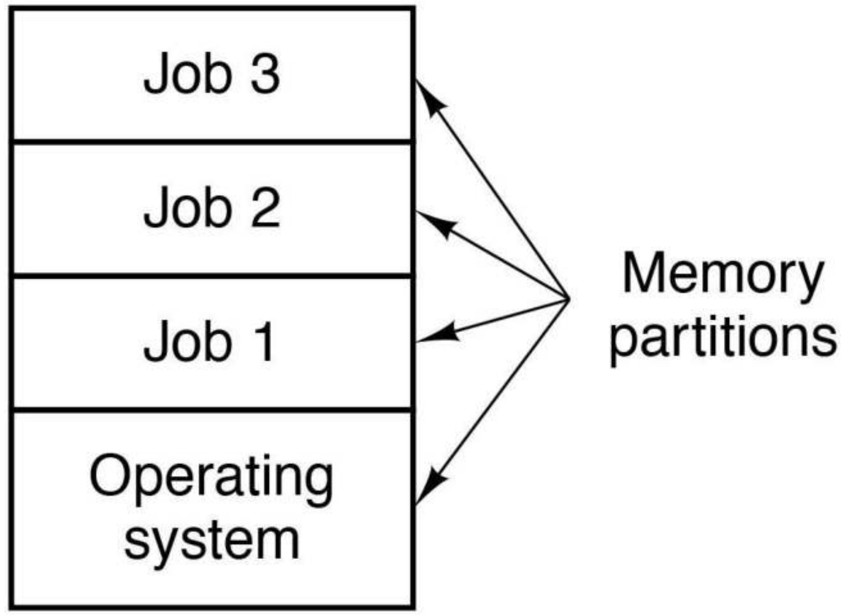


Figure 1-5. A multiprogramming system with three jobs in memory.

Timesharing

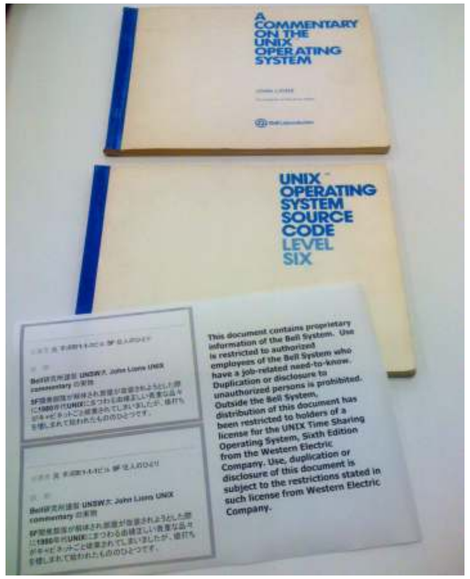
Timesharing and Multics

"*I remarked to Dennis that easily half the code I was writing in Multics was error recovery code*".

— Tom Van Vleck, 1973

Multics introduced many novel features that we find in OSs today. It was also complex.

UNIX – a simpler operating system

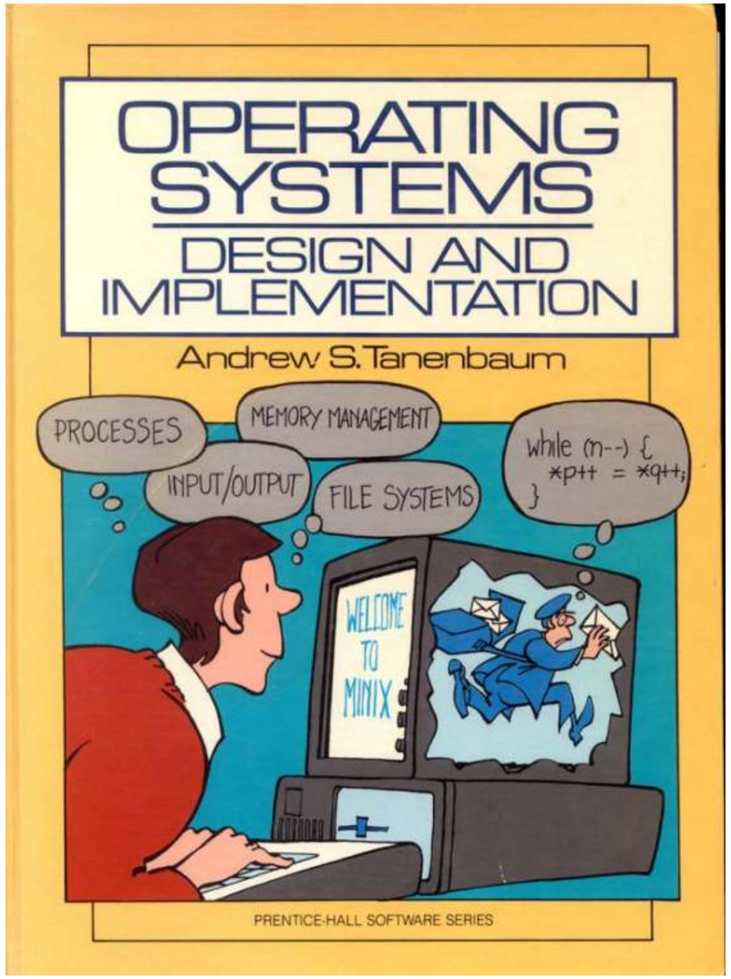

A COMMENTARY ON THE UNIX OPERATING SYSTEM
UNIX OPERATING SYSTEM SOURCE CODE LEVEL SIX

This document contains proprietary information of the Bell System. Use is restricted to authorized employees of the Bell System who have a job-related need-to-know. Duplication or disclosure to unauthorized persons is prohibited. Outside the Bell System, distribution of this document has been restricted to holders of a license for the UNIX Time Sharing Operating System, Sixth Edition from the Western Electric Company. Use, duplication or disclosure of this document is subject to the restrictions stated in such license from Western Electric Company.

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

Comment by Dennis Ritchie in the source code of UNIXv6, 1975. Aka 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)


OPERATING SYSTEMS
DESIGN AND IMPLEMENTATION
Andrew S. Tanenbaum
PROCESSES
MEMORY MANAGEMENT
INPUT/OUTPUT
FILE SYSTEMS
while (n--) {
*p++ = *q++;
}
WELCOME TO MINIX
PRENTICE-HALL SOFTWARE SERIES

Linux (1990s). Source: *comp.os.minix*

"*Hello everybody out there using minix -*

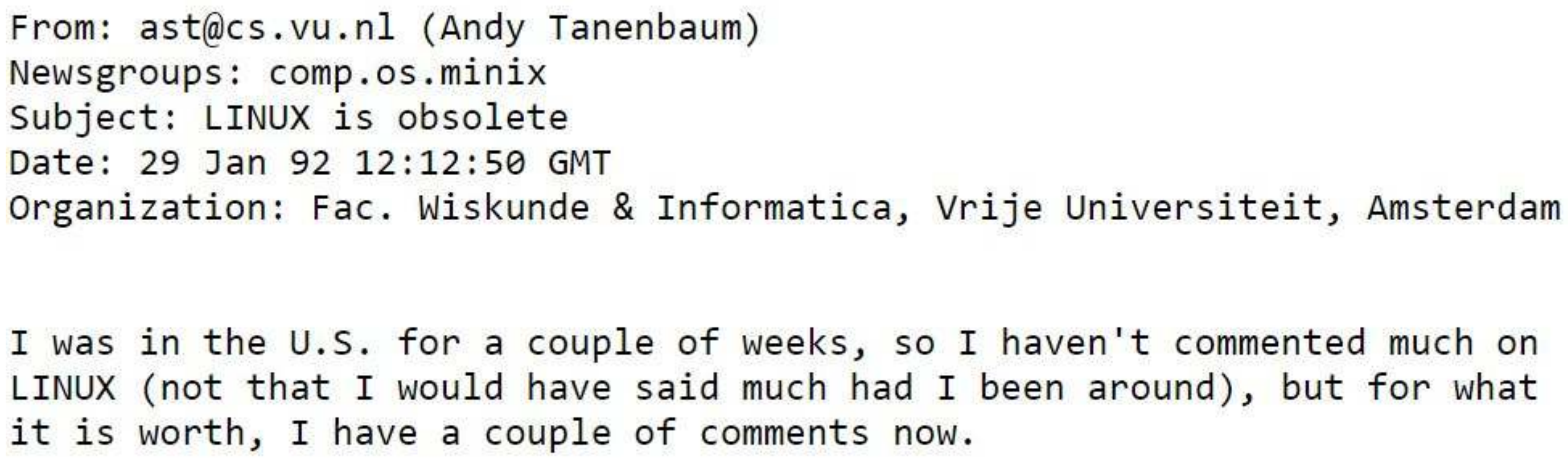
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the 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 protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(".

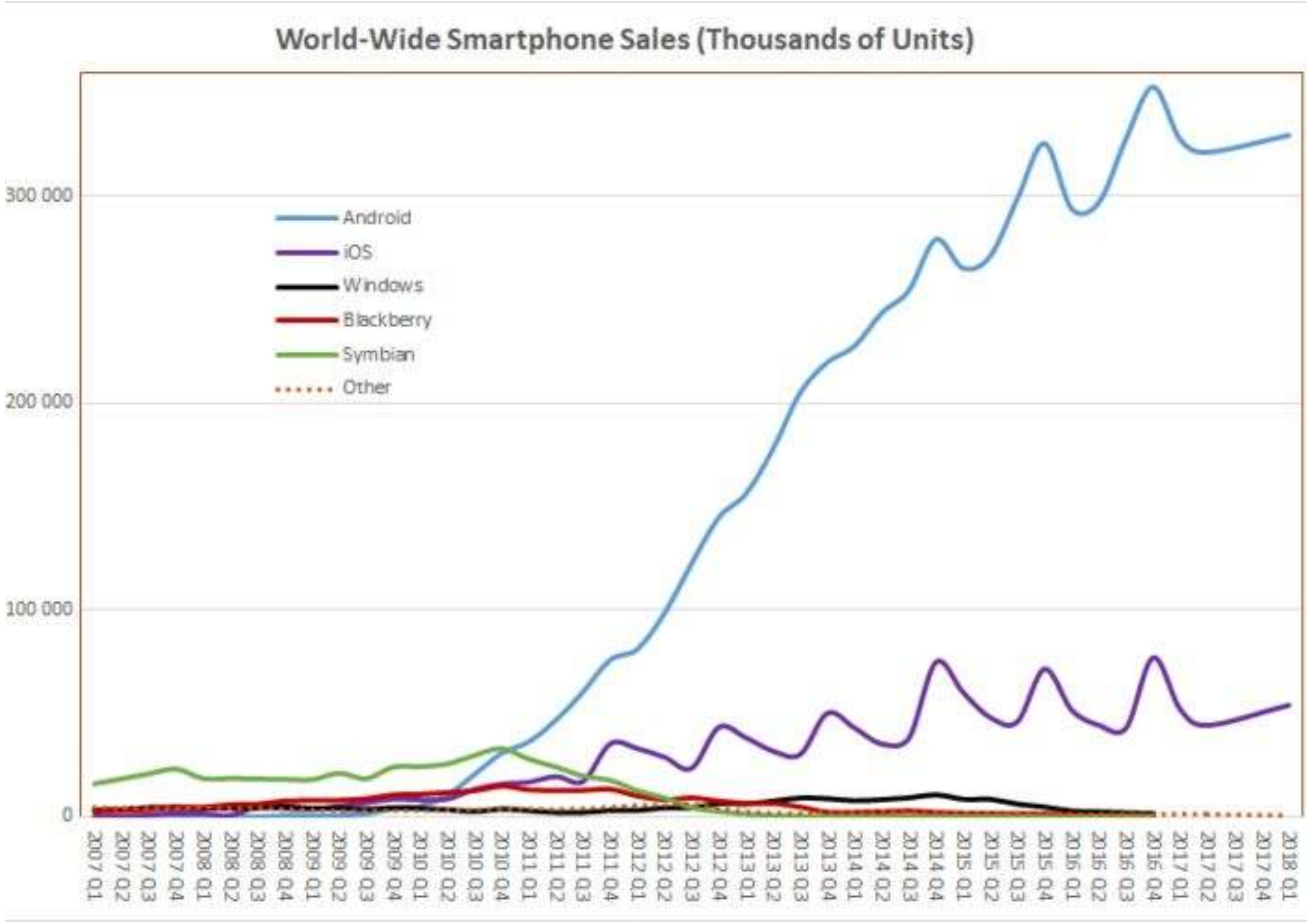
“Linux is obsolete”



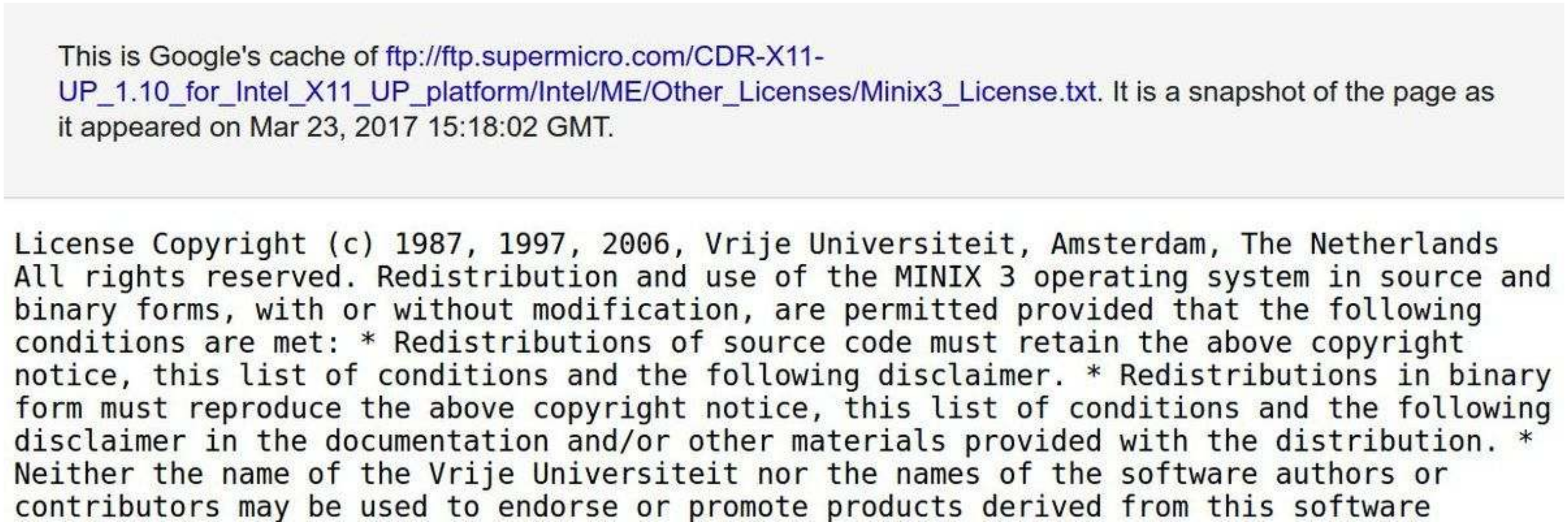
```
From: ast@cs.vu.nl (Andy Tanenbaum)
Newsgroups: comp.os.minix
Subject: LINUX is obsolete
Date: 29 Jan 92 12:12:50 GMT
Organization: Fac. Wiskunde & Informatica, Vrije Universiteit, Amsterdam


I was in the U.S. for a couple of weeks, so I haven't commented much on
LINUX (not that I would have said much had I been around), but for what
it is worth, I have a couple of comments now.
```

The exchange became known as the Tanenbaum – Torvalds debate. Its topic was the proper way to design an OS: using a small microkernel and all OS services in user space (Tanenbaum) or using a large monolithic kernel (Torvalds).

Since then, Linux (and Android) have thrived.



But so has MINIX



This is Google's cache of ftp://ftp.supermicro.com/CDR-X11-UP_1.10_for_Intel_X11_UP_platform/Intel/ME/Other_Licenses/Minix3_License.txt. It is a snapshot of the page as it appeared on Mar 23, 2017 15:18:02 GMT.

```
License Copyright (c) 1987, 1997, 2006, Vrije Universiteit, Amsterdam, The Netherlands
All rights reserved. Redistribution and use of the MINIX 3 operating system in source and
binary forms, with or without modification, are permitted provided that the following
conditions are met: * Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer. * Redistributions in binary
form must reproduce the above copyright notice, this list of conditions and the following
disclaimer in the documentation and/or other materials provided with the distribution. *
Neither the name of the Vrije Universiteit nor the names of the software authors or
contributors may be used to endorse or promote products derived from this software
```

MINIX 3 was adopted by Intel for its Management Engine and is now in desktops, servers & laptops

Hardware

Processors (1 of 2)

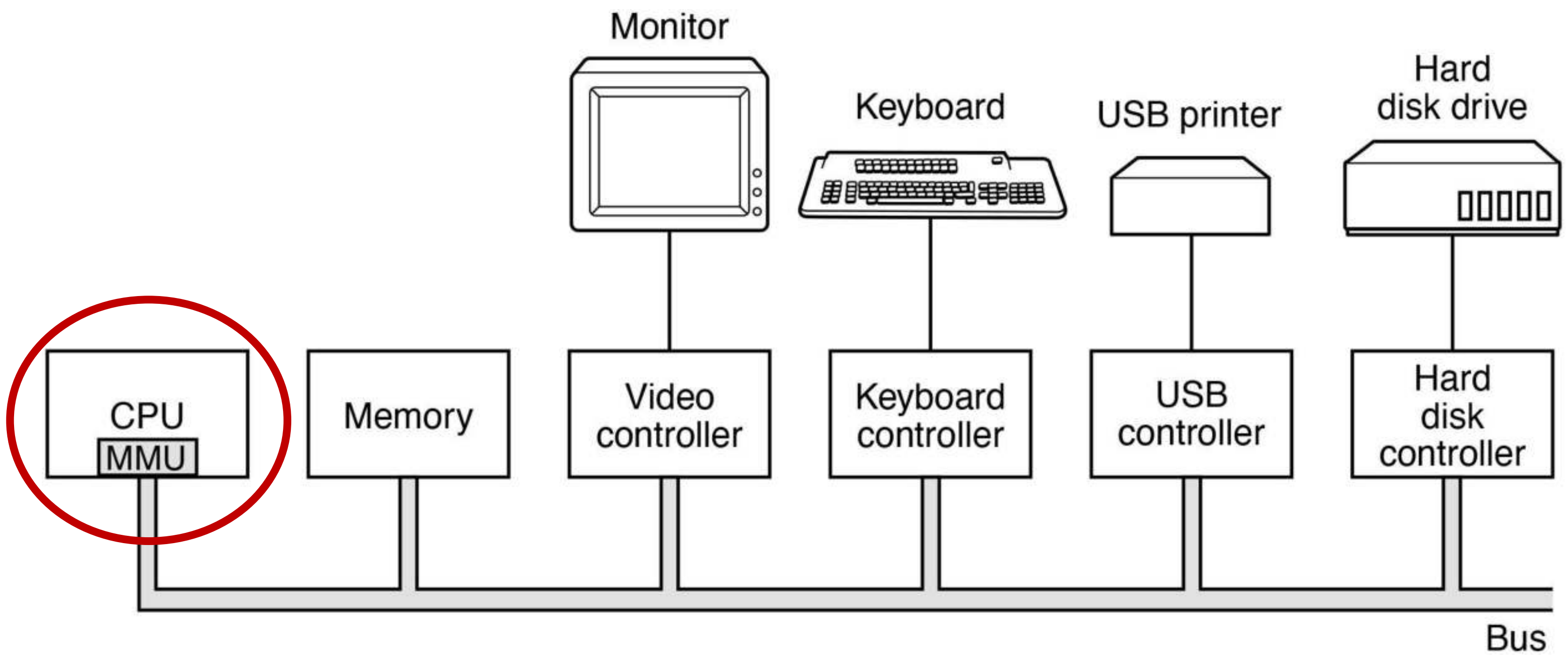


Figure 1-6. Some of the components of a simple personal computer.

Processors (2 of 2)

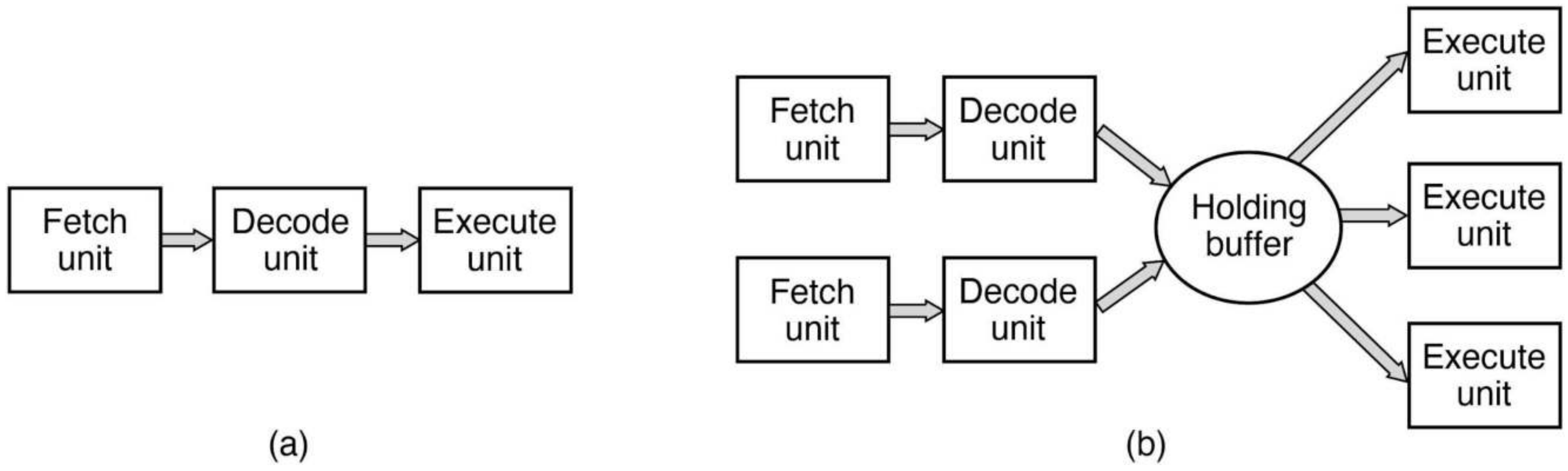


Figure 1-7. (a) A three-stage pipeline. (b) A superscalar CPU.

Memory (1 of 3)

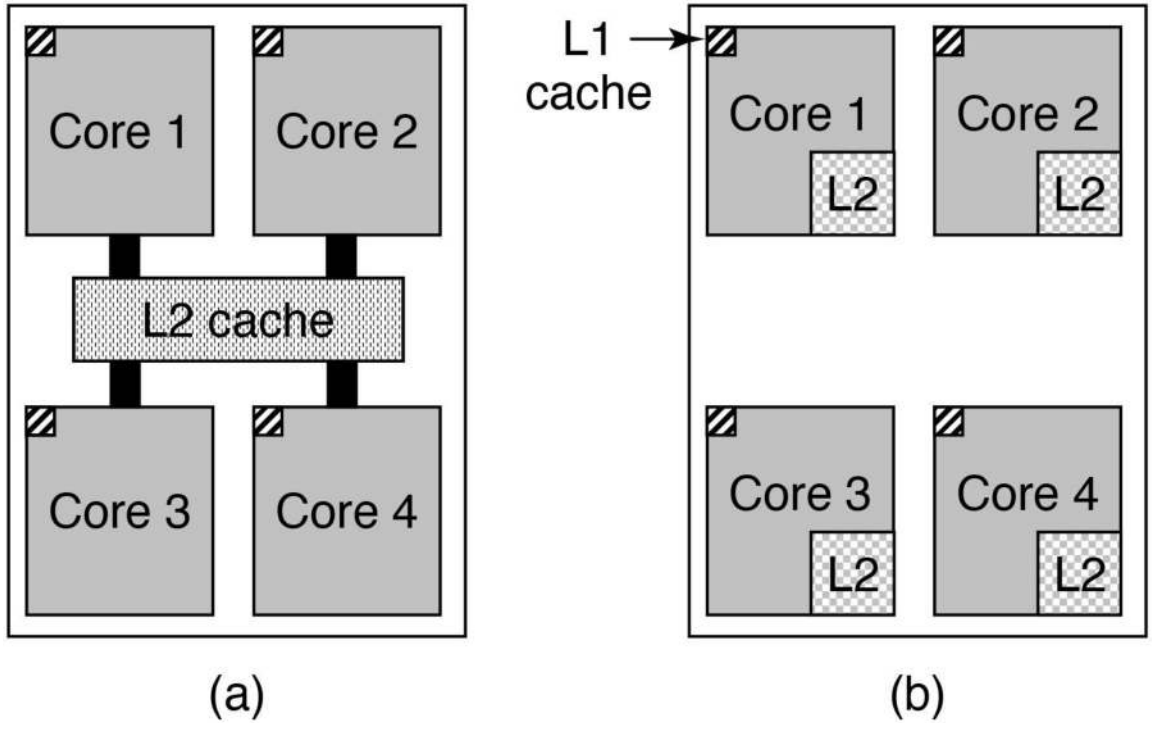


Figure 1-8. (a) A quad-core chip with a shared L2 cache. (b) A quad-core chip with separate L2 caches.

Memory (2 of 3)

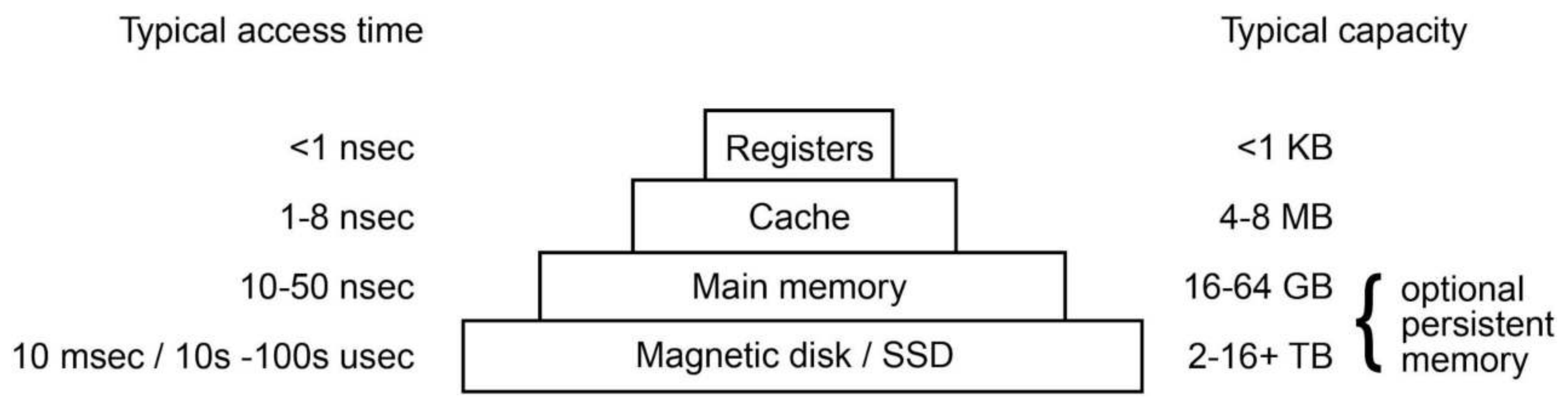
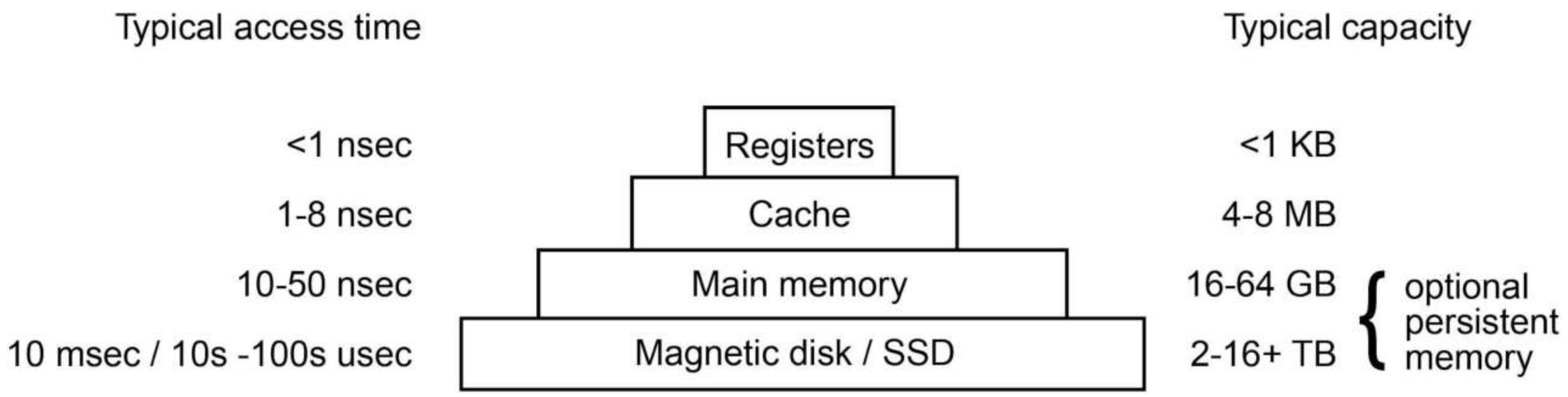


Figure 1-9. A typical memory hierarchy. The numbers are very rough approximations.

Memory (3 of 3)

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

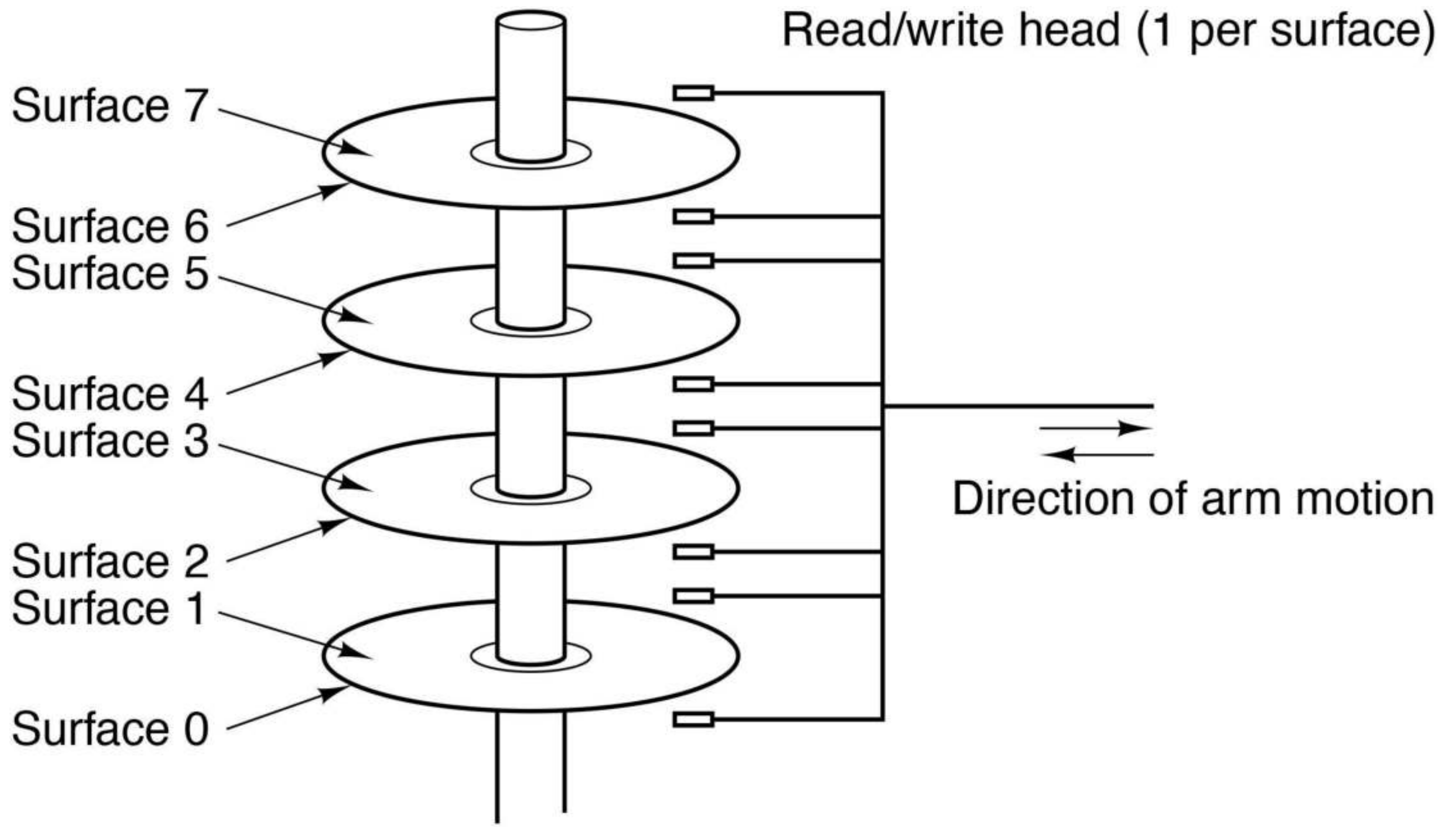


Figure 1-10. Structure of a disk drive.

Solid State Drives (SSD)

Often (incorrectly) referred to as disk also.

No moving parts, data in electronic(flash) memory.

Much faster than magnetic disks.

We will discuss SSDs in more detail later.

I/O devices (1 of 5)

Many types of I/O device.

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 (2 of 5)

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 (3 of 5) - Performing I/O

To perform I/O:

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

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

  or asks device generate an interrupt when it is done (and returns)

  more advanced: make use of special hardware – DMA (later)

I/O Devices (4 of 5) - Interrupts

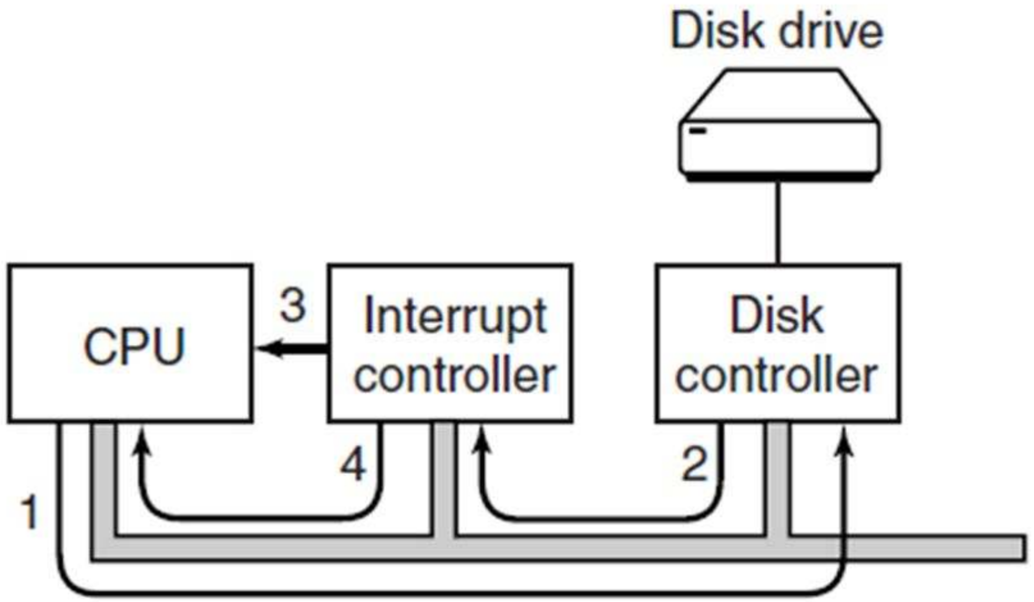


Figure 1-11. (a) The steps in starting an I/O device and getting an interrupt.

I/O Devices (5 of 5) - Interrupts

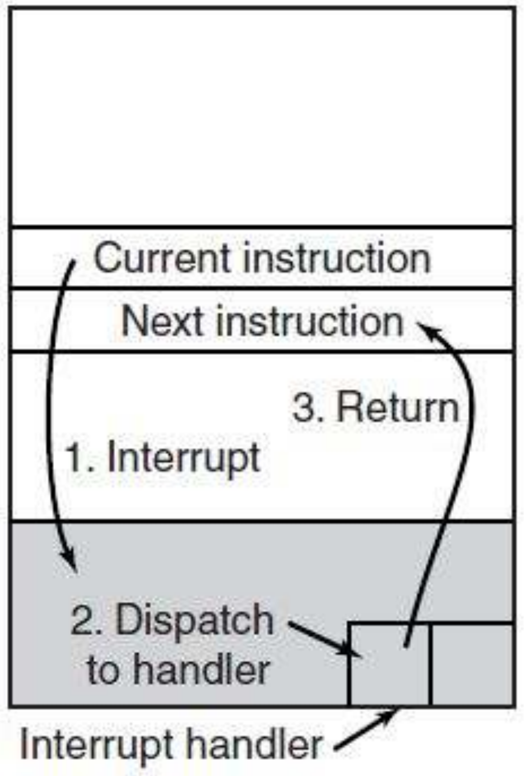


Figure 1-11. (b) Interrupt processing involves taking the interrupt, running the interrupt handler, and returning to the user program.

Buses

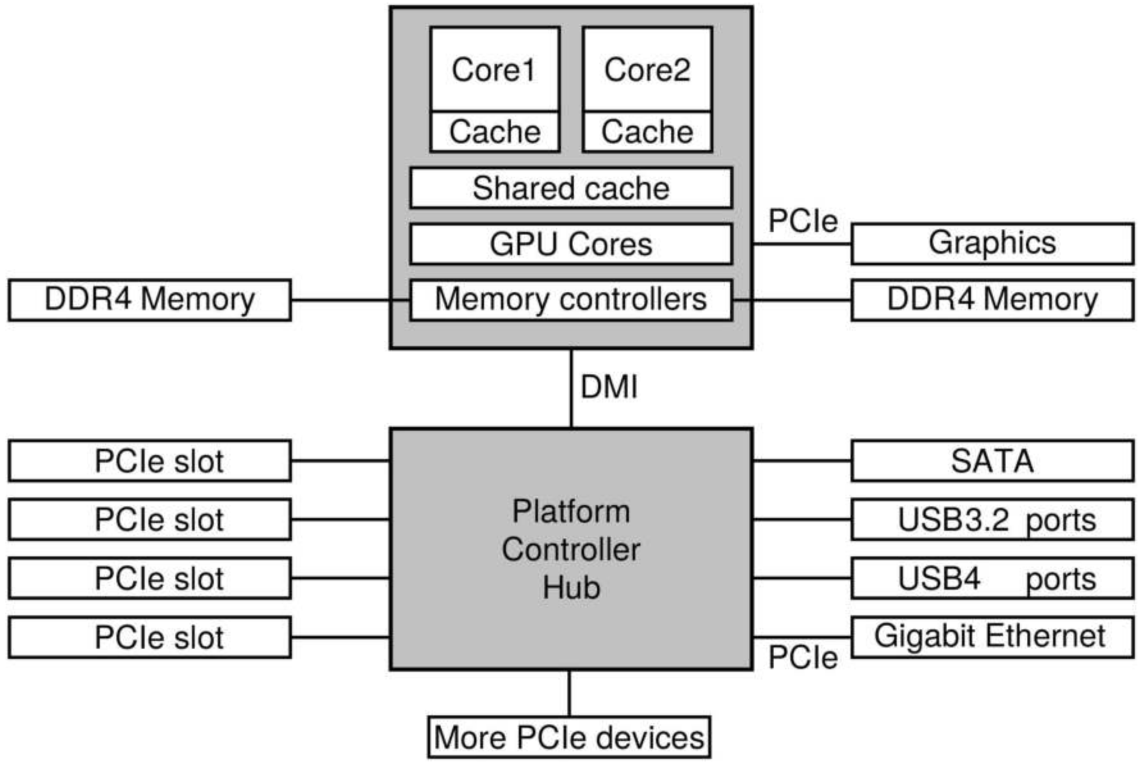


Figure 1-12. The structure of a large x86 system: many buses (e.g., cache, memory, PCIe, USB, SATA, and DMI)

Booting (UEFI)

Flash memory on motherboard contains firmware (aka 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

BIOS can read simple file systems (e.g., FAT-32), and starts first bootloader program (from partition indicated by UEFI boot manager)

- The bootloader may load other bootloader programs

Eventually the OS is loaded.

The Operating System Zoo

- 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

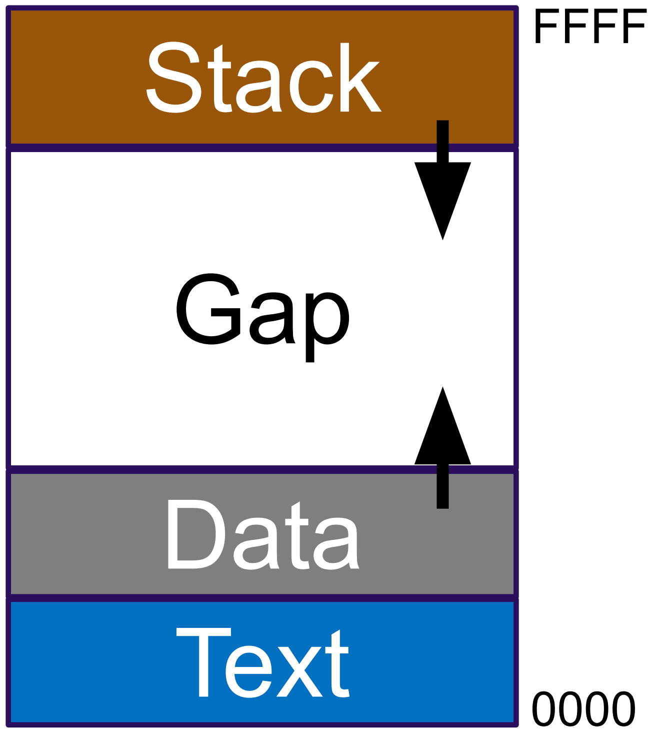
- OS offers functionality through **system calls**
- Groups of system calls implement **services**
  - File System Service
  - Process Management Service
- **Processes** are user-level abstractions to execute a program on behalf of a user
- Each process has its own **address space**
- Data involved in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

Processes (1 of 5)

- 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 (2 of 5) - Address Space

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



Processes (3 of 5)

Information about process is kept in the OS’ process table.

- A suspended process consists of process table entry (saved registers and other info needed to restart the process) *and* its address space.

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 (3 of 5)

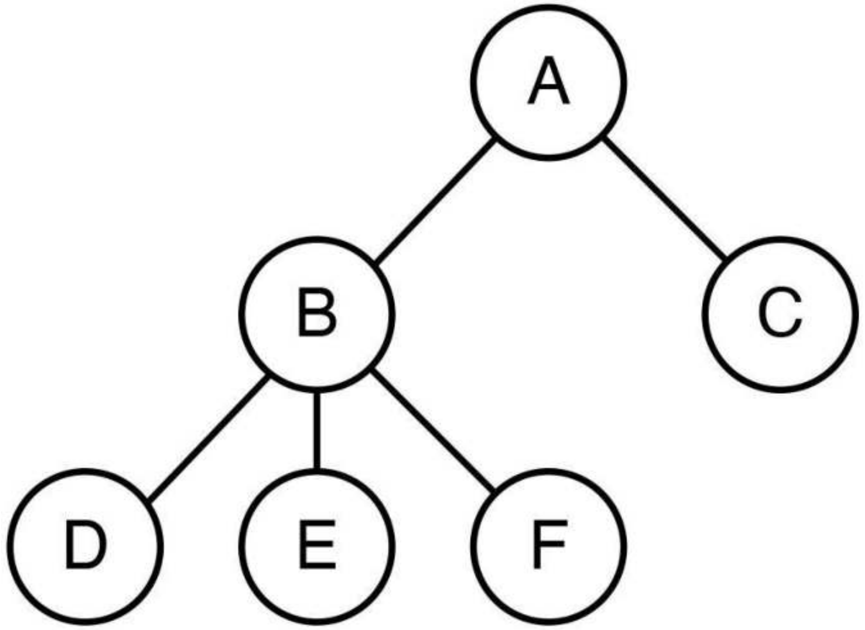


Figure 1-13. A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

Processes (4 of 5)

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

One process (superuser/root/administrator) is special: has more privileges

Files (1 of 9)

- 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 (2 of 9)

- 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 (3 of 9)

- 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 mygroup 14492 Dec 4 18:04 myfile`
  - 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 (4 of 9)

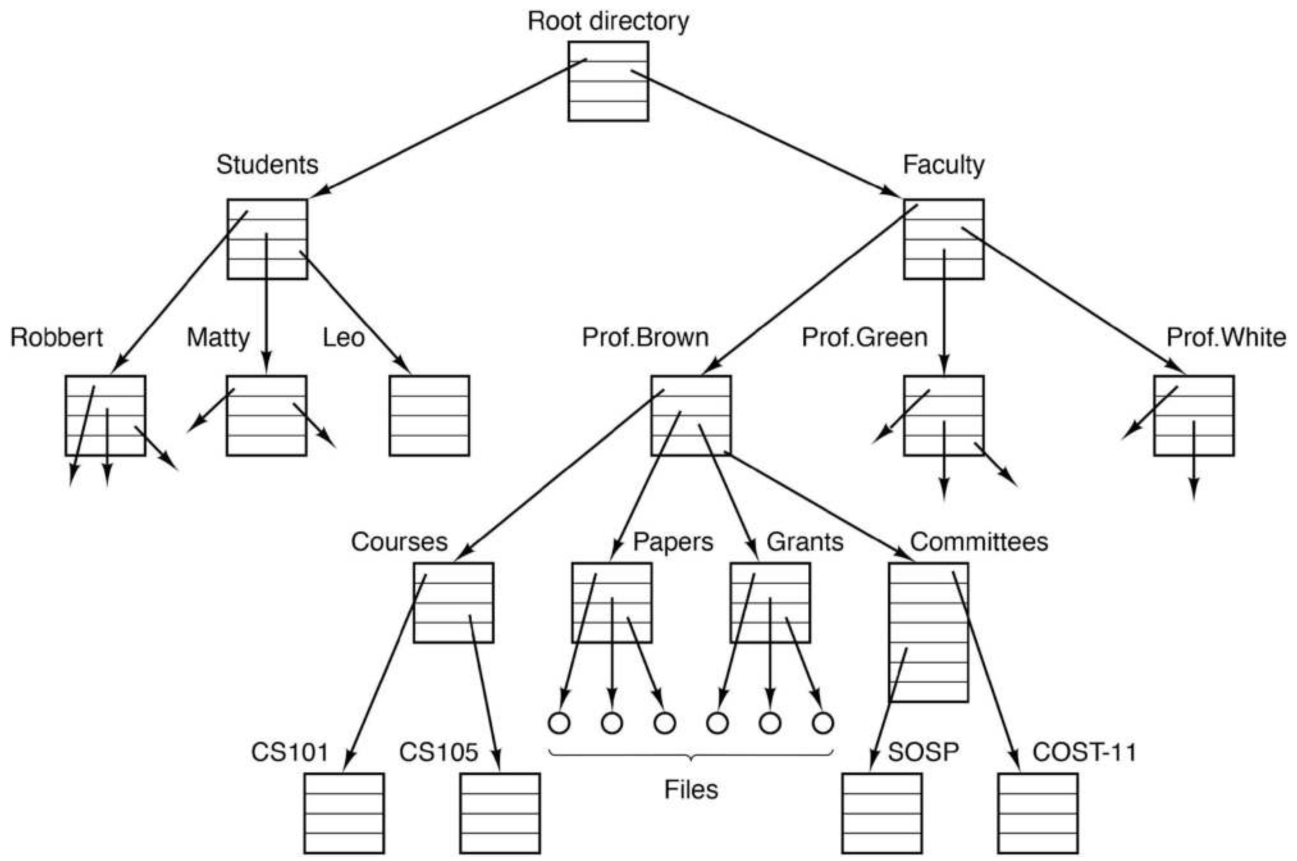


Figure 1-14. A file system for a university department.

Files (5 of 9)

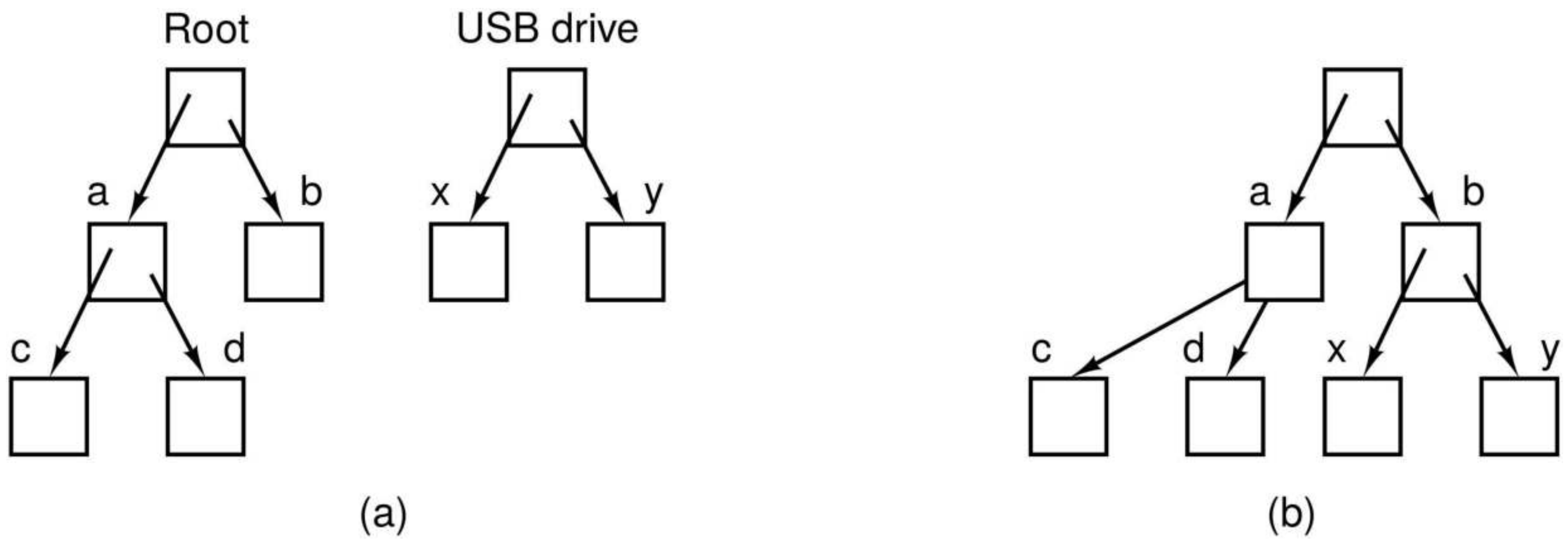


Figure 1-15. (a) Before mounting, the files on the USB drive are not accessible. (b) After mounting, they are part of the file hierarchy.

Files (6 of 9)

Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
Herb ertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os            ← What happens here?
herbertb@sleet:~/tmp$ cat > os/qq.txt         ← Will this work?
herbertb@sleet:~/tmp$ ls os/                  ← Will this work?
herbertb@sleet:~/tmp$ os/hello.sh             ← Will this work?
herbertb@sleet:~/tmp$ rm os/hello.sh          ← Will this work?
```

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ cat > os/qq.txt
```

← What happens here?

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ cat > os/qq.txt
-bash: os/qq.txt: Permission denied
herbertb@sleet:~/tmp$
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ cat os/hello.sh
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ cat os/hello.sh
-bash: os/hello.sh: Permission denied
herbertb@sleet:~/tmp$
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ ls os/
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ ls os/
ls: cannot access 'os/hello.sh': Permission denied
hello.sh
herbertb@sleet:~/tmp$
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ ls -l os/
ls: cannot access 'os/hello.sh': Permission denied
total 0
-????????? ? ? ? ?            ? hello.sh
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ os/hello.sh
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ pwd
/home/herbertb/tmp
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ os/hello.sh
hello world!
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ os/hello.sh
-bash: os/hello.sh: Permission denied
herbertb@sleet:~/tmp$
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 444 os/hello.sh
herbertb@sleet:~/tmp$ chmod 744 os
herbertb@sleet:~/tmp$ rm os/hello.sh
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 444 os/hello.sh
herbertb@sleet:~/tmp$ chmod 744 os
herbertb@sleet:~/tmp$ rm os/hello.sh                     ← Will this work?
rm: remove write-protected regular file 'os/hello.sh'? y
herbertb@sleet:~/tmp$
```

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ rm os/hello.sh
```

← Will this work?

Quiz:
Next Slide

Permissions - Quiz

```
herbertb@sleet:~/tmp$ mkdir os
herbertb@sleet:~/tmp$ cat > os/hello.sh
#!/bin/sh
echo "hello world!"
herbertb@sleet:~/tmp$ chmod 744 os/hello.sh
herbertb@sleet:~/tmp$ chmod 644 os
herbertb@sleet:~/tmp$ rm os/hello.sh
rm: cannot remove 'os/hello.sh': Permission denied
herbertb@sleet:~/tmp$
```

← Will this work?

Quiz:
Next Slide

Files (7 of 9)
Special Files

“Everything is a file”:

- Hardware devices are abstracted as files:
  - **Block special files**, e.g., disk:

```
brw-rw---- 1 root root 8, 2 Dec 4 18:04 /dev/sda2
```

  - **Character special files**, e.g., serial port:

```
crw-rw---- 1 root root 4,64 Dec 4 18:04 /dev/ttyS0
```

- Other special files:
  - symbolic links
  - named/anonymous FIFOs (sockets/pipes)

“Everything is a file *descriptor*”

Files (8 of 9)

- Pipes: pseudo files allowing for multiple 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

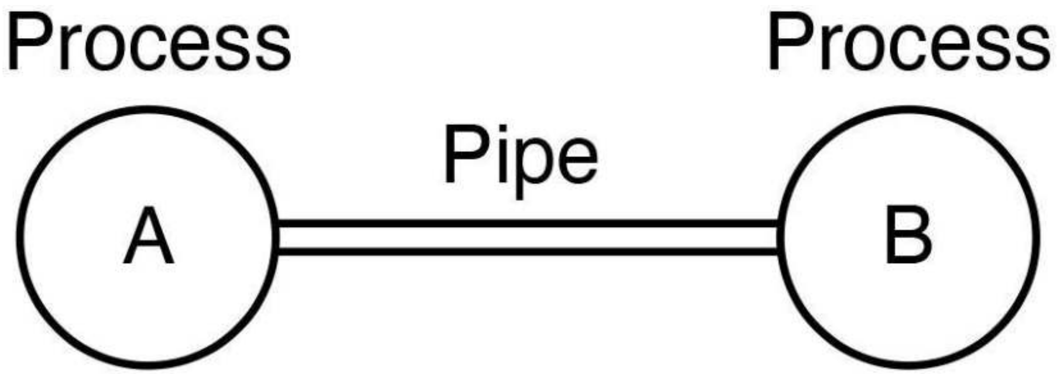


Figure 1-16. Two processes connected by a pipe.

Files (9 of 9)

Terminology

Important terms

- Path
- Folder / map / directory
- Root directory
- Working directory
- File descriptor
- Mounting
- Block/character special files
- Pipe

Ontogeny Recapitulates Phylogeny

- Each new “species” of computer
  - Goes through same development as “ancestors”
- Consequence of impermanence
  - Text often looks at “obsolete” concepts
  - Changes in technology may bring them back
- Happens with large memory, protection hardware, disks, virtual memory

System calls

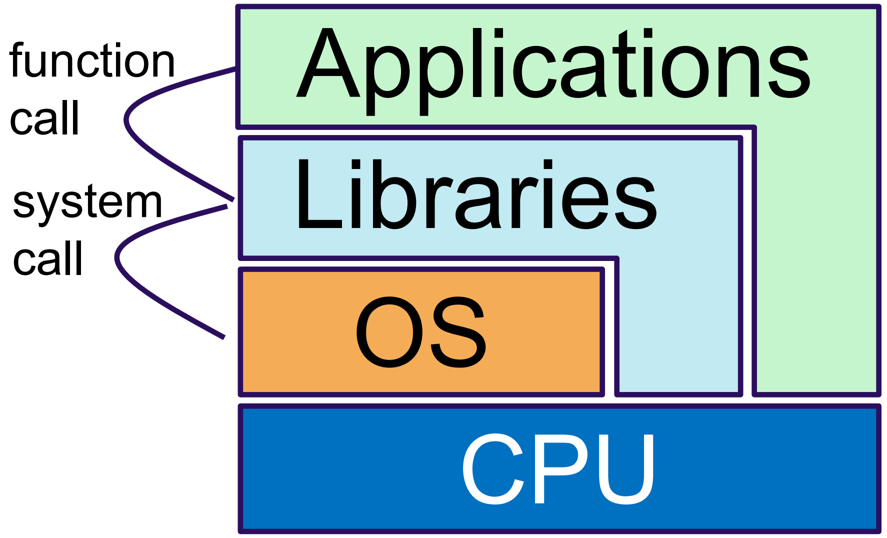
System calls are the interface the OS offers to applications to issue service requests

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

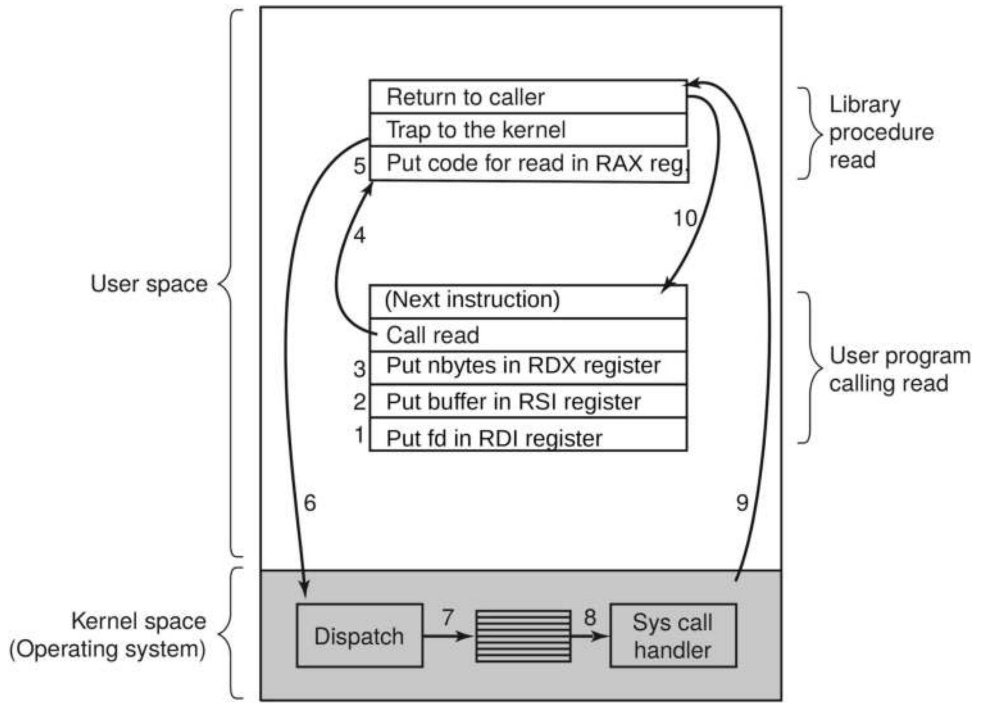


Figure 1-17. The 10 steps in making the system call **read(fd, buffer, nbytes).**

System Calls

Figure 1-18. 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.

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

System Calls

Figure 1-18. 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.

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 = Iseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

System Calls

Figure 1-18. 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.

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

System Calls

Figure 1-18. 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.

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
s= time(&seconds)	Get the elapsed time since Jan. 1, 1970

System Calls

```
#define TRUE 1

while (TRUE) {                                      /* repeat forever */
     type_prompt( );                                /* display prompt on the screen */
     read_command(command, parameters);             /* read input from terminal */

     if (fork( ) != 0) {                            /* fork off child process */
         /* Parent code. */
         waitpid(-1, &status, 0);                   /* wait for child to exit */
     } else {
         /* Child code. */
         execve(command, parameters, 0);            /* execute command */
     }
}
```

Figure 1-19. A stripped-down shell. Throughout this book, **TRUE** is assumed to be defined as 1.

System Calls for Process Management (6 of 6)

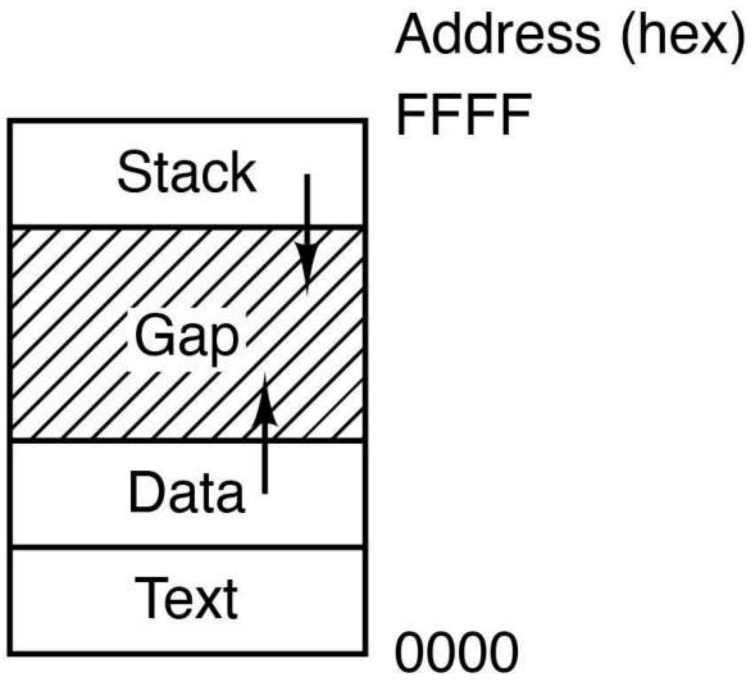


Figure 1-20. Processes have three segments: text, data, and stacks

System Calls for Directory Management (1 of 2)

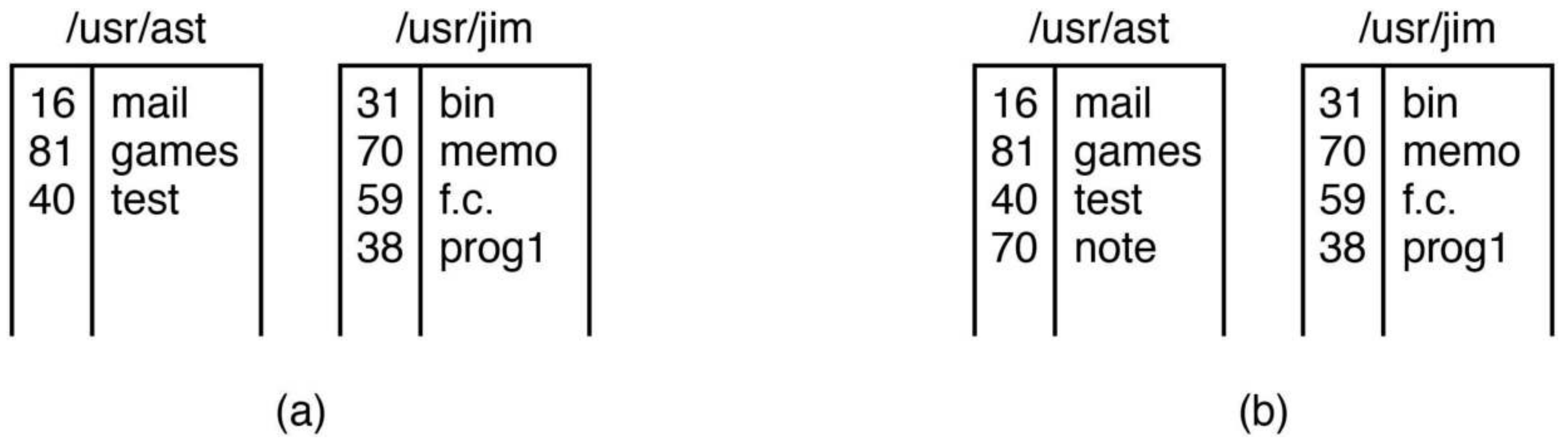


Figure 1-21. (a) Two directories before linking **usr/jim/memo to ast’s** directory. (b) The same directories after linking.

System Calls for Directory Management (2 of 2)

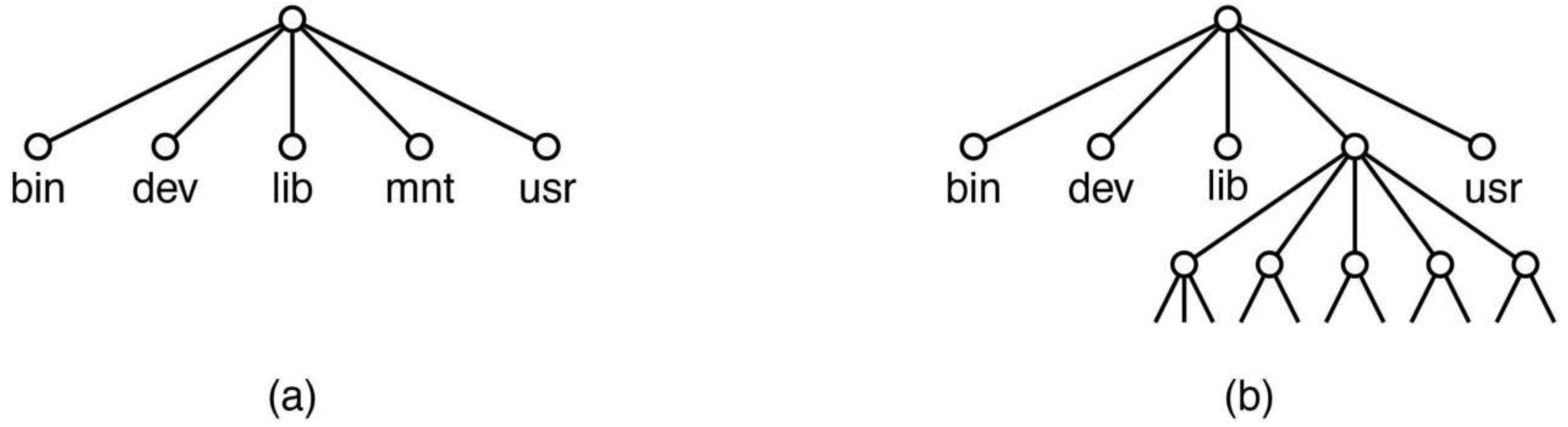


Figure 1-22. (a) File system before the mount. (b) File system after the mount.

The Windows API (1 of 2)

Figure 1-23. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18.

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 tile
Write	WriteFile	Write data to a file
I seek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory

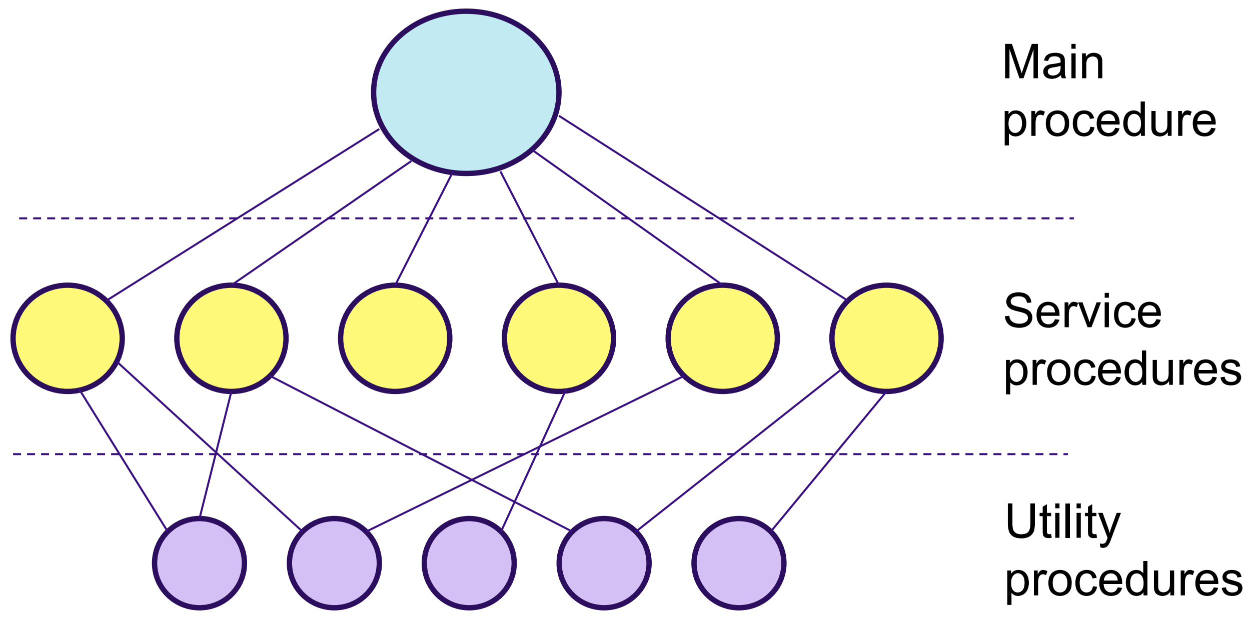
The Windows API (2 of 2)

Figure 1-23. The Win32 API calls that roughly correspond to the UNIX calls of Fig. 1-18.

Iseek	SetFilePointer	Move the tile pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdjr	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	GetLocamme	Get the current time

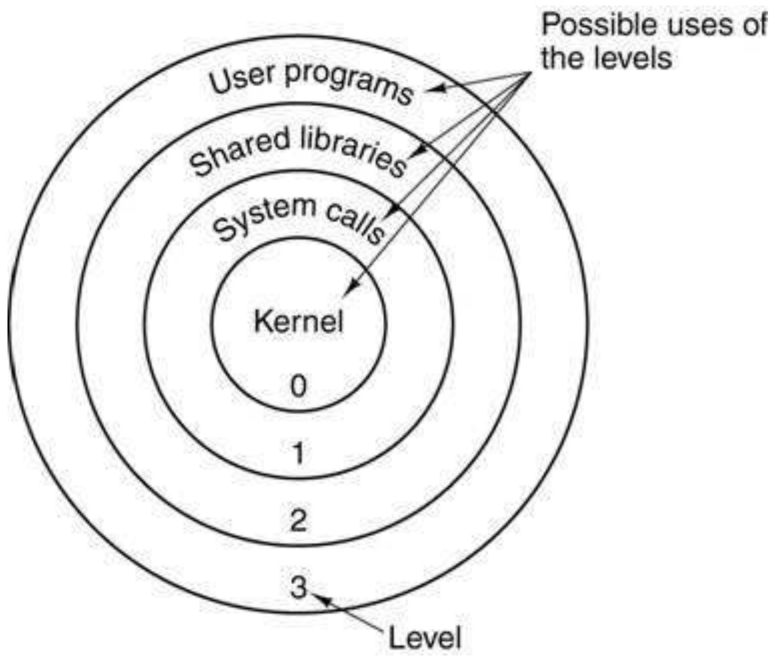
OS structure: Monolithic

- Main program invokes requested system calls
- Kernel is monolithic block with:
  - Service procedures that carry out the system calls
  - Utility procedures that help implement service procedures

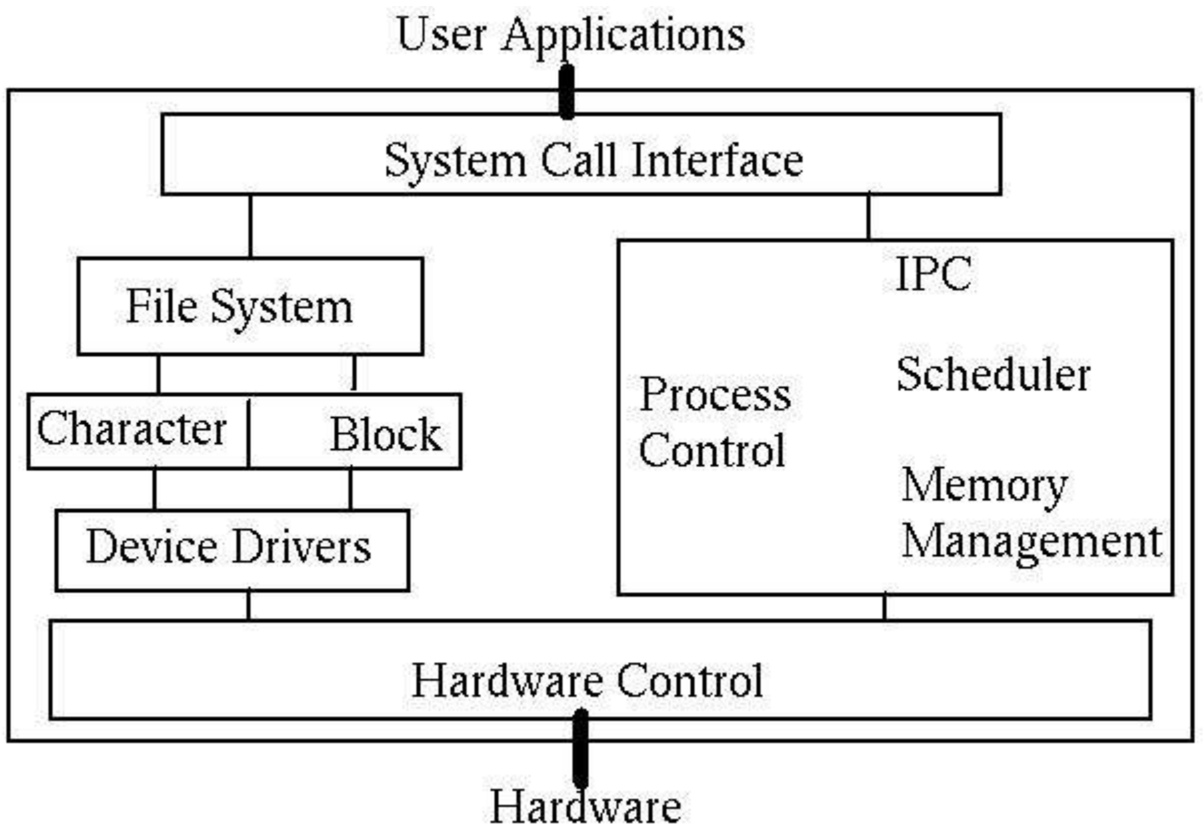


OS structure: Monolithic

- Monolithic kernels separate applications and OS using 2 privilege levels:
  - **User** (ring 3)
  - **Kernel** (ring 0)
- 4 **privilege levels** are available on x86
- Only early layered systems (e.g., Multics) used more than 2 levels in practice.



OS structure: Monolithic



Typical UNIX structure

OS structure: Virtualization

- Invented in the 70s to separate multiprogramming from extended machine
- Revamped today in several domains
- *N* independent OS system call interfaces

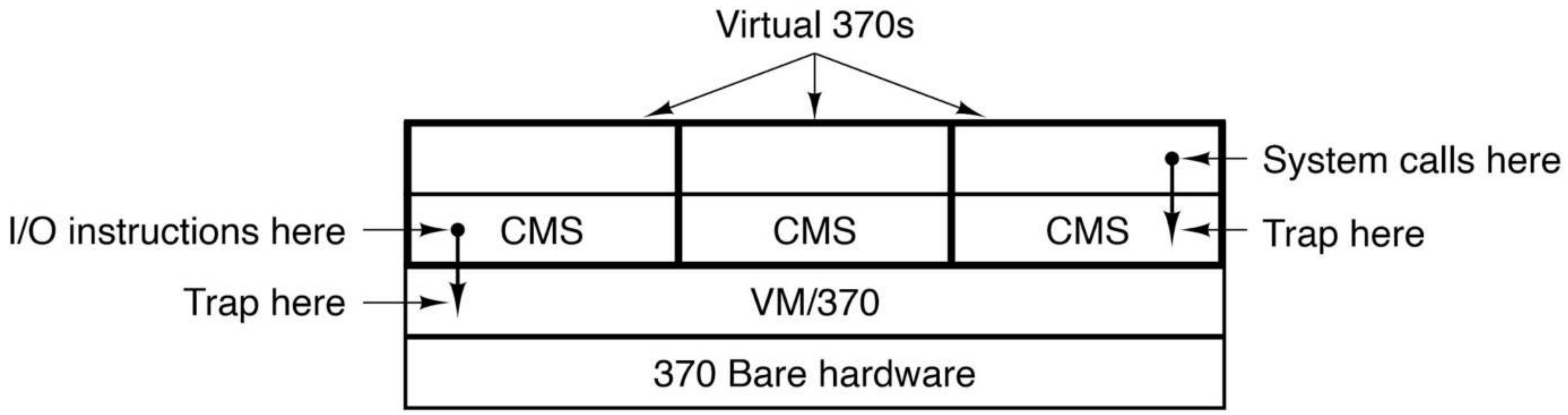


Figure 1-28. The structure of VM/370 with CMS.

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)

Alternative: para-virtualization (discussed later)

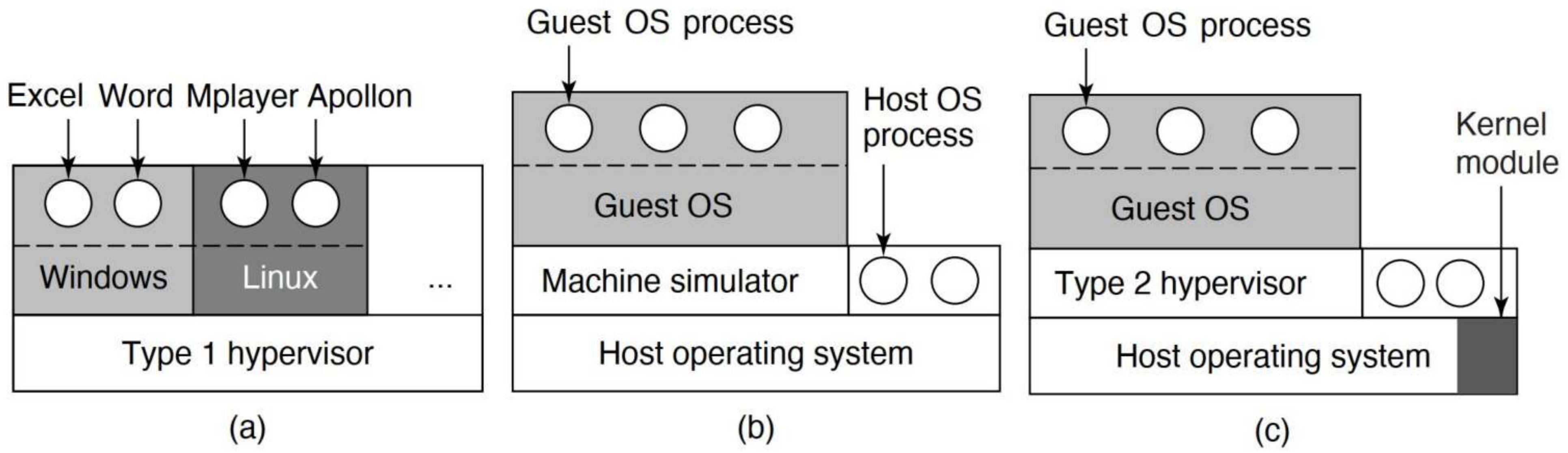


Figure 1-29. (a) A type 1 hypervisor. (b) A pure type 2 hypervisor. (c) A practical type 2 hypervisor

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

OS structure: Exokernel

- Idea: Separate resource control from extended machine
- Similar to a VMM/Hypervisor, but:
    - Exokernel does not emulate hardware
    - Provides only safe low-level resource sharing
    - Service procedures are offered as a library directly linked to the application → **Library OS**
    - System calls...?
- Different library OSes for different programs
    - Allows application-level OS specialization

OS structure: Unikernel

- Modern incarnation of LibOS
- Just enough functionality to support a single application (e.g., Web server)
- Often on top of VM
- Only one application on VM □ all code can run in kernel mode

OS structure: Microkernel-based Client/Server

- Organize service procedures in programs that run in separate processes → **System Servers/Drivers**
- System processes communicate via message passing
- System calls rely on the same messaging mechanism
- Messaging mechanism implemented in minimal kernel → **Microkernel**

OS structure: Microkernel-based Client/Server

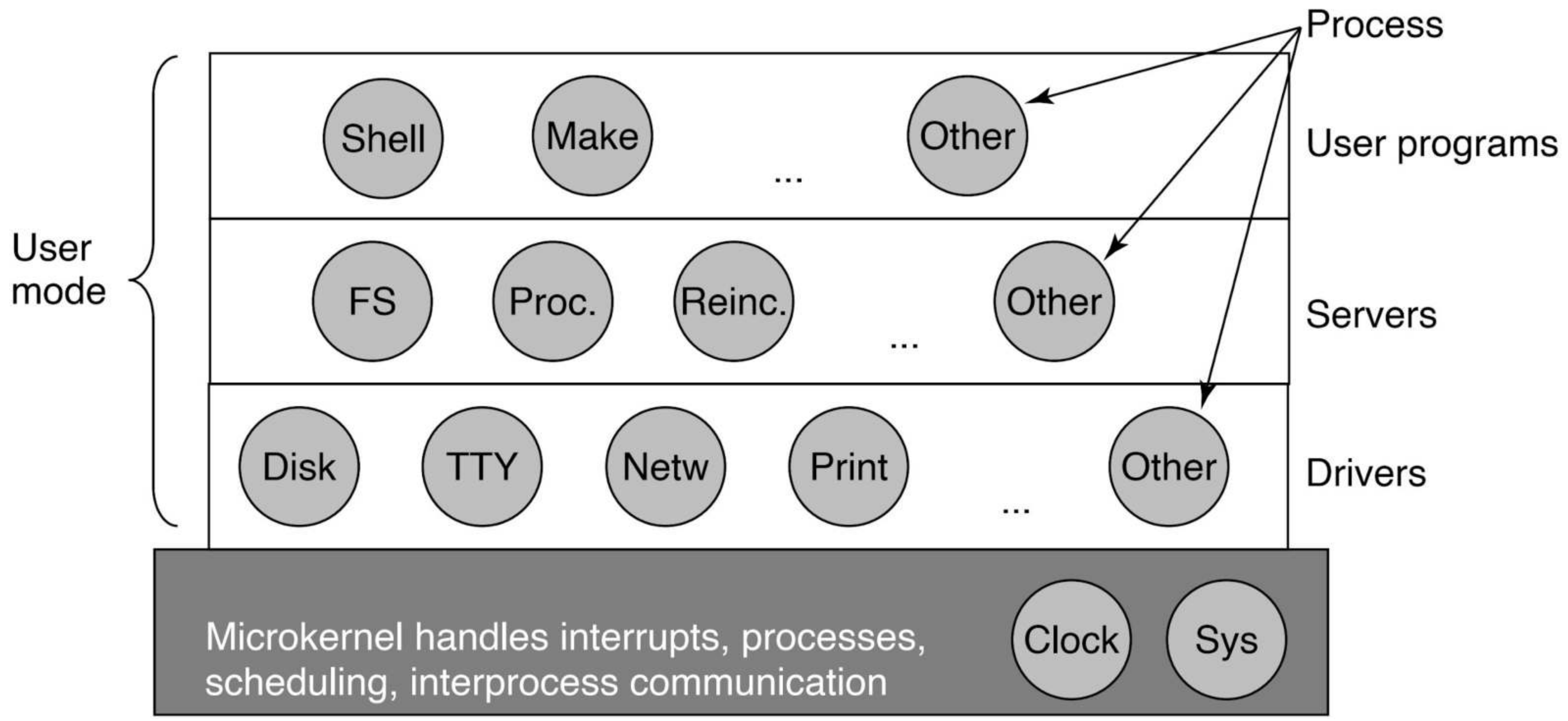


Figure 1-26. Simplified structure of the MINIX system

OS structure: Microkernel-based Client/Server

- Important advantage: easier to adhere to Principle of Least Authority (POLA):
    - Relatively small Trusted Computing Base (TCB)
    - Each OS process allowed to do only what is needed to perform its task
    - Compromise of, say, the printer driver will not affect rest of the OS
- Disadvantage
    - Message passing is slower than a function call (as in a monolithic kernel)

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: Everything is a file

Sockets

Devices

Hard drives

Printers

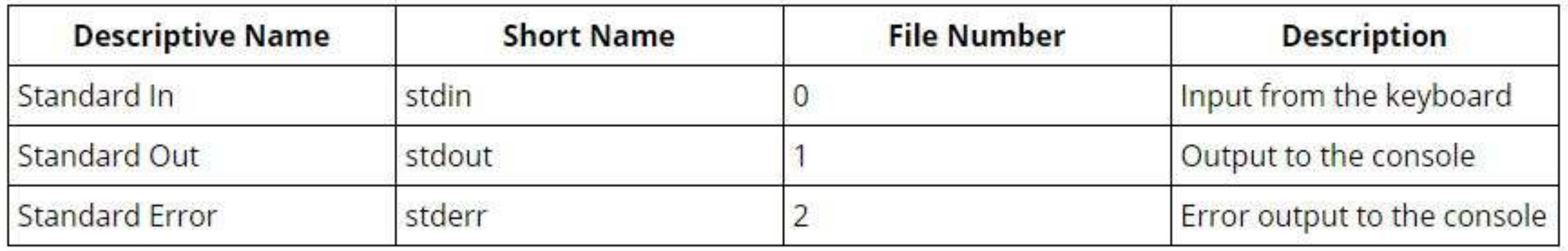
Modems

Pipes

...

C: Everything is a file

Default every process starts with 3 “files” opened



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

Hello World?

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

- `printf(char *str, …);`

Hello World!

```
#include <stdio.h>

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

Build process

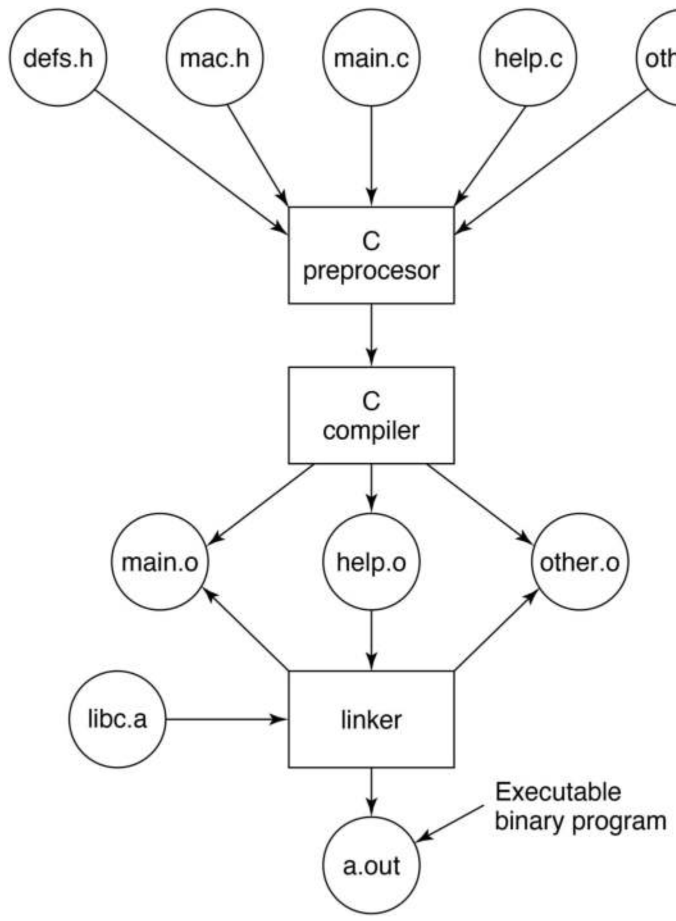


Figure 1-30. The process of compiling C and header files to make an executable binary program.

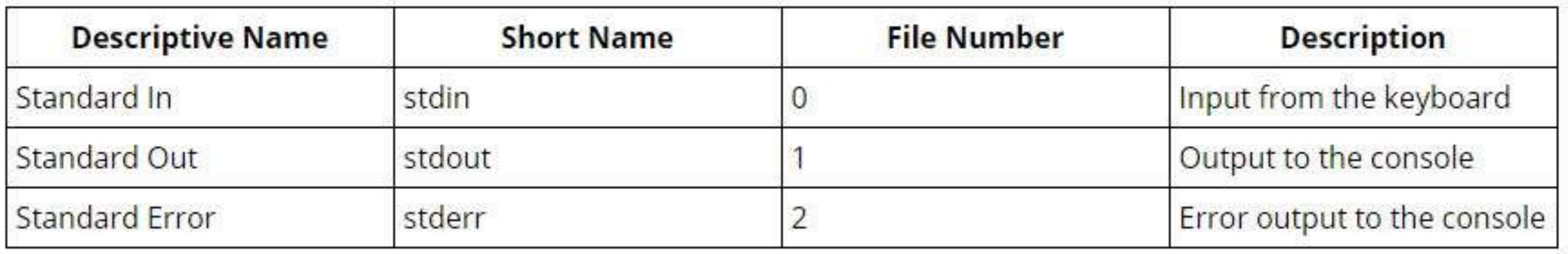
Hello World?

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!

Default every process starts with 3 “files” opened



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

Hello World!

```
#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?

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!

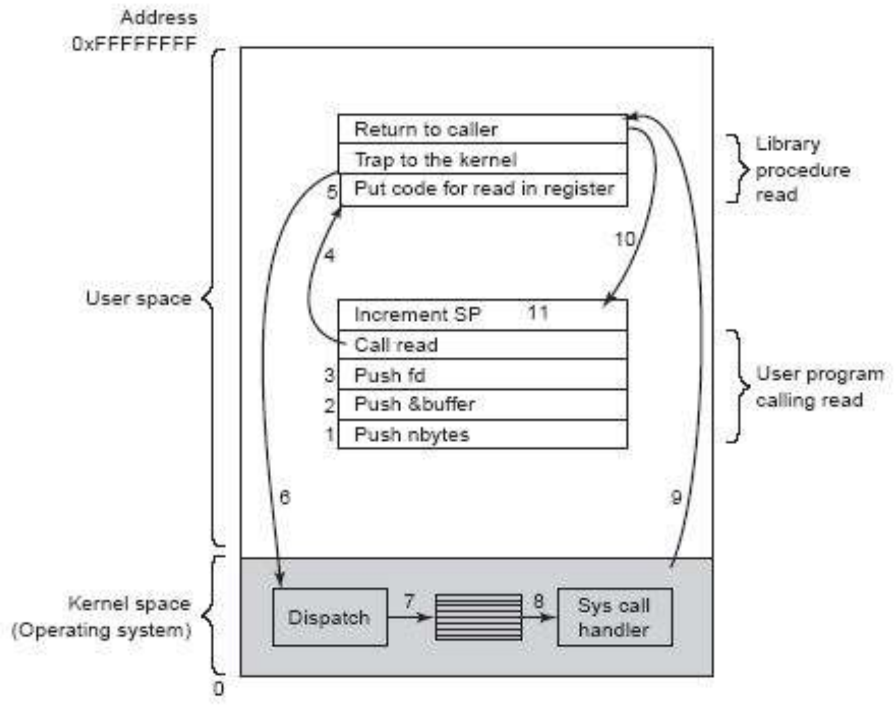
```
#define _GNU_SOURCE
#include <sys/syscall.h>
#define STDOUT 1

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

Standard 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, …)`

System call on x86: difference between 32b and 64b

System Calls: Internals (x86, 32b)



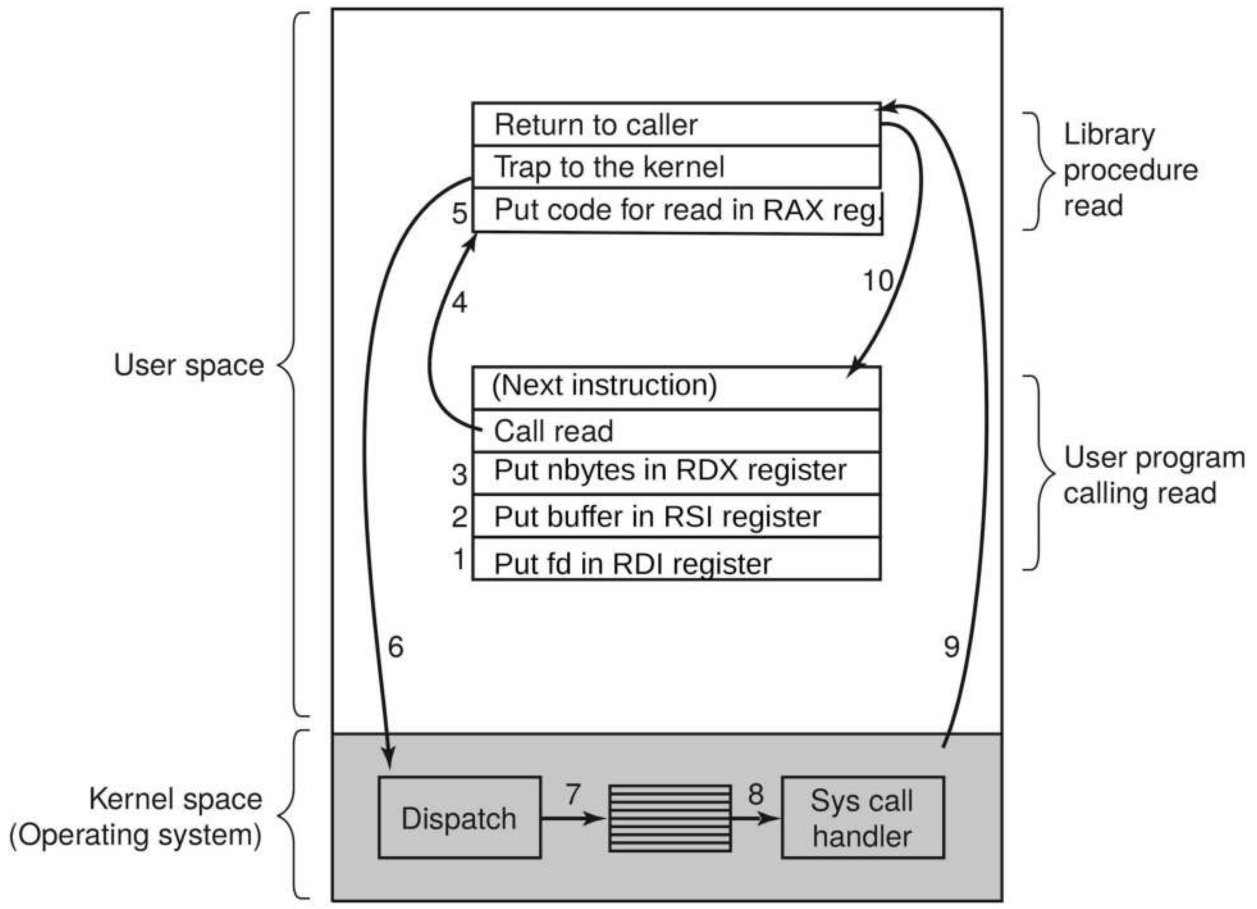
1-4: Prepare, and call library routine `read(fd,buffer,nbytes)`.
5-6: Prepare, and switch to kernel
7-9: Lookup & sys. handler, and return to user mode
10-11: Return to program and pop the stack

Linux System Calls (x86, 32b)

- System call is triggered by special instruction (e.g., *int 0x80*)
  - Privilege level is changed to kernel mode
  - Program counter is set to specific location
- Arguments for system call are passed in registers:
    - **`eax`** ← syscall number
    - other general purpose registers are used for arguments: `ebx, ecx, edx, esi, edi, ebp`
    - Additional arguments on stack

Linux System Calls (x86-64)

- Supports legacy x86-style **`int 0x80`**
  - Uses old calling convention
- New instruction: **`syscall`**
- Arguments for `syscall` passed in registers:
  - **`rax`** ← syscall number (different from 32-bit x86)
  - other general purpose registers used for arguments: `rdi, rsi, rdx, r10, r8, r9`



Hello World! (Linux, x86)

```
.text                                 # The tekst section contains the program *code*.

          .global _start              # Export entry point of program (linker/loader uses _start by convention).

_start:
                                      # To write helloworld string to stdout, arguments should be in registers
          movl    $len,%edx           # %edx register contains 3rd argument (length)
          movl    $msg,%ecx           # %ecx register contains 2nd argument (pointer to message to write)
          movl    $1,%ebx             # %ebx register contains 1st argument (file handle □ stdout)
          movl    $4,%eax             # %eax register contains system call number (4 == sys_write)
          int     $0x80               # instruction to call (“trap into”) kernel

                                      # These last 3 instructions call the exit system call (with return value 0)
          movl    $0,%ebx             # %ebx register contains 1st argument: exit code
          movq    $1,%eax             # %eax register contains system call number (1 == sys_exit)
          int     $0x80               # instruction to call (“trap into”) kernel

.data                                 # here we end the code section and start the data section
msg:
          .ascii "Hello World!\n"     # this is our string
          len = . - msg               # the length of our string: current address – address of start of string
```

Hello World! (Linux, x86-64)

```
.text                               # The tekst section contains the program *code*.

          .global _start            # Export entry point of program (linker/loader uses _start by convention).


_start:
                                    # To write helloworld string to stdout, arguments should be in registers
          movq    $len,%rdx         # %rdx register contains 3rd argument (length)
          movq    $msg,%rsi         # %rsi register contains 2nd argument (pointer to message to write)
          movq    $1,%rdi           # %rdi register contains 1st argument (file handle □ stdout)
          movq    $1,%rax           # %erax register contains system call number (1 == sys_write)
          syscall                   # new way to call (“trap into”) kernel

                                    # These last 3 instructions call the exit system call (with return value 0)
          movq    $0,%rdi           # %rdi register contains 1st argument: exit code
          movq    $60,%rax          # %eax register contains system call number (60 == sys_exit)
          syscall                   # new way to call (“trap into”) kernel

.data                               # here we end the code section and start the data section

msg:
          .ascii "Hello World!\n" # this is our string
          len = . - msg             # the length of our string: current address – address of start of string
```

System Calls: Process Management

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

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

- `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)

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**

System Calls: 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 issues a CTRL+C or CTRL+Z keyboard sequence?

(signal, alarm, kill)

Signal, alarm, kill

- `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)

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 `fd`s 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

```c
#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;
}
```

Pipe Example Quiz: Can we skip these closes?

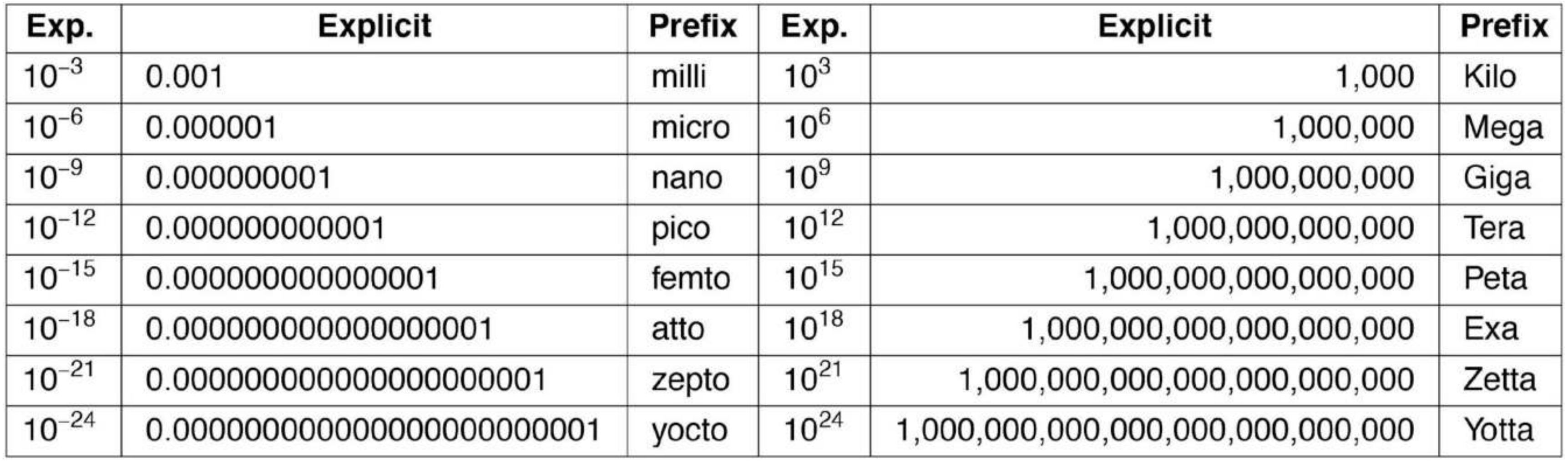
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#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;
}
```

And why / why not?

Recap

- We have seen syscalls
- How to start a new process
- Simple shell example
- Signals
- Pipes

Metric Units



Exp.	Explicit	Prefix	Exp.	Explicit	Prefix
10^{-3}	0.001	milli	10^{3}	1,000	Kilo
10^{-6}	0.000001	micro	10^{6}	1,000,000	Mega
10^{-9}	0.000000001	nano	10^{9}	1,000,000,000	Giga
10^{-12}	0.000000000001	pico	10^{12}	1,000,000,000,000	Tera
10^{-15}	0.000000000000001	femto	10^{15}	1,000,000,000,000,000	Peta
10^{-18}	0.000000000000000001	atto	10^{18}	1,000,000,000,000,000,000	Exa
10^{-21}	0.000000000000000000001	zepto	10^{21}	1,000,000,000,000,000,000,000	Zetta
10^{-24}	0.000000000000000000000001	yocto	10^{24}	1,000,000,000,000,000,000,000,000	Yotta

Figure 1-31. The principal metric prefixes.

Copyright