CSCI-GA.2250-001

Operating Systems

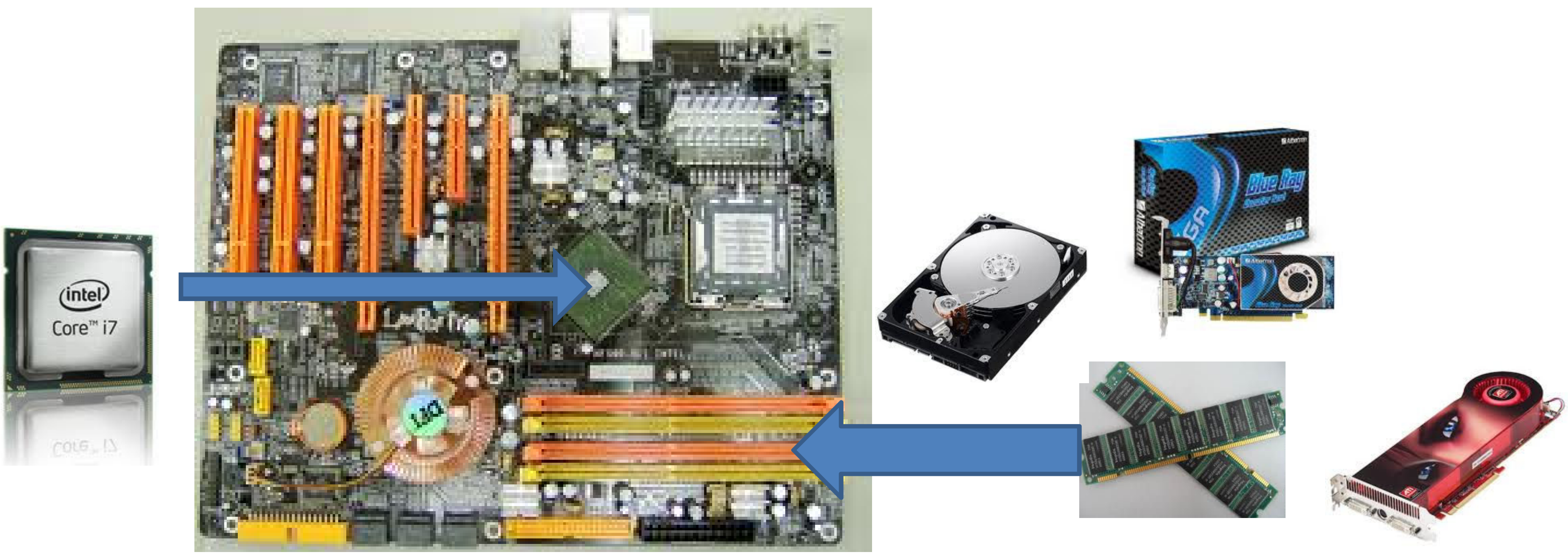
Introduction

Hubertus Franke

frankeh@cims.nyu.edu

Components of a Modern Computer

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

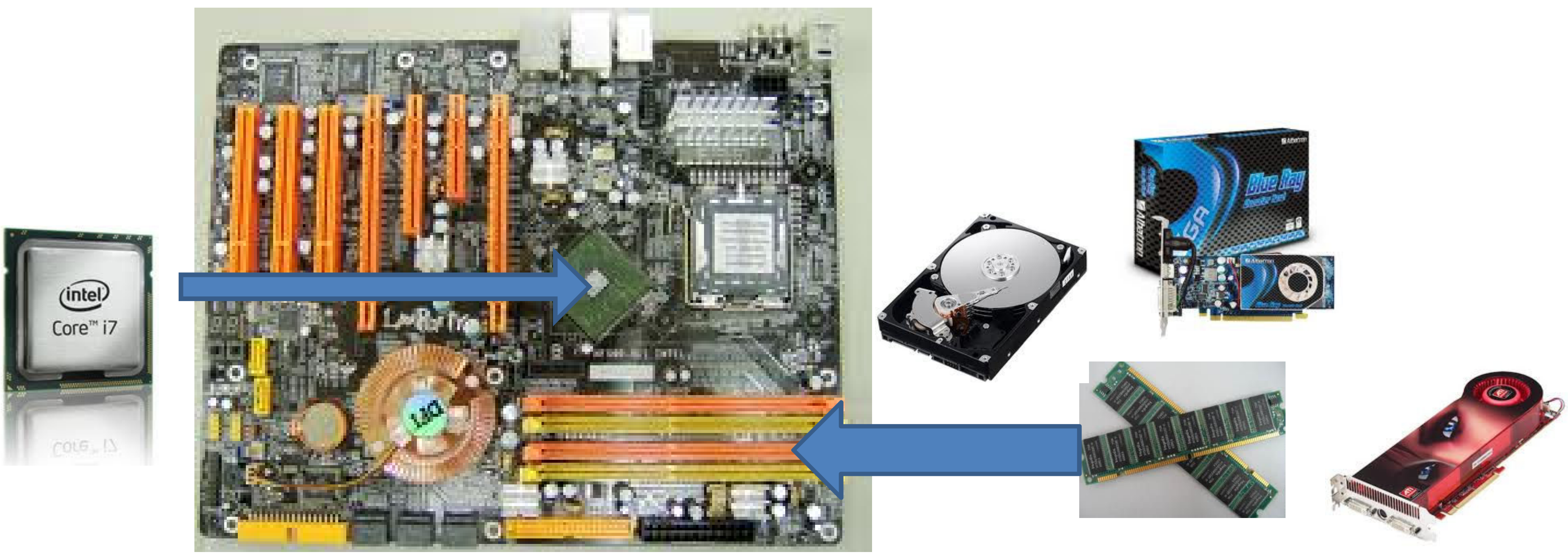


Media Player

emails

Games

Word Processing



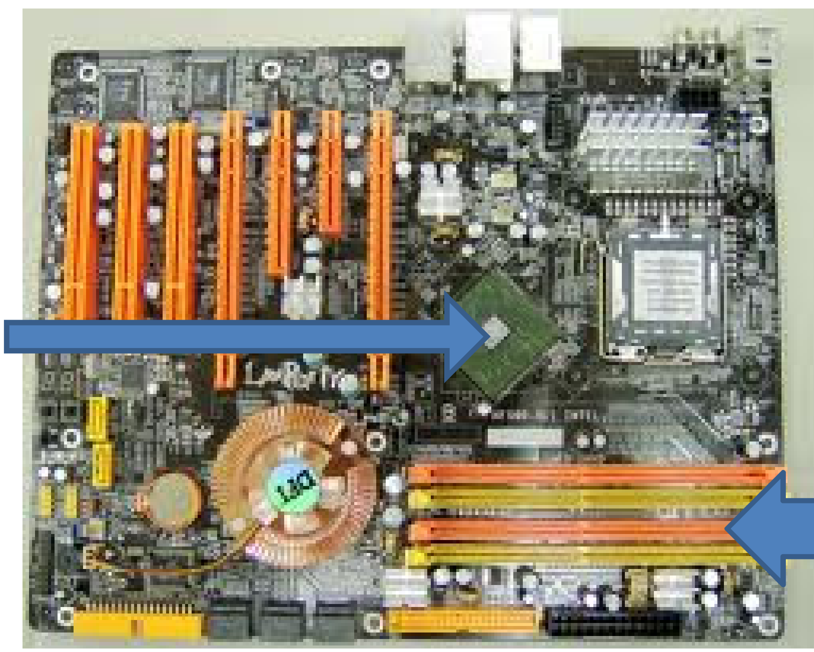
Media Player

emails

Games

Word Processing

Does a programmer need to understand all this hardware in order to write these software programs?



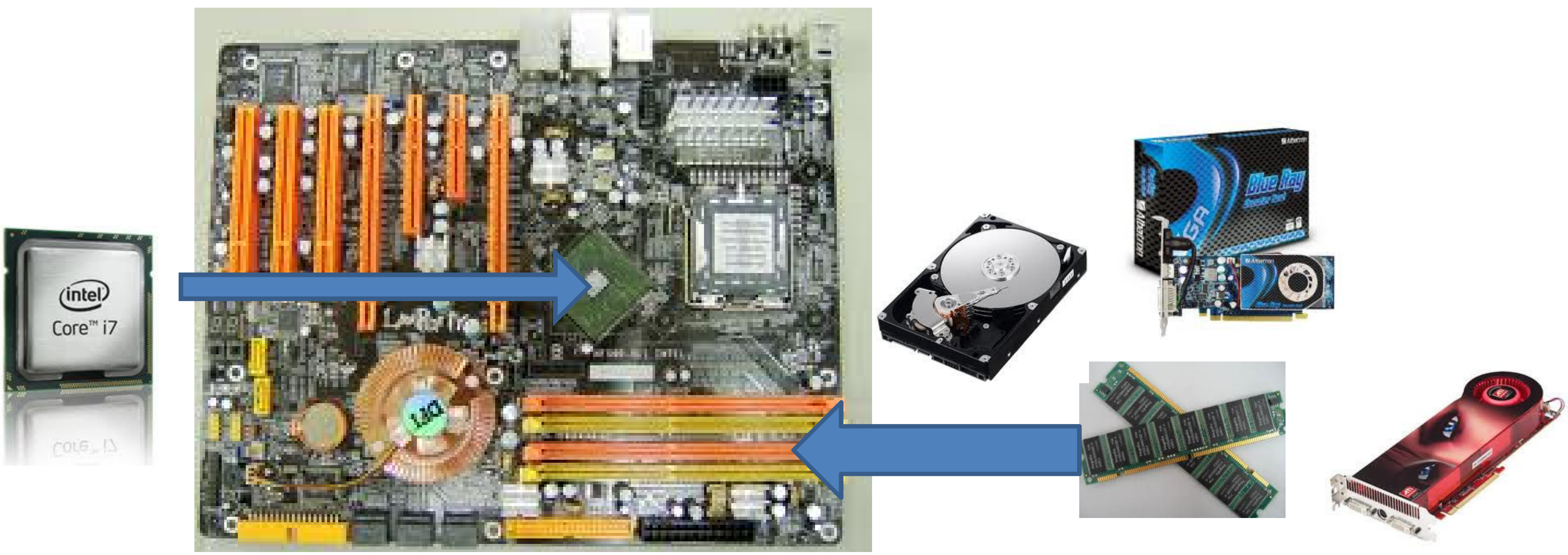
Media Player

emails

Games

Word Processing

Operating System



Components of a Modern Computer (2)

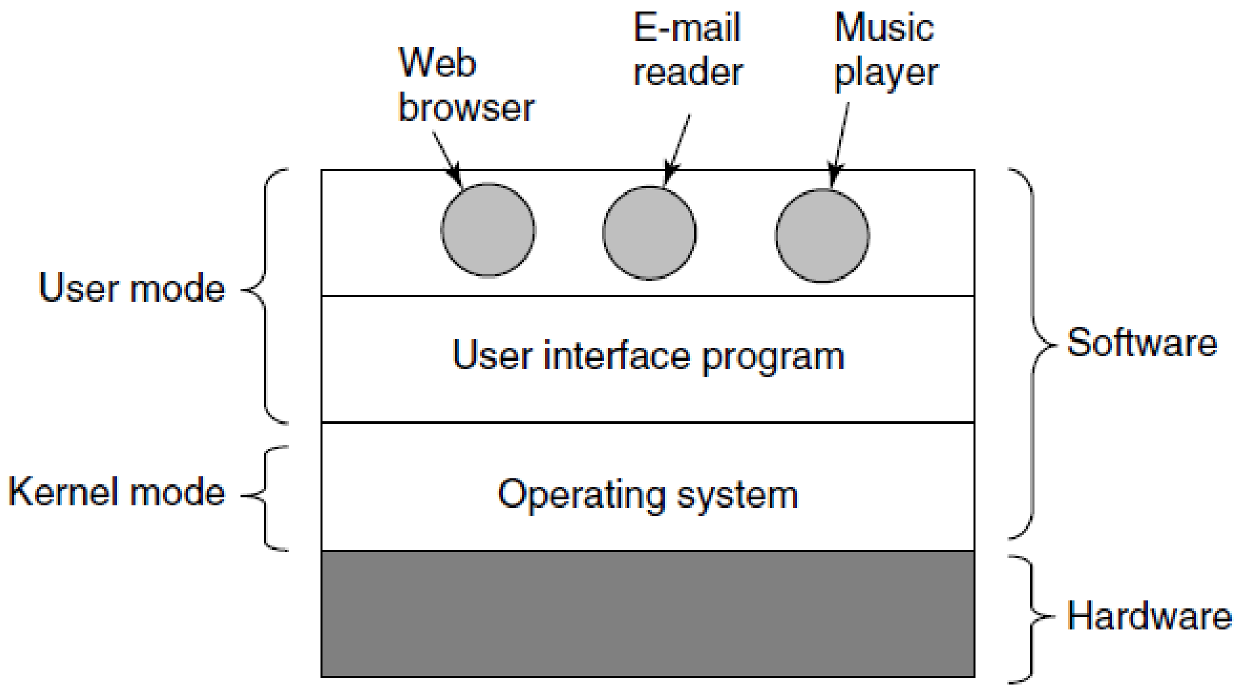
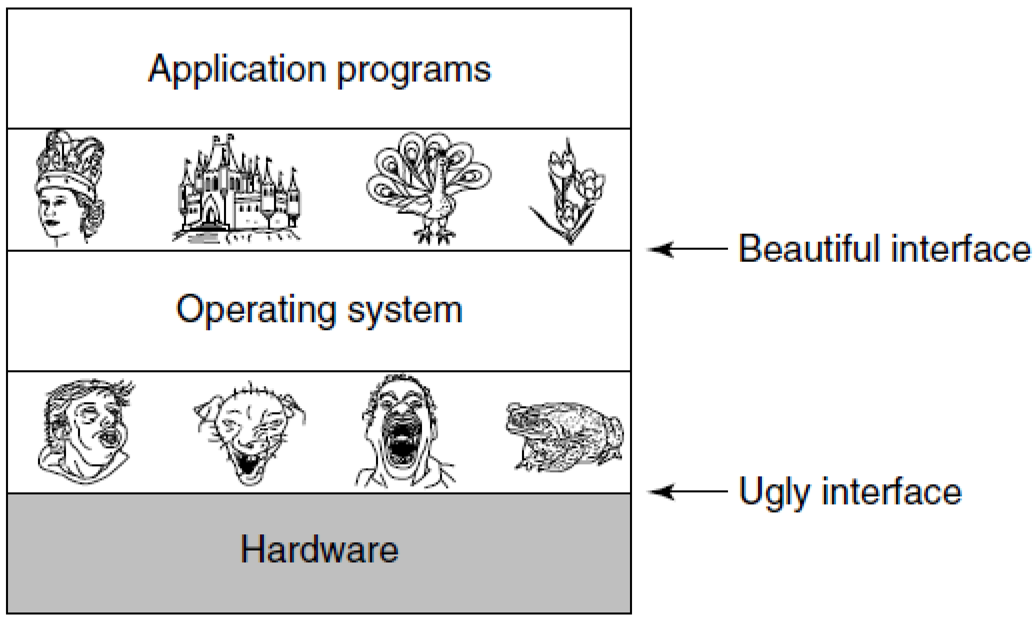


Figure 1-1. Where the operating system fits in.

The Operating System as an Extended Machine



Operating systems turn ugly hardware into beautiful abstractions.

The Operating System as a Resource Manager

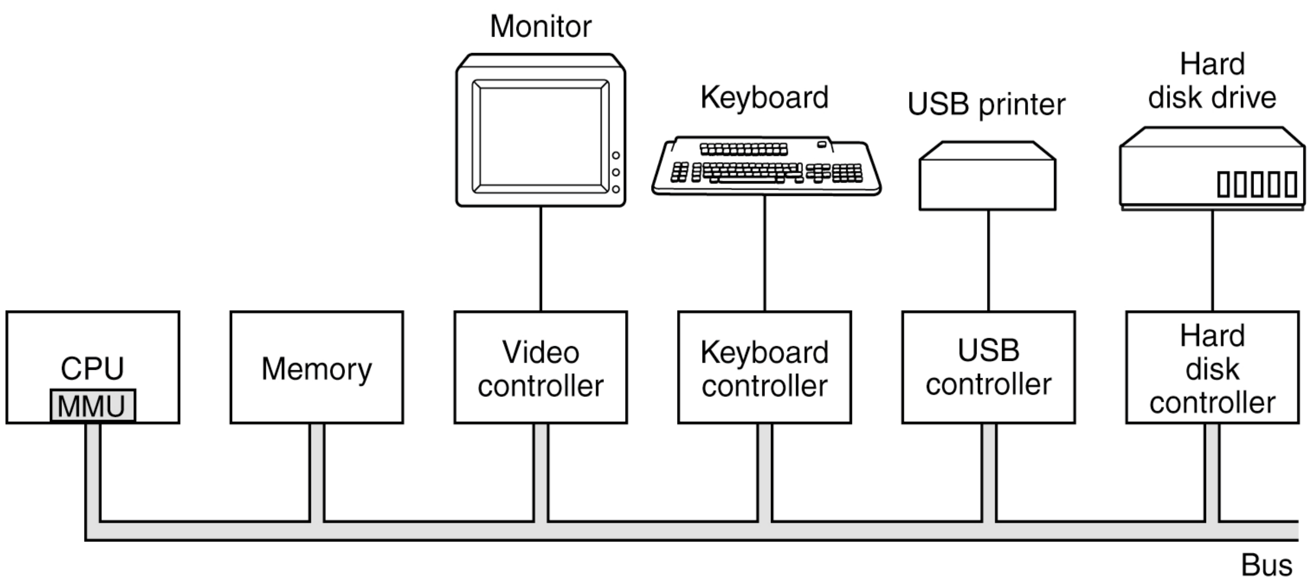
- Top down view
  - Provide abstractions to application programs
- Bottom up view
  - Manage pieces of complex system
- Alternative view
  - Provide orderly, controlled allocation of resources

The Two Main Tasks of OS

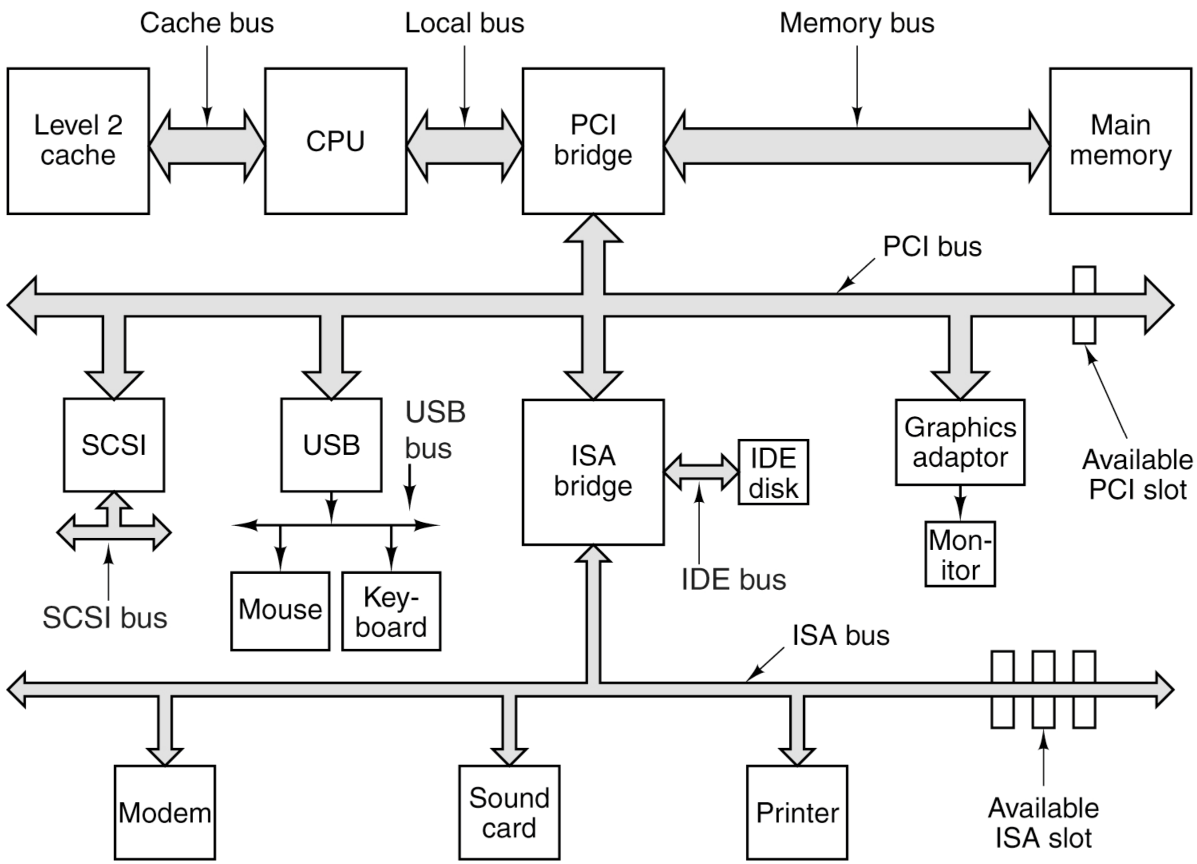
- Provide programmers (and programs) a clean abstract set of resources

- Manage the hardware resources

A Glimpse on Hardware



A Glimpse on Hardware



What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

Services

- Abstraction
- Simplification
- Convenience
- Standardization

CONTAINER

Makes computer usage simpler

What Is an OS?

Government

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

Finite resources

Competing demands

Examples:

- CPU
- Memory
- Disk
- Network

Limited budget,
Land,
Oil,
Gas,

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

You can't hurt me

I can't hurt you

Implies some degree of safety & security

Government

Law and order

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

The OS gives
The OS takes away

Voluntary at run time
Implied at termination
Involuntary
Cooperative

Government

Income Tax

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

illusion of infinite, private resources

Memory versus disk

Timeshared CPU

More extreme cases possible (& exist)

Government

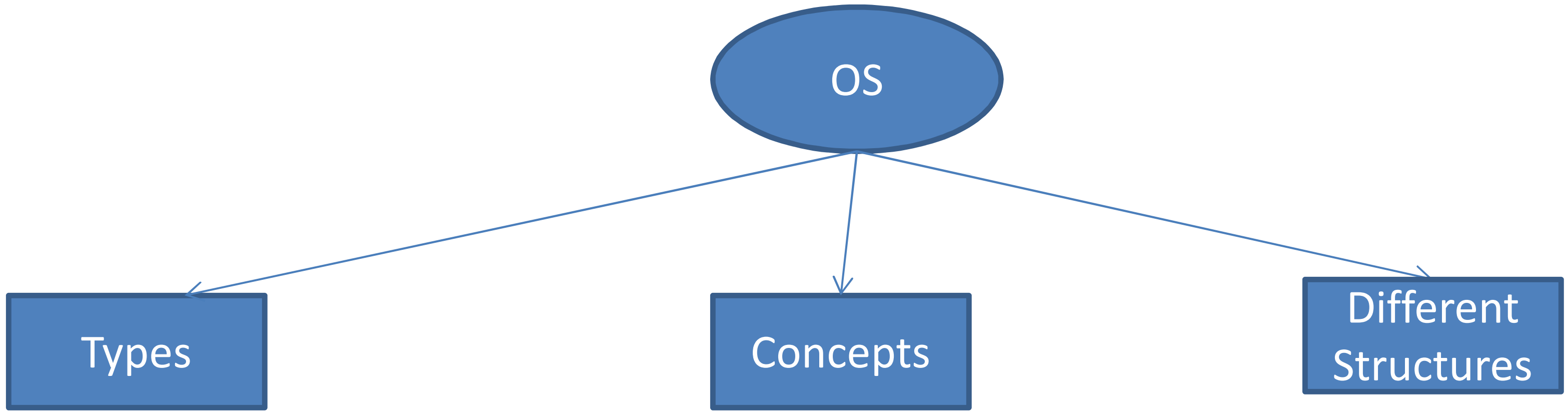
Social security

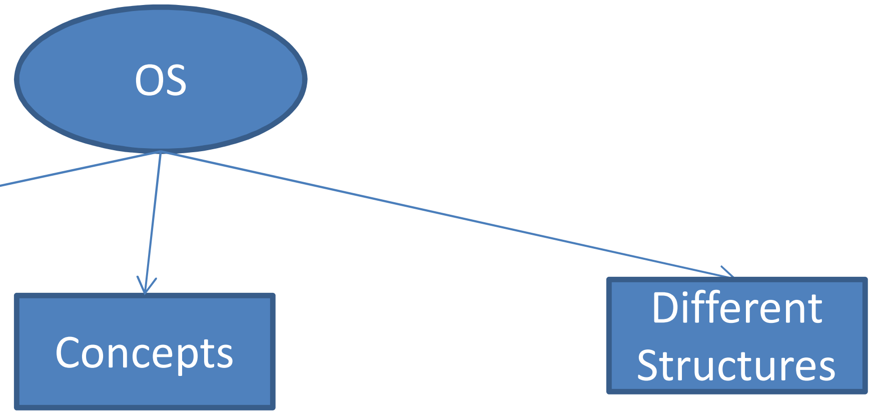
Booting Sequence

- BIOS starts
  - checks how much RAM
  - keyboard
  - other basic devices

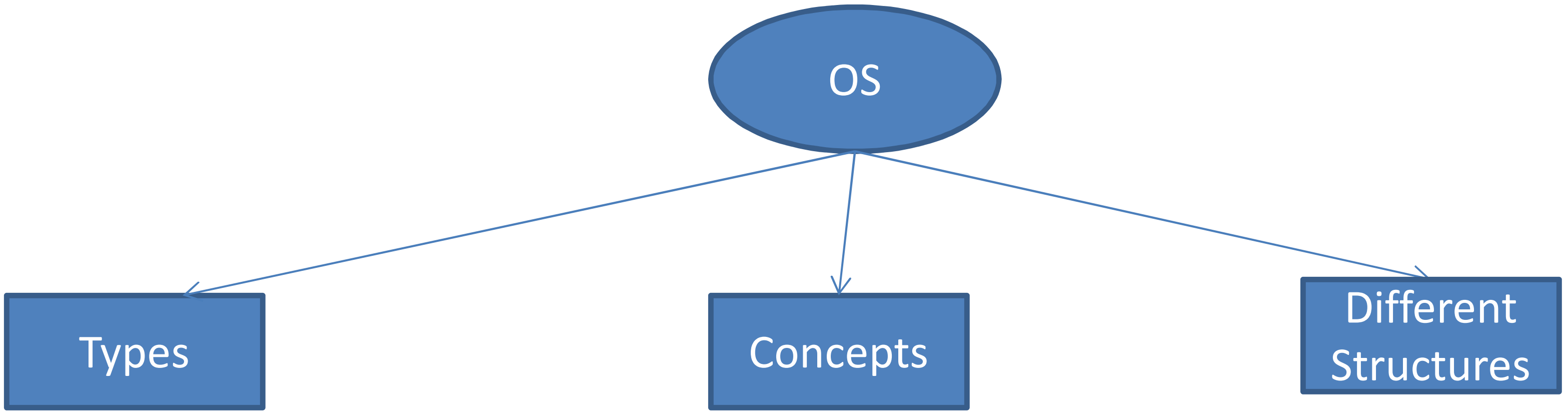
POST (Power On Self Test)

- BIOS determines boot Device
- The first sector in boot device is read into memory and executed to determine active partition
- Secondary boot loader is loaded from that partition.
- This loaders loads the OS from the active partition and starts it.


OS
Types
Concepts
Different Structures

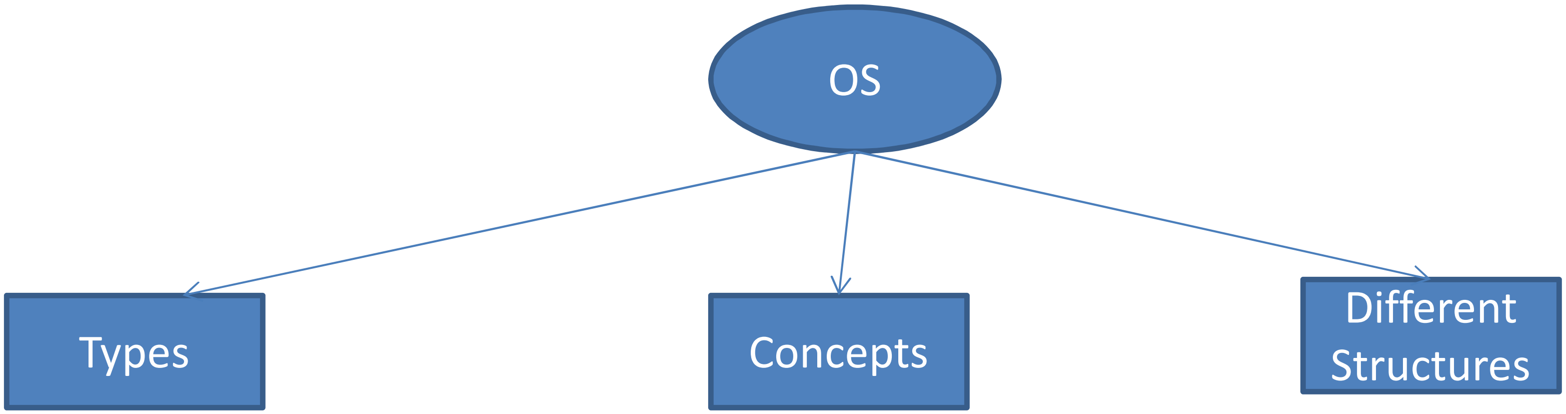


- Mainframe/supercomputer OS
    - batch
    - transaction processing
    - timesharing
    - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS



- Mainframe OS/supercomputer
    - batch
    - transaction processing
    - timesharing
    - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

- **Processes**
    - **Its address space**
    - **Its resources**
    - **Process table**
- **Address space**
- **File system**
- **I/O**
- **Protection**



- Mainframe OS/supercomputer
  - batch
  - transaction processing
  - timesharing
  - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

- Processes
  - Its address space
  - Its resources
  - Process table
- Address space
- File system
- I/O
- Protection

- **Monolithic**
- **Layered systems**
- **Microkernels**
- **Client-server**
- **Virtual machines**

OS

Types

- Mainframe OS
    - batch
    - transaction processing
    - timesharing
    - e.g. OS/390
- Server OS
- Multiprocessor OS
- PC OS
- Embedded OS
- Sensor node OS
- RTOS
- Smart card OS

Concepts

- Processes
    - Its address space
    - Its resources
    - Process table
- Address space
- File system
- I/O
- Protection

Different Structures

- **Monolithic**
- **Layered systems**
- **Microkernels**
- **Client-server**
- **Virtual machines**

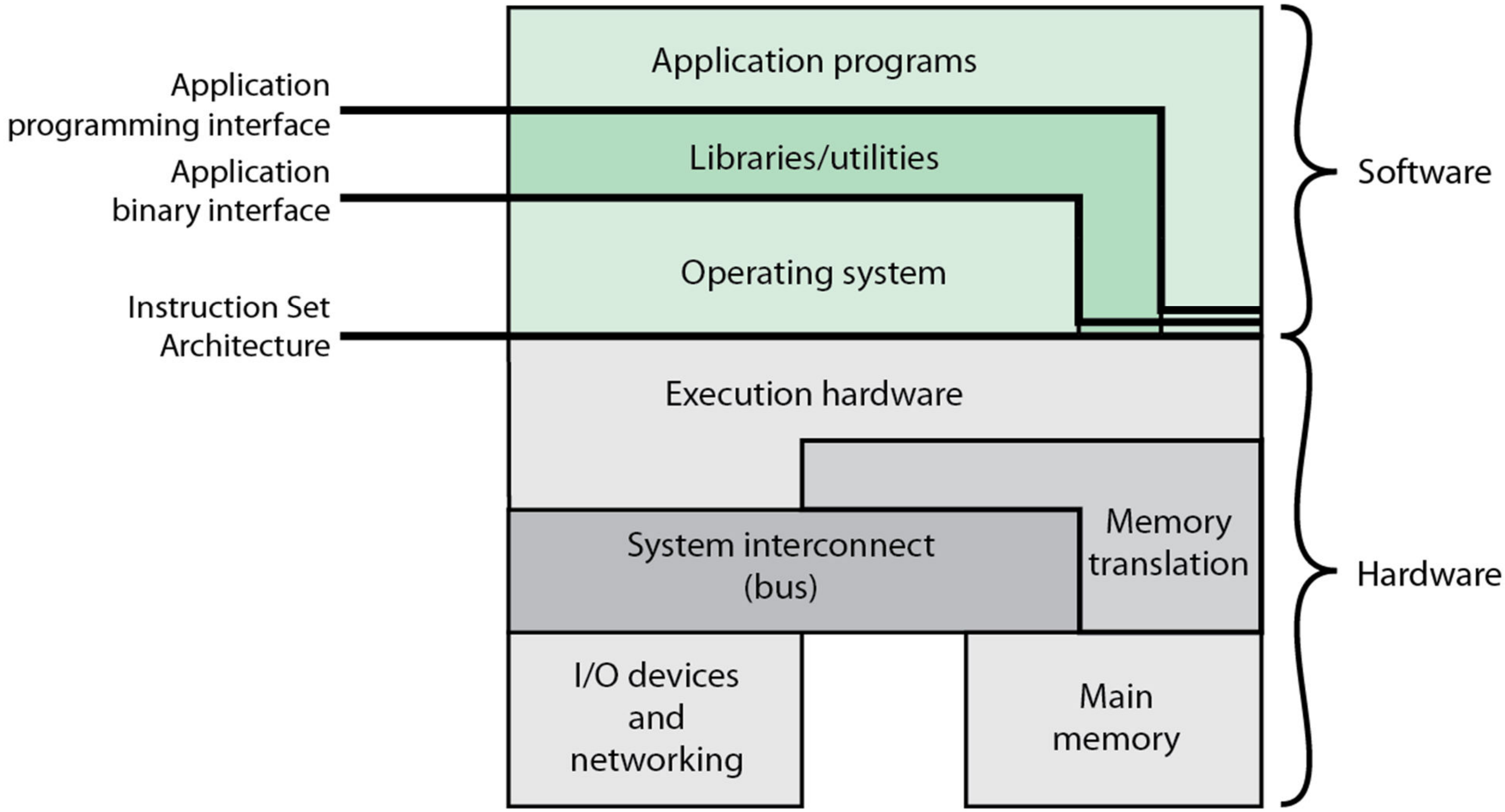
Main objectives of an OS:

- Convenience
- Efficiency
- Ability to evolve

OS Services

- Program development
- Program execution
- Access I/O devices
- Controlled access to files
- System access
- Error detection and response
- Accounting

Hardware and Software Infrastructure



Computer Hardware and Software Infrastructure

In a nutshell

- OS is really a manager:
  - programs, applications, and processes are the customers
  - The hardware provide the resources
- OS works in different environments and under different restrictions (supercomputers, workstations, notebooks, tablets, smartphones, real-time, …)

History of Operating Systems

- "*We can chart our future clearly and wisely only when we know the path which has led to the present.*"
  - **dlai E. Stevenson, Lawyer and Politician**

- First generation 1945 - 1955
  - vacuum tubes, plug boards (no OS)
- Second generation 1955 - 1965
  - transistors, batch systems
- Third generation 1965 – 1980
  - ICs and multiprogramming
- Fourth generation 1980 – present
  - server computers
  - personal computers, hand-held devices, sensors

History of Operating Systems (1945-55)

- Programming and Controlled tied to the Computer

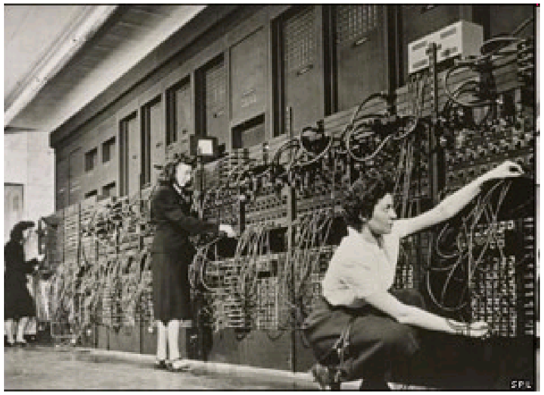
Defining characteristics of some early digital computers of the 1940s (In the history of computing hardware)

Name	First operational	Numeral system	Computing mechanism	Programming	Turing complete
Zuse Z3 (Germany)	May 1941	Binary floating point	Electro-mechanical	Program-controlled by punched 35 mm film stock (but no conditional branch)	Yes (1998)
Atanasoff–Berry Computer (US)	1942	Binary	Electronic	Not programmable—single purpose	No
Colossus Mark 1 (UK)	February 1944	Binary	Electronic	Program-controlled by patch cables and switches	No
Harvard Mark I – IBM ASCC (US)	May 1944	Decimal	Electro-mechanical	Program-controlled by 24-channel punched paper tape (but no conditional branch)	No
Colossus Mark 2 (UK)	June 1944	Binary	Electronic	Program-controlled by patch cables and switches	No
Zuse Z4 (Germany)	March 1945	Binary floating point	Electro-mechanical	Program-controlled by punched 35 mm film stock	Yes
ENIAC (US)	July 1946	Decimal	Electronic	Program-controlled by patch cables and switches	Yes
Manchester Small-Scale Experimental Machine (Baby) (UK)	June 1948	Binary	Electronic	Stored-program in Williams cathode ray tube memory	Yes
Modified ENIAC (US)	September 1948	Decimal	Electronic	Read-only stored programming mechanism using the Function Tables as program ROM	Yes
EDSAC (UK)	May 1949	Binary	Electronic	Stored-program in mercury delay line memory	Yes
Manchester Mark 1 (UK)	October 1949	Binary	Electronic	Stored-program in Williams cathode ray tube memory and magnetic drum memory	Yes
CSIRAC (Australia)	November 1949	Binary	Electronic	Stored-program in mercury delay line memory	Yes

Source: wikipedia

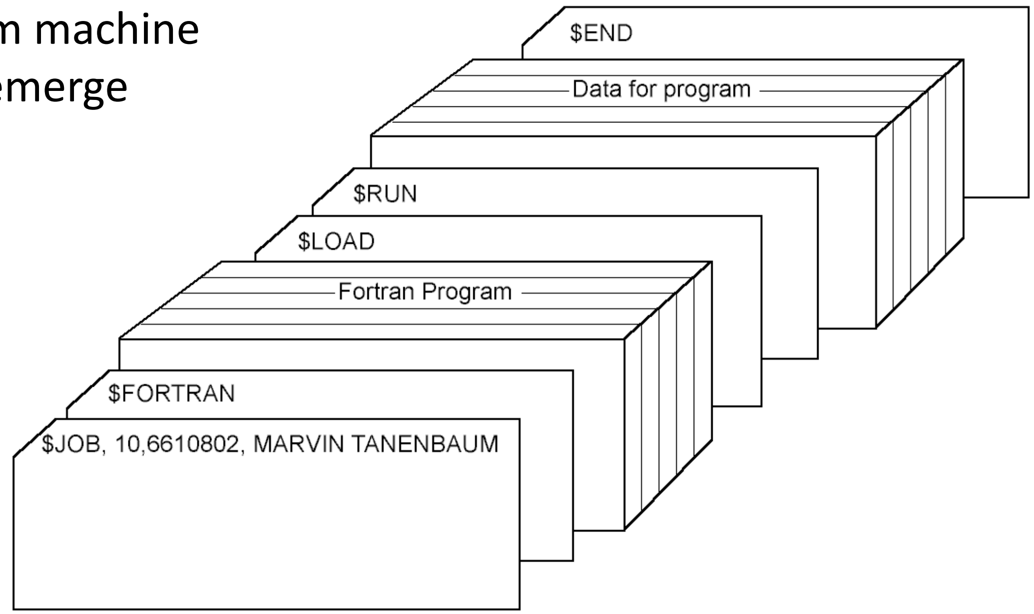
History of Operating Systems (1945-1955)

- Vacuum tubes, plug boards (no OS)
  - ENIAC (UPenn 1944)
    - 30 tons, 150m, 5000calcs/sec
  - Zuse's Z3 (1941)
    - 2000 relays
    - 22 bit words
    - 5-10 Hz
- What's a bug?



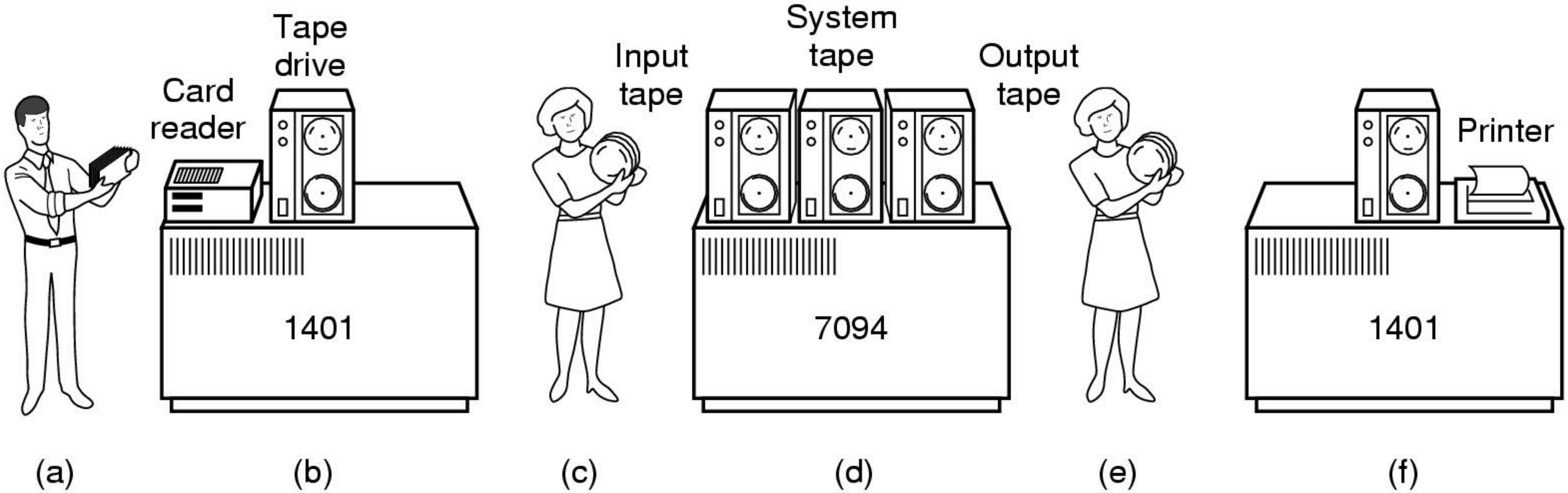
History of Operating Systems (1955-65)

- Emergence of the Mainframe
- Programmers isolated from machine
- Programming Languages emerge
  - Fortran
  - Cobol



- Structure of a typical JCL job – 2nd generation
- Single user
- Programmer/User as the operator
- Secure, but inefficient use of expensive resources
- Low CPU utilization-slow mechanical I/O devices

History of Operating System (1955-65)

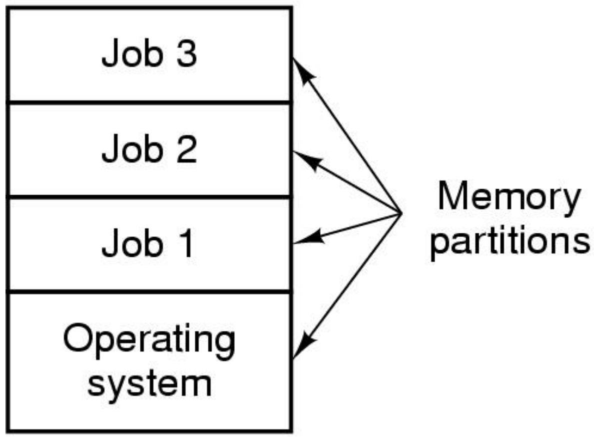


Early batch system

- bring cards to 1401
- read cards to tape
- put tape on 7094 which does com
- put tape on 1401 which prints ou

History of Operating Systems (1965-80)

- Multiprogramming systems
  - Multiple jobs in memory – 3rd generation
  - Allow overlap of CPU and I/O activity
  - Polling/Interrupts, Timesharing
  - Spooling



- Different types
  - Epitomized by the IBM 360 machine
  - MFT (IBM OS/MFT) Fixed Number of Tasks
  - MVT (IBM OS/MVT) Variable Number of Tasks

- Birth of Modern Operating System Concepts
  - Time Sharing: when and what to run → scheduling
  - Resource Control: memory management, protection

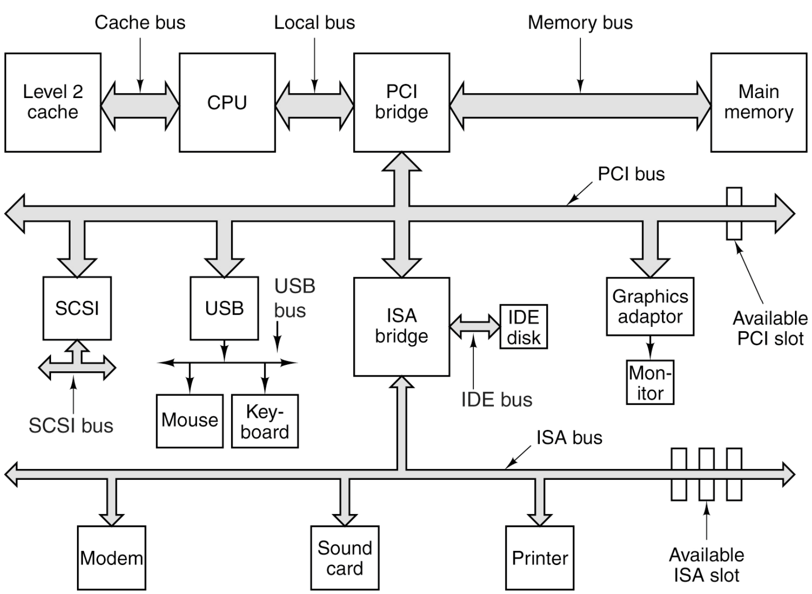
The Operating System Jungle / Zoo (1980-present)

- Mainframe operating systems
- Server operating systems
- Multiprocessor operating systems
- Personal computer operating systems
- Real-time operating systems
- Embedded operating systems
- Smart card operating systems
- Cellphone/tablet operating systems
- Sensor operating systems

Computer Architecture

(a closer look)

We must know and understand
what is actually managed by an OS



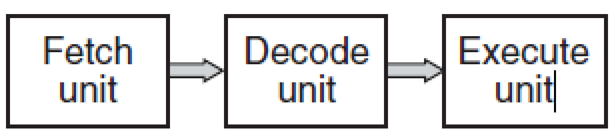
Processors

- Each CPU has a specific set of instructions
  - ISA (Instruction Set Architecture)
    - RISC: Sparc, MIPS, PowerPC
    - CISC: x86, zSeries

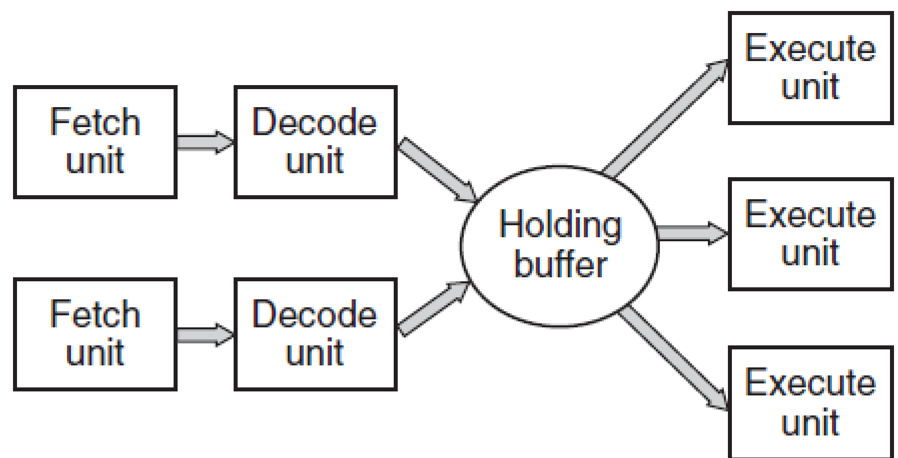
- All CPUs contain
  - General registers inside to hold key variables and temporary results
  - Special registers visible to the programmer
    - Program counter contains the memory address of the next instruction to be fetched
    - Stack pointer points to the top of the current stack in memory
    - PSW (Program Status Word) contains the condition code bits which are set by comparison instructions, the CPU priority, the mode (user or kernel) and various other control bits.

How Processors Work

- Execute instructions
  - CPU cycles
    - Fetch (from mem) → decode → execute
    - Program counter (PC)
      - When is PC changed?
    - Pipeline: fetch n+2 while decode n+1 while execute n



(a)

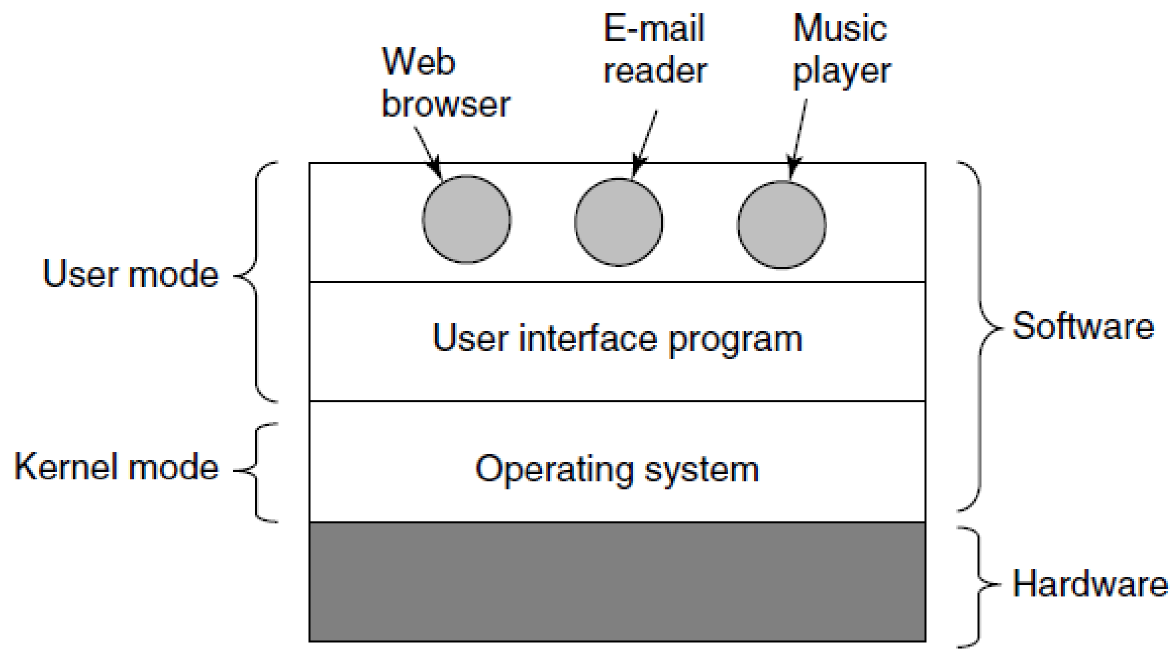


(b)

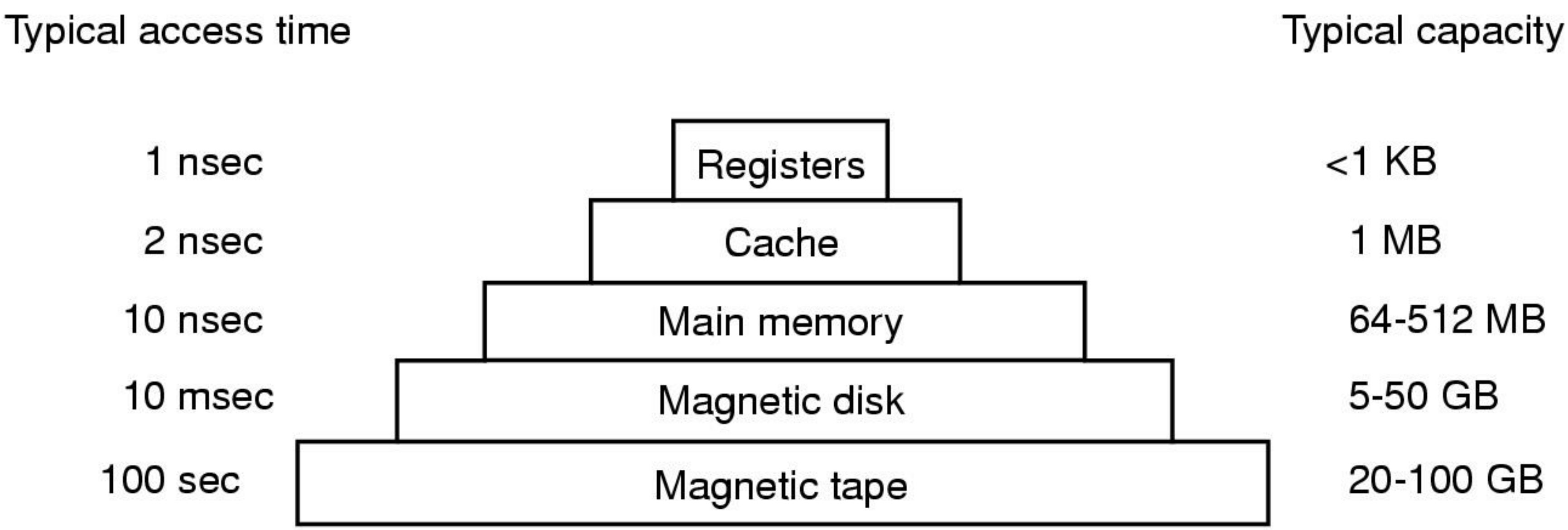
(a) A three-stage pipeline. (b) A superscalar CPU.

How Processors Work

- Two modes of CPU (why?)
  - User mode (a subset of instructions)
  - Privileged mode (all instruction)
- Trap (special instruction)



Memory-Storage Hierarchy



- Other metrics:
  - Bandwidth (e.g. MemBandwidth 30GB/s → 200GB/s, Disk ~70-200MB/s)
- What can an OS do to increase its "performance"
  - Active Cache management…

Memory Access

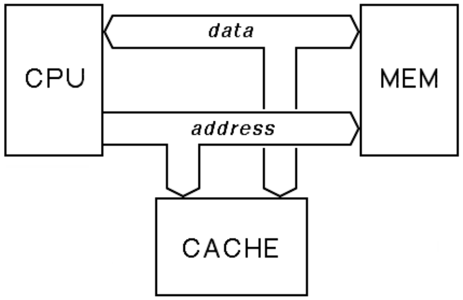
- Memory read:
  - Assert address on address lines
  - Wait till data appear on data line
  - Much slower than CPU!

- How many mem access for one instruction?

Memory Access

- Memory read:
  - Assert address on address lines
  - Wait till data appear on data line
  - Much slower than CPU!
- How many mem access for one instruction?
  - Fetch instruction
  - Fetch operand (0, 1 or 2)
  - Write results (0 or 1)
- How to speed up instruction execution?

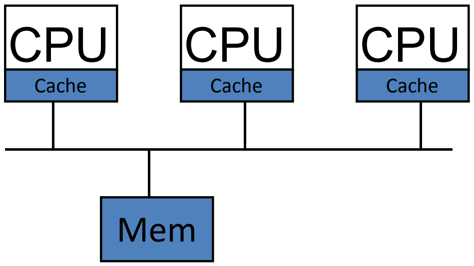
CPU Caches



- Principle:
  - Data/Instruction that were recently used are "likely" used again in short period
  - Used in "many" subsystems (I/O, filesystems, …)

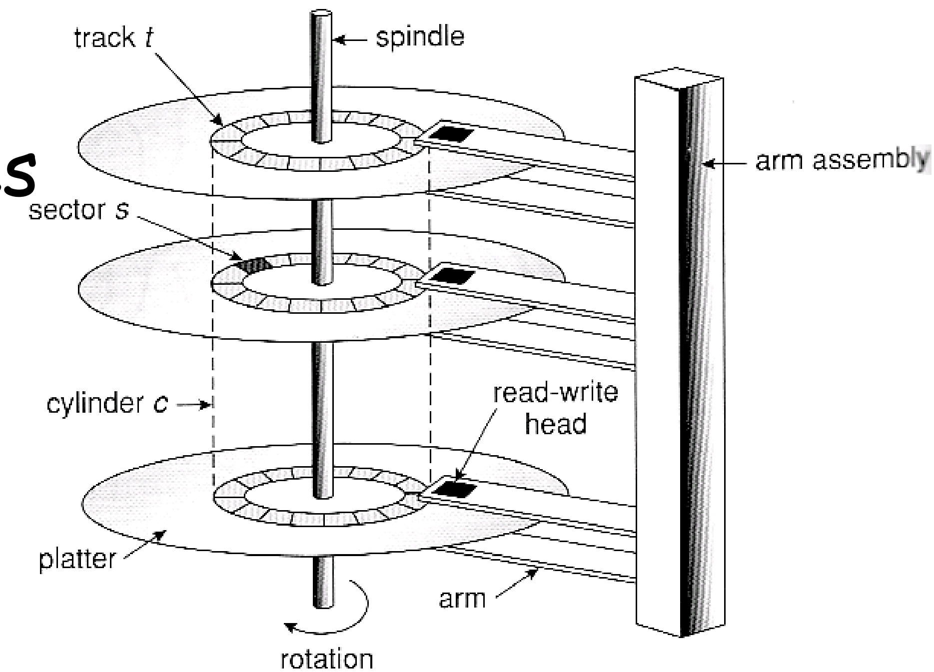
- Cache hit:
  - no need to access memory
- Cache miss:
  - data obtained from mem, possibly update cache

- Issues
  - Operation MUST be correct
  - Cache management for Memory done in hardware



Example of Device (resource and operation)

- Disk:
  - Multiple-subdevices
  - Translations
    - Block -> sector
  - Head Movement
  - Seek Time
  - Data Placement



Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Eighth Edition ", Chapter 12

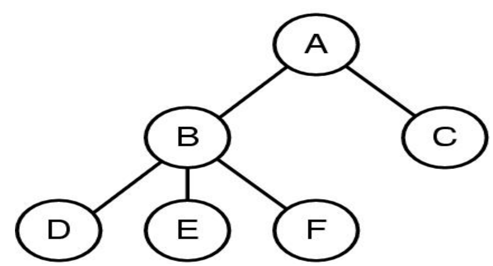
  - Power Management

OS Major Components

- Process and thread management
- Resource management
  - CPU
  - Memory
  - Device
- File system
- Bootstrapping

Process: a running program

- A process includes
  - Address space
  - Process table entries (state, registers)
    - Open files, thread(s) state, resources held
- A process tree
  - A created two child processes, B and C
  - B created three child processes, D, E, and F



Some System Calls For Process Management

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

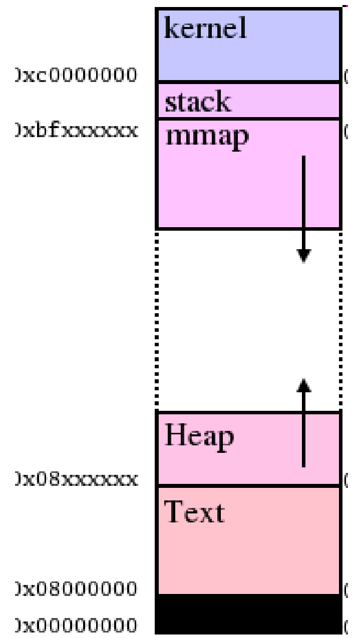
Signal

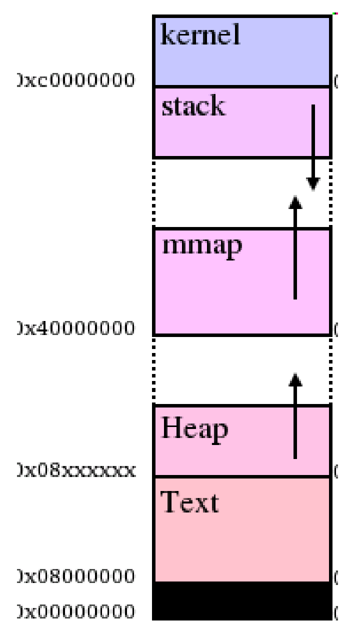
Call	Description
kill(pid, signal)	Deliver signal to the process pid
signal(signal, function)	Define which function to call for signal

Address Space

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections
- ReadOnly, ReadWrite, Execute

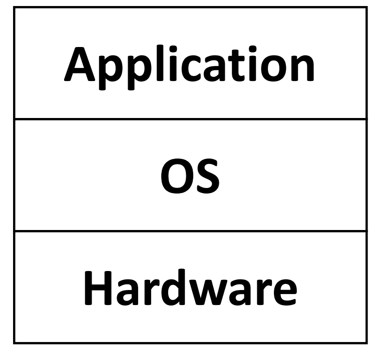
- Confined "private" addressing concept
  - ➔ requires form of address virtualization





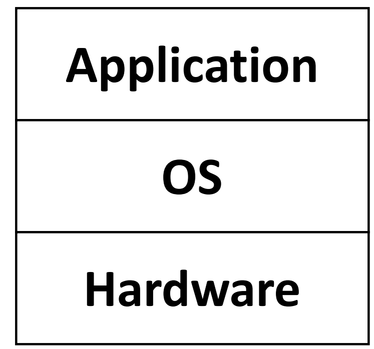
Recap: What is an OS ?

- Code that:
  - Sits between programs & hardware
  - Sits between different programs
  - Sits betweens different users
- Job of OS:
  - Manage hardware resources
    - Allocation, protection, reclamation, virtualization
  - Provide services to app.
    - Abstraction, simplification, standardization

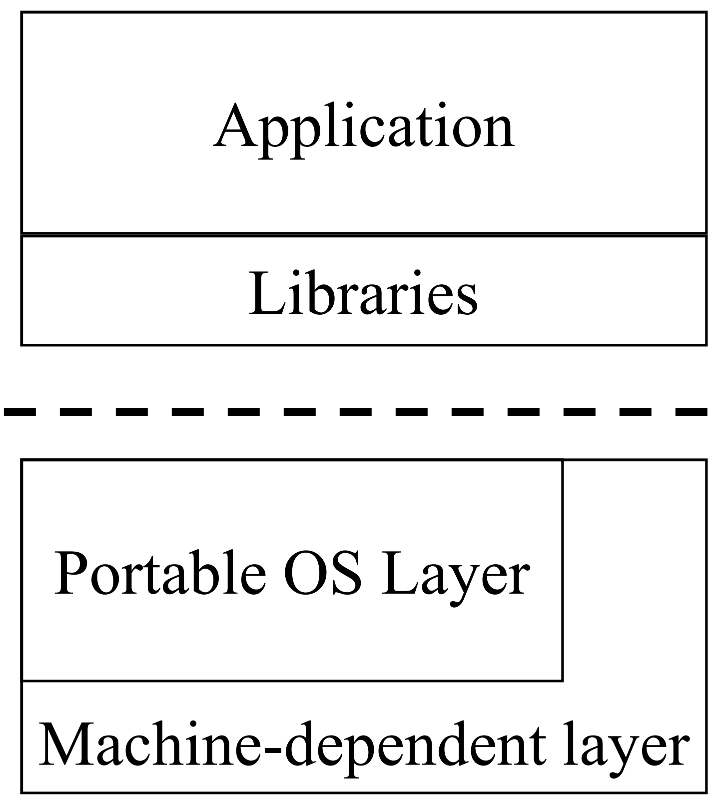


Recap: What is an OS ?

- Code that:
  - Sits between programs & hardware
  - Sits between different programs
  - Sits betweens different users
- Job of OS:
  - Manage hardware resources
    - Allocation, protection, reclamation, virtualization
  - Provide services to app. How ? → system call
    - Abstraction, simplification, standardization



A peek into Unix/Linux

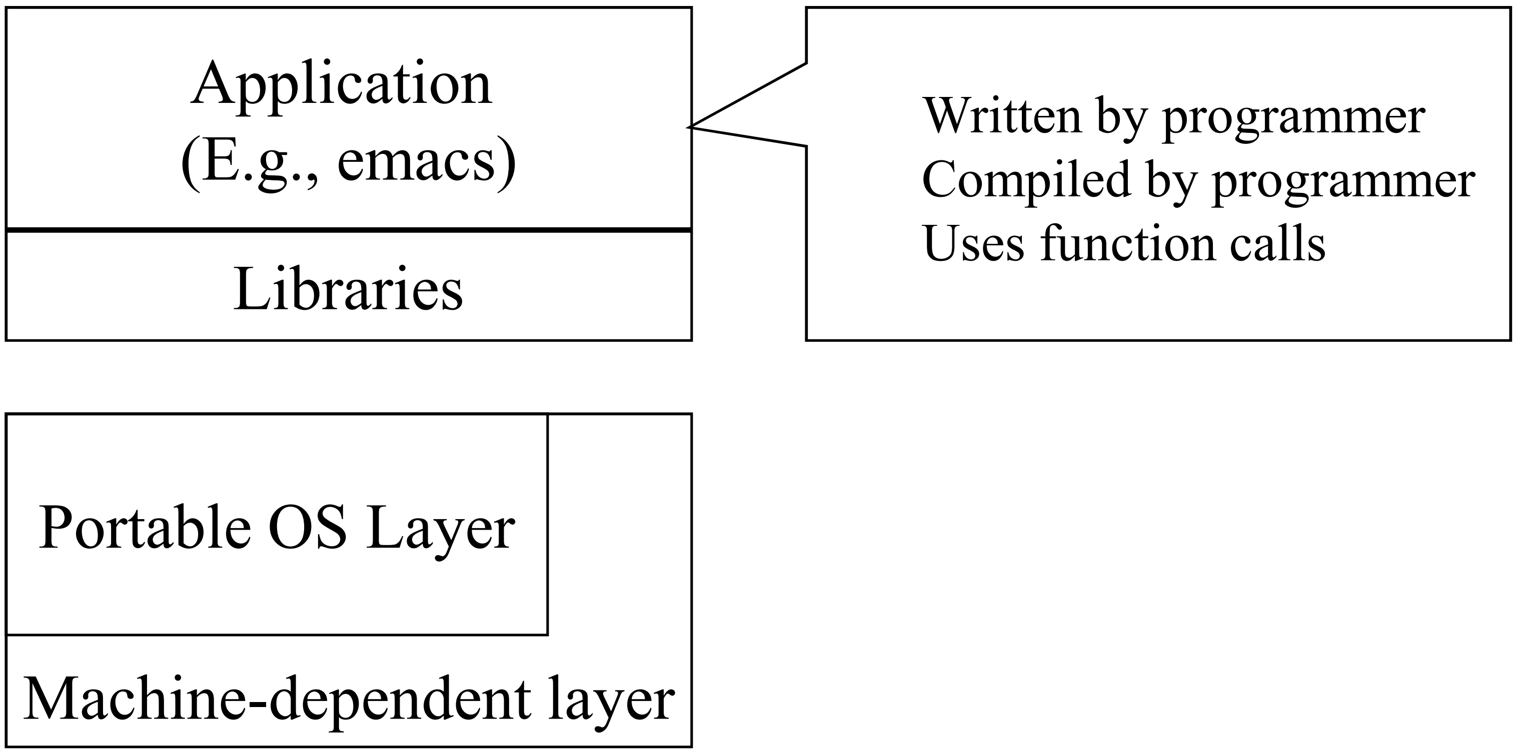


User space/level

Kernel space/level

- User/kernel modes are supported by hardware
- Some systems do not have clear user-kernel boundary

Unix: Application



Unix: Libraries

Application

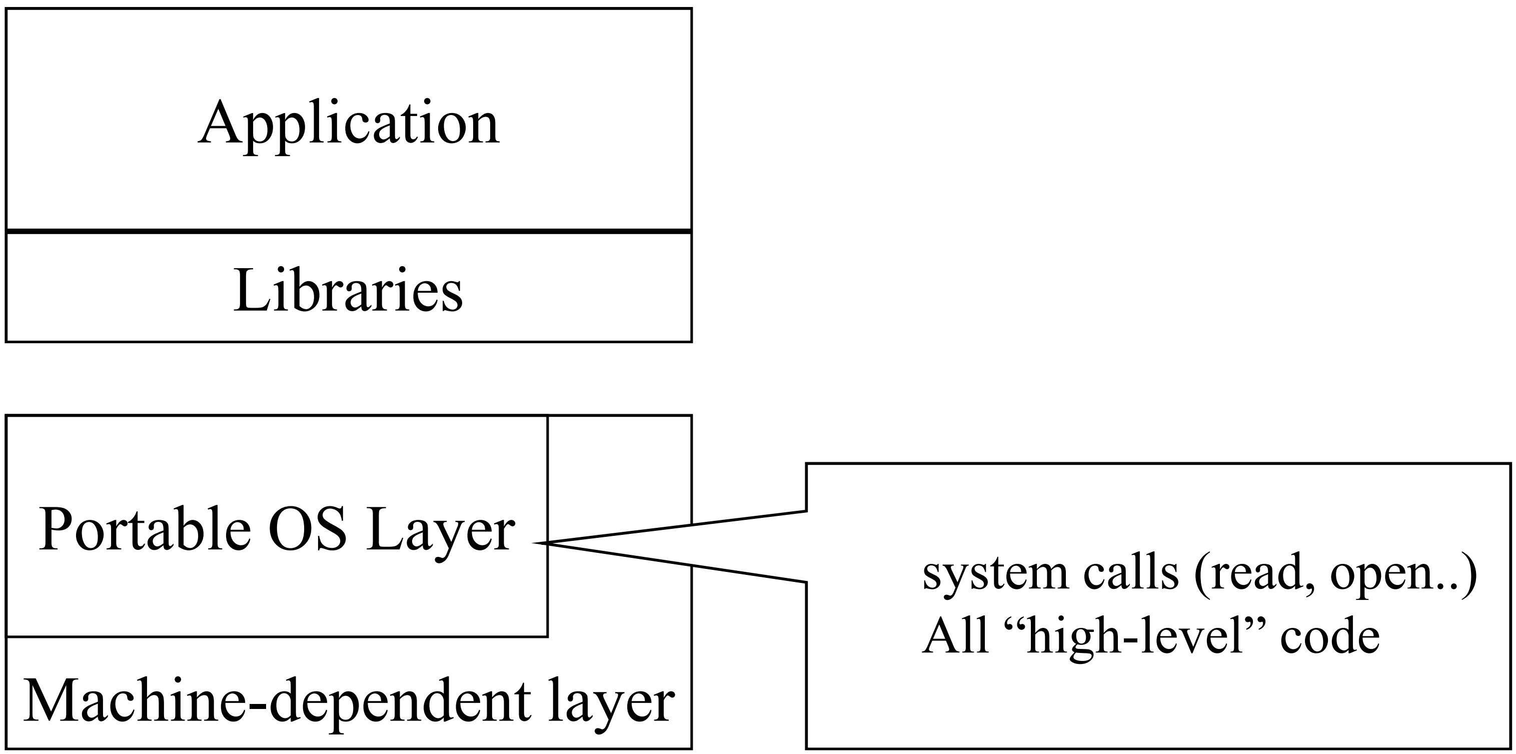
Libraries (e.g., stdio.h)

Portable OS Layer

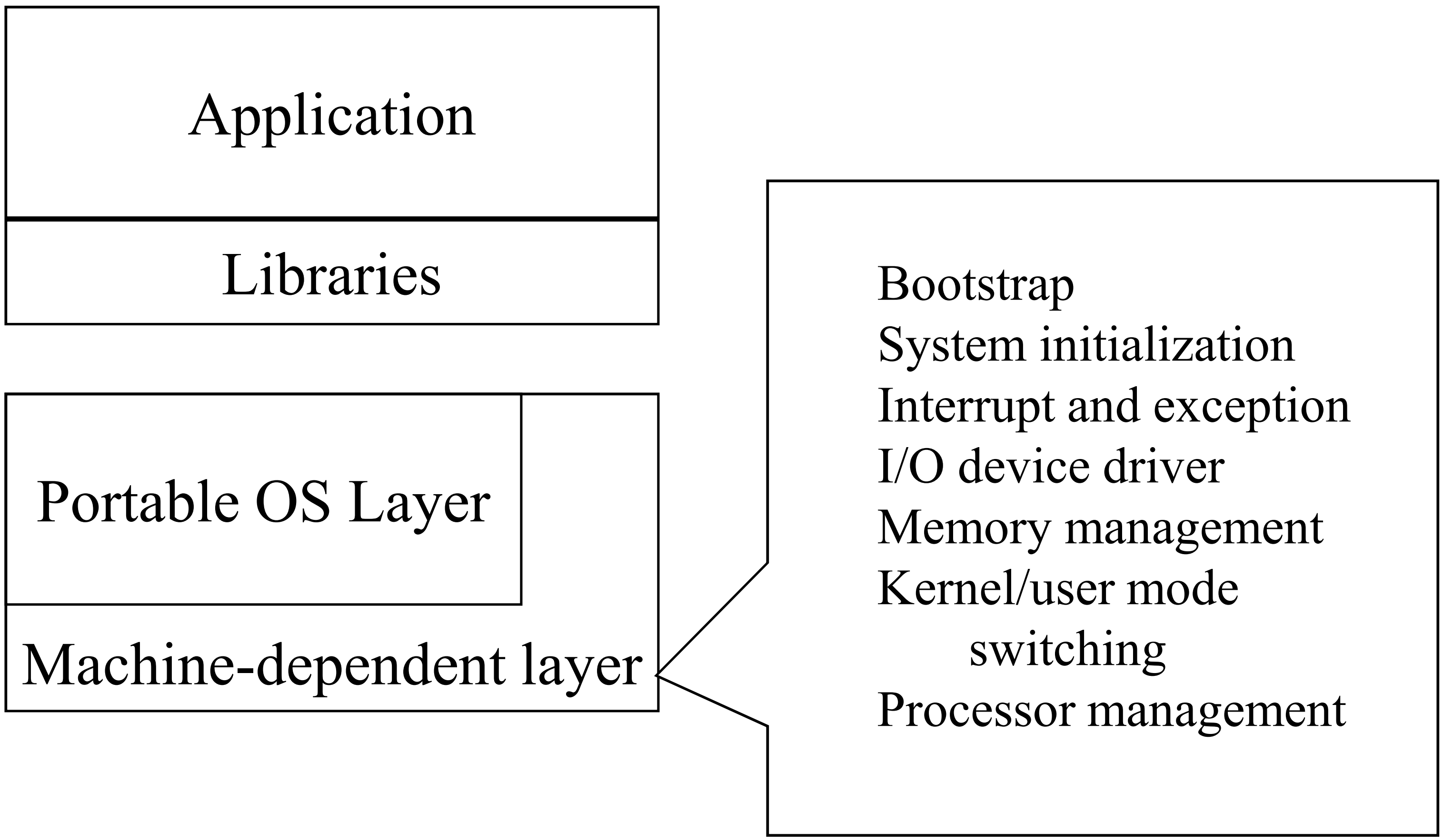
Machine-dependent layer

Provided pre-compiled
Defined in headers
Input to linker (compiler)
Invoked like functions
May be "resolved" when
program is loaded

Typical Unix OS Structure



Typical Unix OS Structure



System Call

- Invoked via non-priviliged instruction
  - TRAP
  - Treated often like an interupt, but its "somewhat" different

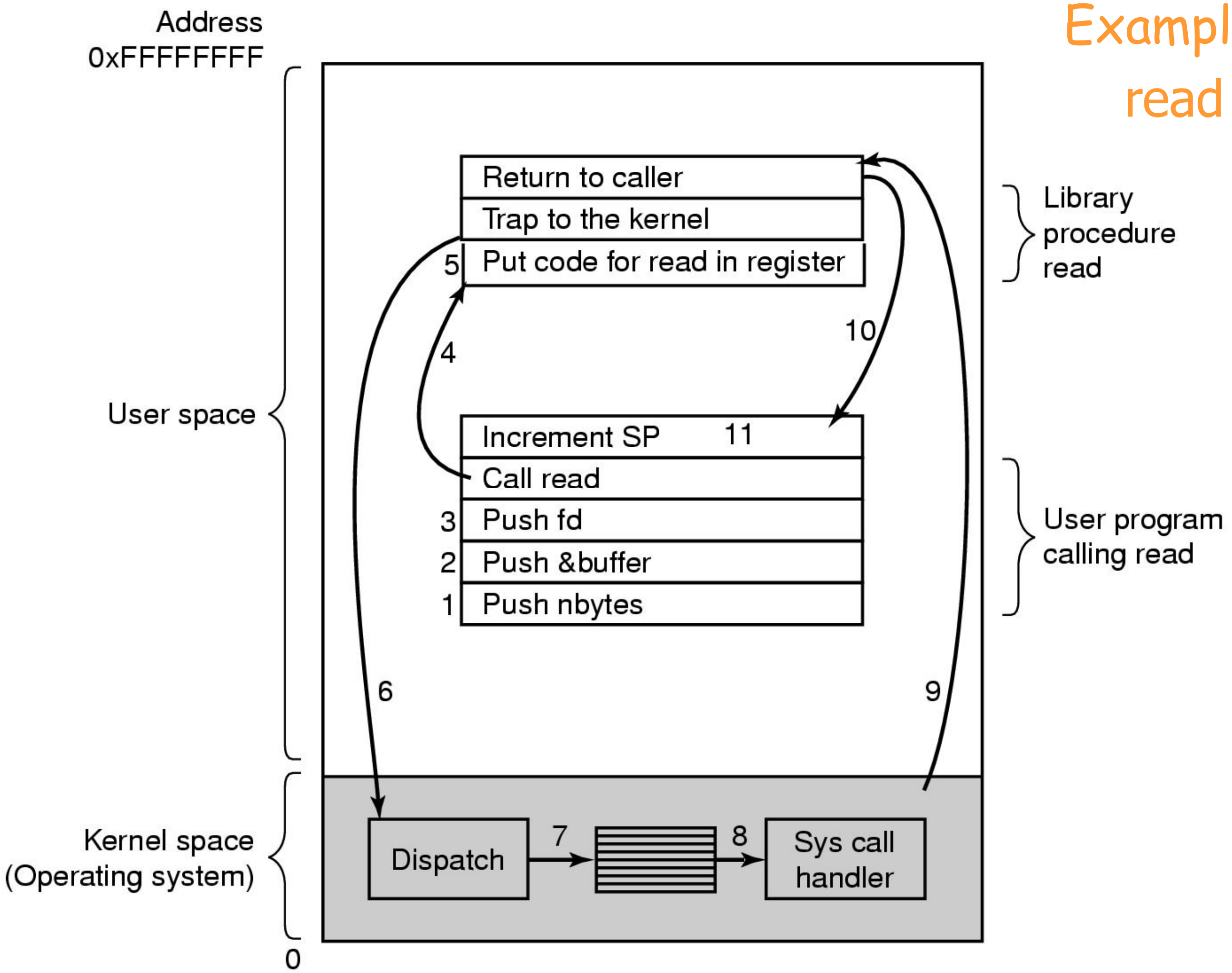
- Synchronous transfer control

- Side-effect of executing a trap in userspace is that an exception is raised and program execution continues at a prescribed instruction in the kernel → syscall_handler()

Steps in Making a System Call

Example:

read (fd, buffer,nbytes)



System Calls (POSIX)

- System calls for process management
- Example of fork used in simplified shell program

```
#define TRUE 1
 while(TRUE) {
   type_prompt();
   read_command(command, parameters);
   if (fork()!=0) {
              /* some code*/
               waitpid(-1,&status, 0);}
   else {
               /* some code*/
               execve(command, parameters,0);
   } }
```

Portable Operating System Interface for Unix   (IEEE standard 90's)

System Calls (POSIX)

- System calls for file/directory management
  - fd=open(file,how,....)
  - n=write(fd,buffer,nbytes)
  - s=rmdir(name)
- Miscellaneous
  - s=kill(pid,signal)
  - s=chmod(name,mode)

System Calls (Windows Win32 API)

- Process Management
  - CreateProcess- new process (combined work of fork and execve in UNIX)
    - In Windows – no process hierarchy, event concept implemented
  - WaitForSingleObject – wait for an event (can wait for process to exit)
- File Management
  - CreateFile, CloseHandle, CreateDirectory, …
  - Windows does not have signals, links to files, …, but has a large number of system calls for managing GUI

List of important syscalls

Posix	Win32	Description
Process Management		
Fork	CreateProcess	Clone current process
exec(ve)		Replace current process
wait(pid)	WaitForSingleObject	Wait for a child to terminate.
exit	ExitProcess	Terminate process & return status
File Management		
open	CreateFile	Open a file & return descriptor
close	CloseHandle	Close an open file
read	ReadFile	Read from file to buffer
write	WriteFile	Write from buffer to file
lseek	SetFilePointer	Move file pointer
stat	GetFileAttributesEx	Get status info
Directory and File System Management		
mkdir	CreateDirectory	Create new directory
rmdir	RemoveDirectory	Remove *empty* directory
link	(none)	Create a directory entry
unlink	DeleteFile	Remove a directory entry
mount	(none)	Mount a file system
umount	(none)	Unmount a file system
Miscellaneous		
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Change permissions on a file
kill	(none)	Send a signal to a process
time	GetLocalTime	Elapsed time since 1 jan 1970

A Few Important Posix/Unix/Linux and Win32 System Calls

OS Service Examples

- Services that need to be provided at kernel level
  - System calls: file open, close, read and write
  - Control the CPU so that users won't stuck by running

    ```
    while ( 1 ) ;
    ```
  - Protection:
    - Keep user programs from crashing OS
    - Keep user programs from crashing each other
- Services that can be provided at user level
  - Read time of the day

Is Any OS Complete?
(Criteria to Evaluate OS)

Portability

Security

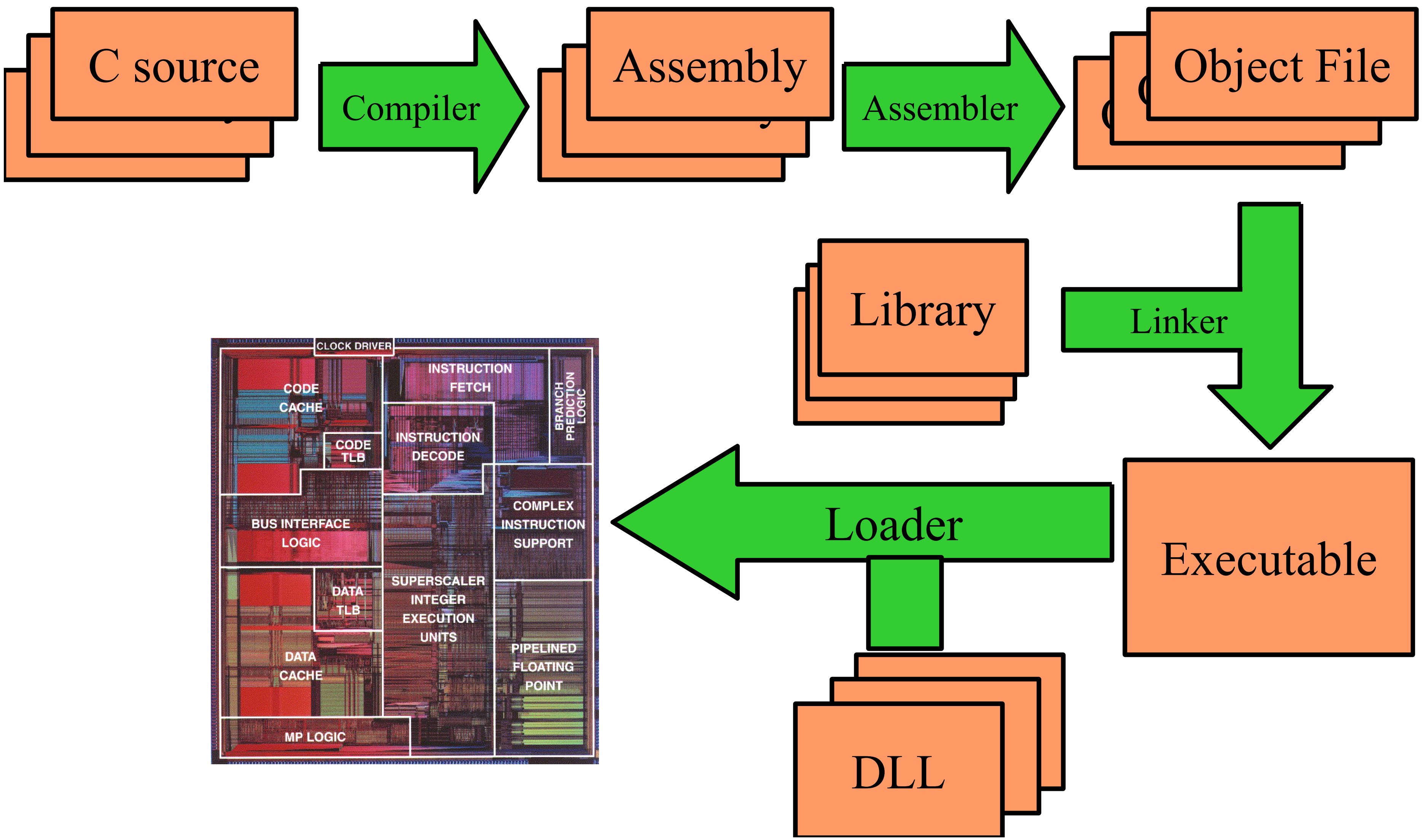
Fairness

Robustness

Efficiency

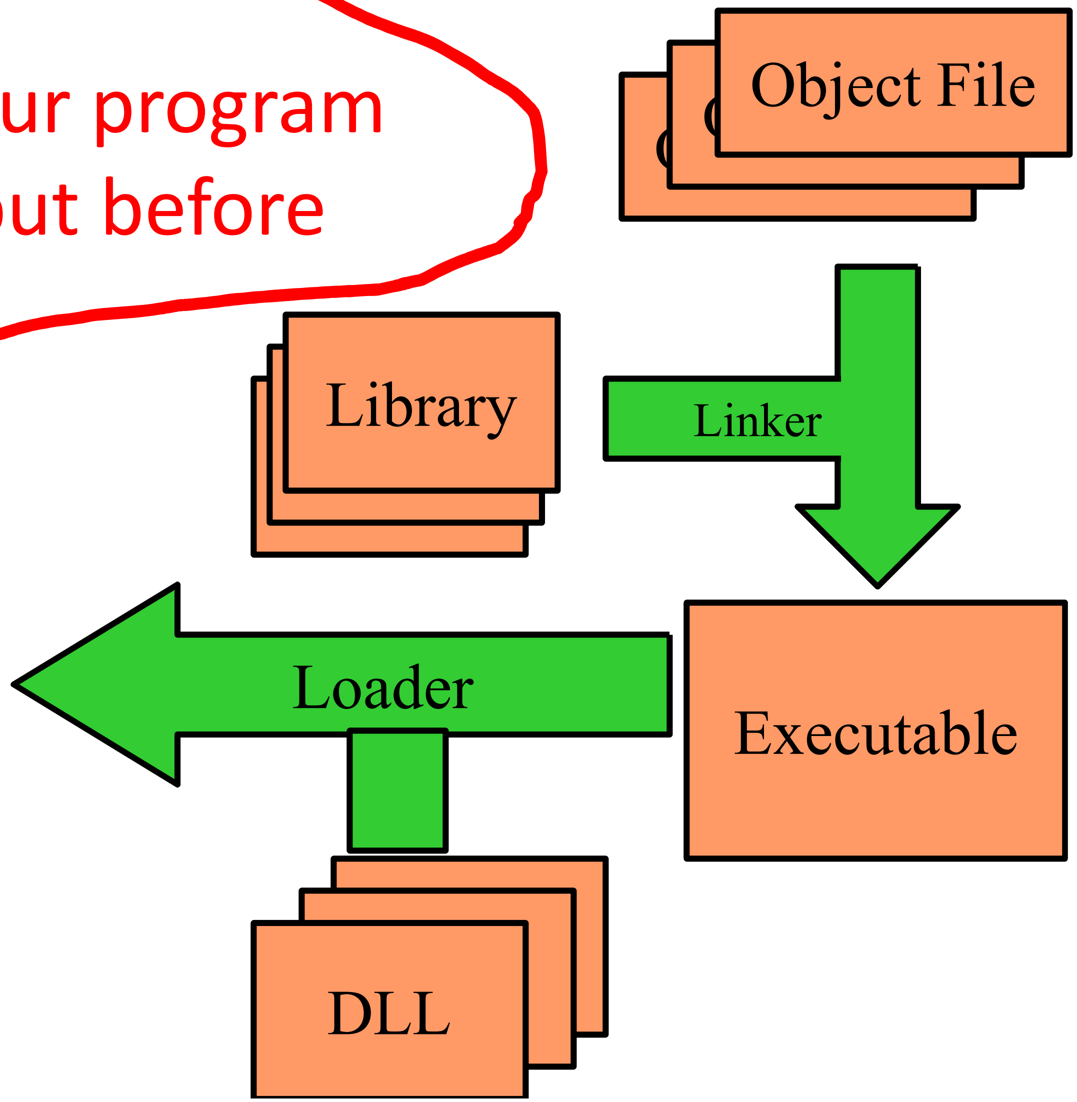
Interfaces

Source Code to Execution



Source Code to Execution

What happens to your program after it is compiled but before it can be executed?



The OS Expectation

- The OS expects executable files to have a specific format
  - *Header info*
    - Code locations and size
    - Data locations and size
  - Code & data
  - *Symbol Table*
    - List of names of things defined in your program and where they are defined
    - List of names of things defined elsewhere that are used by your program, and where they are used.

Example of Things

```
#include <stdio.h>
extern int errno;

int main () {

  printf ("hello,
  world\n")

  <check errno for
  errors>
}
```

- Symbol defined in your program and used elsewhere
  - `main`
- Symbol defined elsewhere and used by your program
  - `printf`
  - `errno`

Two Steps Operation: Parts of OS

Linking

- Stitches independently created object files into a single executable file (i.e., a.out)
- Resolves cross-file references to labels
- Listing symbols needing to be resolved by loader

Loading

- copying a program image from hard disk to the main memory in order to put the program in a ready-to-run state
- Maps addresses within file to physical memory addresses
- Resolves names of dynamic library items
- schedule program as a new process

Libraries (I)

- Programmers are expensive.
- Applications are more sophisticated.
  - Pop-down menus, streaming video, etc
- Application programmers rely more on library code to make high quality apps while reducing development time.
  - This means that most of the executable is library code

Libraries (II)

- A collection of subprograms
- Libraries are distinguished from executables in that they are not independent programs
- Libraries are "helper" code that provides services to some other programs
- Main advantages: reusability and modularity

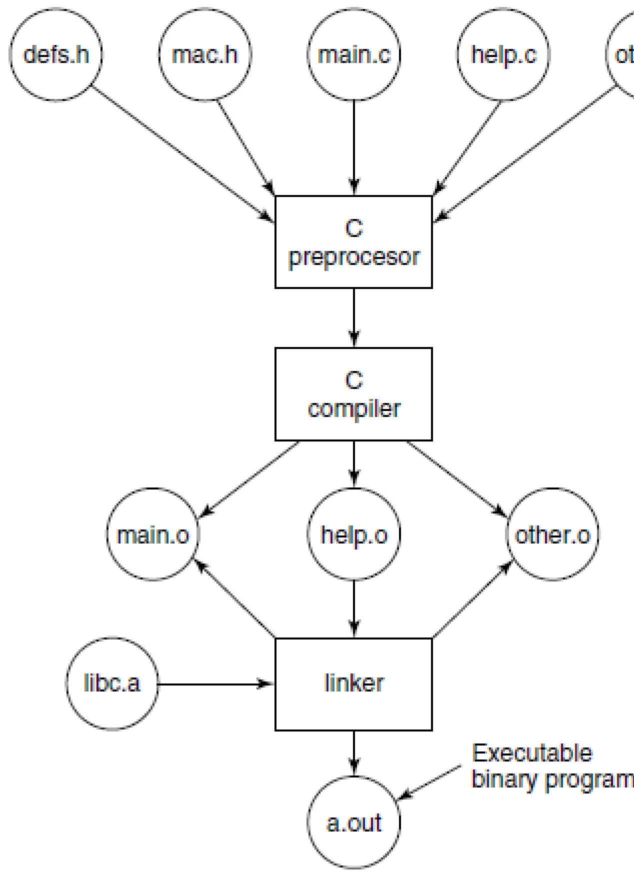
Static Libraries

- These libraries are stored on disk.
- Linker links only the libraries referenced by the program
- Main disadvantage: needs a lot of memory (for example, consider standard functions such as printf and scanf. They are used almost by every application. Now, if a system is running 50-100 processes, each process has its own copy of executable code for printf and scanf. This takes up significant space in the memory.)

Dynamic Link Libraries (Shared Libraries)

- Why not keep those shared library routines in memory and link at object file when needed? (DLLs)
- A shared library is an object module that can be loaded at run time at an arbitrary memory address, and it can be linked to by a program in memory.
- An application can request a dynamic library during execution
- Main advantage: saving memory
- Main disadvantage: ~10% performance hit

Large Programming Projects



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

A Bit About Relocation

- modifies the object program so that it can be loaded at an address different from the location originally specified
- The compiler and assembler (mistakenly) treat each module as if it will be loaded at location zero

(e.g. ***jump 120***
is used to indicate a jump to location 120 of the current module)

A Bit About Relocation

- To convert this relative address to an absolute address, the linker adds the base address of the module to the relative address.
- The base address is the address at which this module will be loaded.

Example: Module A is to be loaded starting at location 2300 and contains the instruction

jump 120

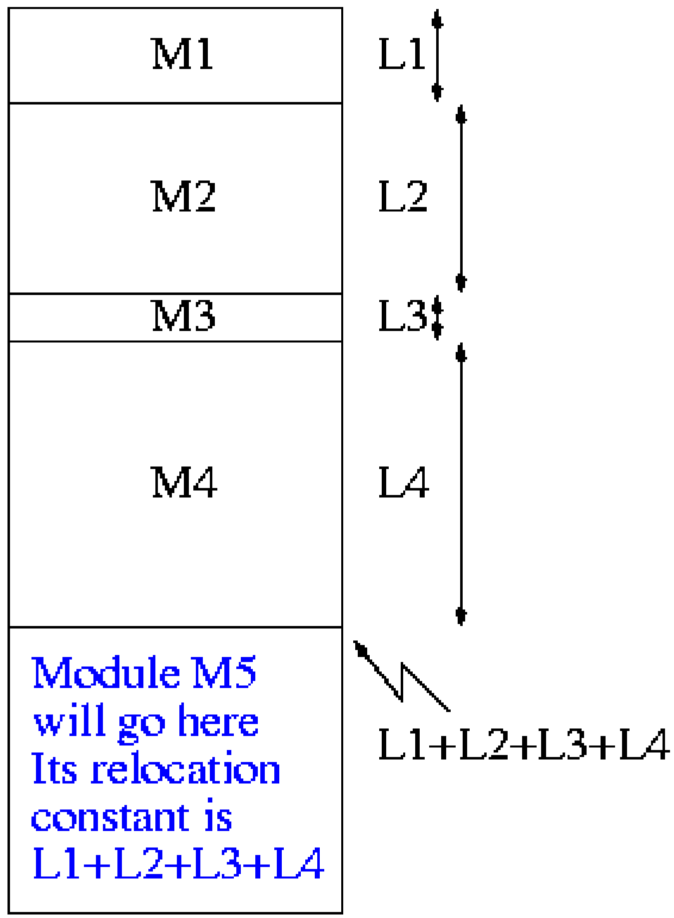
The linker changes this instruction to

jump 2420

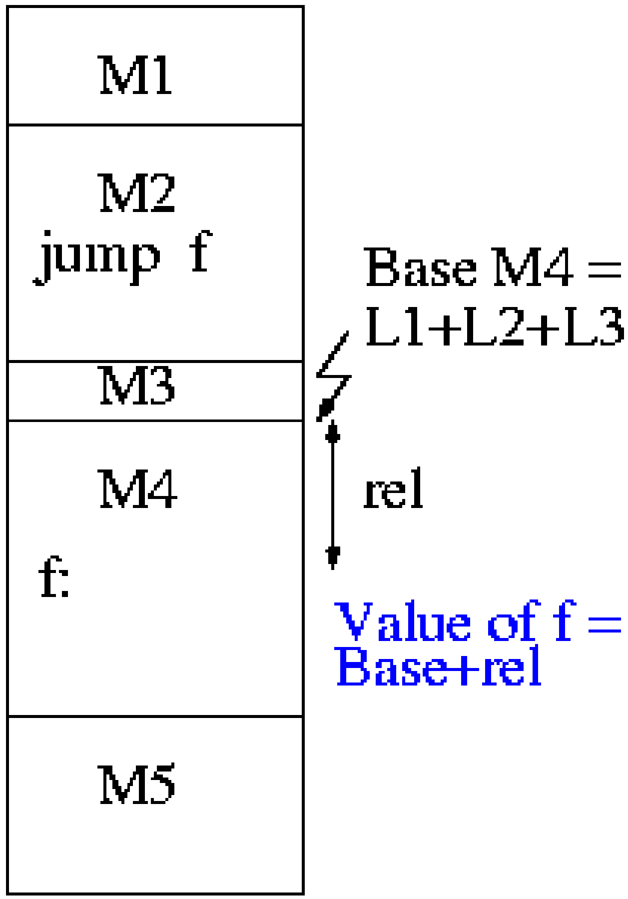
A Bit About Relocation

- How does the linker know that Module A is to be loaded starting at location 2300?
  - It processes the modules one at a time. The first module is to be loaded at location zero. So relocating the first module is trivial (adding zero). We say that the relocation constant is zero.
  - After processing the first module, the linker knows its length (say that length is L1).
  - Hence the next module is to be loaded starting at L1, i.e., the relocation constant is L1.
  - In general the linker keeps the sum of the lengths of all the modules it has already processed; this sum is the relocation constant for the next module.

A Bit About Relocation

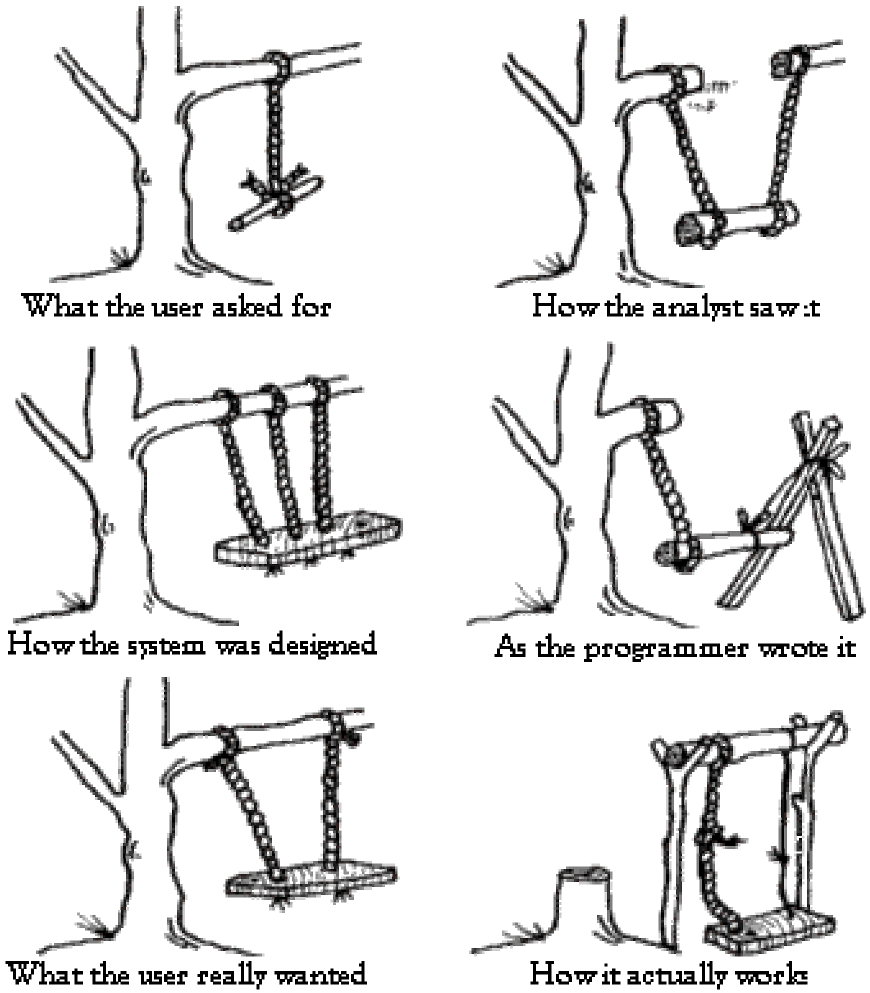


A Bit About Relocation

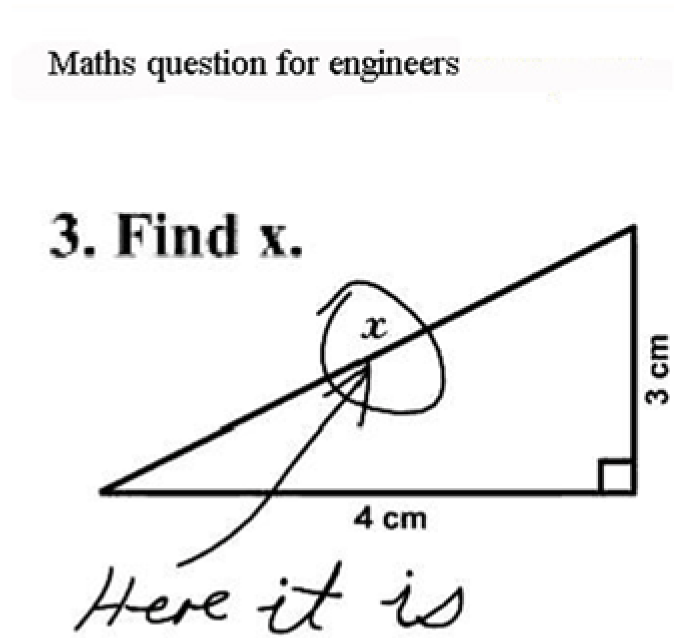


LAB assignment #1

Lab instructions



Possible solutions



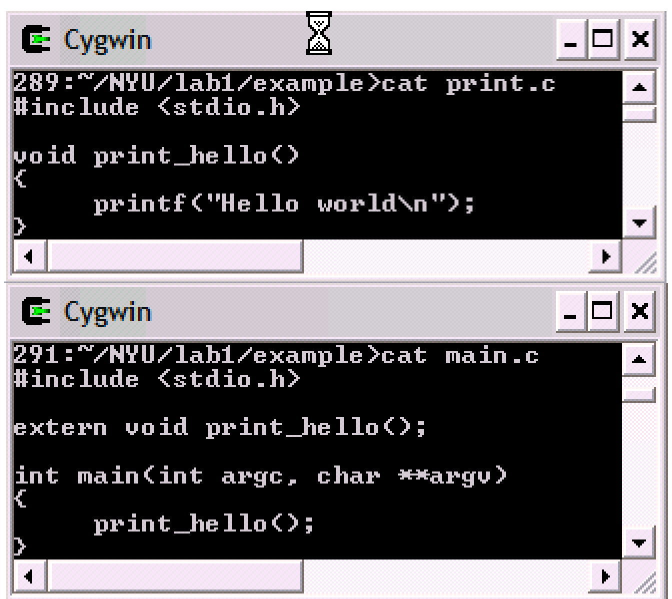
Read and Understand the instructions

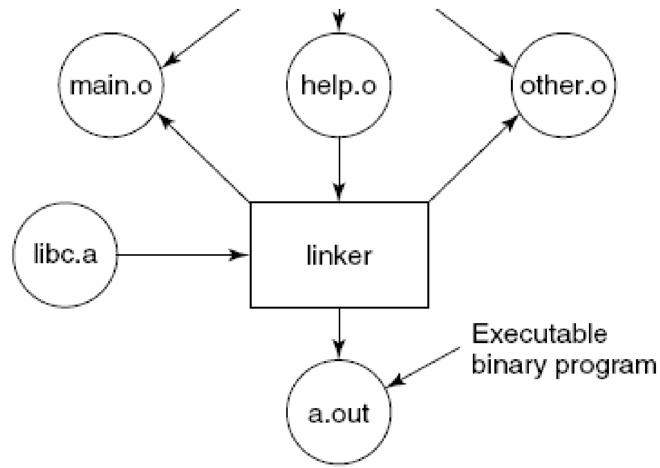
Solution

LAB #1: Write a Linker

- Link "==merge" together multiple parts of a program

- What problem is solved?
  - External references need to be resolved
  - Module relative addressing needs to be fixed





LAB #1: Write a Linker

- Simplified module specification
  - List of symbols defined and their value by module
  - List of symbols used in module (including external)
  - List of "instructions"

of (symbol,value) defined: xy=2

of symbols used

Module-1

Module-3

```
1 xy 2
2 z xy
5 R 1004  I 5678  E 2000  R 8002  E 7001
0
1 z
6 R 8001  E 1000  E 1000  E 3000  R 1002  A 1010
0
1 z
2 R 5001  E 4000
1 z 2
2 xy z
3 A 8000  E 1001  E 2000
```

instruction

Addressing

OpCode

Address/Operand

Addressing

I: Immediate
R: Relative
A: Absolute
E: External

Lab #1: Write a Linker

input

```
1 xy 2
2 z xy
5 R 1004  I 5678  E 2000  R 8002  E 7001
0
1 z
6 R 8001  E 1000  E 1000  E 3000  R 1002  A 1010
0
1 z
2 R 5001  E 4000
1 z 2
2 xy z
3 A 8000  E 1001  E 2000
```

Fancy Output (not req)

```
Symbol Table
xy=2
z=15
Memory Map
+0
0:       R 1004         1004+0 =  1004
1:       I 5678                   5678
2: xy:   E 2000 ->z               2015
3:       R 8002         8002+0 =  8002
4:       E 7001 ->xy              7002
+5
0:       R 8001         8001+5 =  8006
1:       E 1000 ->z               1015
2:       E 1000 ->z               1015
3:       E 3000 ->z               3015
4:       R 1002         1002+5 =  1007
5:       A 1010                   1010
+11
0:       R 5001         5001+11=  5012
1:       E 4000 ->z               4015
+13
0:       A 8000                   8000
1:       E 1001 ->z               1015
2 z:     E 2000 ->xy              2002
```

Required output

```
Symbol Table
xy=2
z=15
Memory Map
000: 1004
001: 5678
002: 2015
003: 8002
004: 7002
005: 8006
006: 1015
007: 1015
008: 3015
009: 1007
010: 1010
011: 5012
012: 4015
013: 8000
014: 1015
015: 2002
```