CSCI-GA.2250-001

Operating Systems

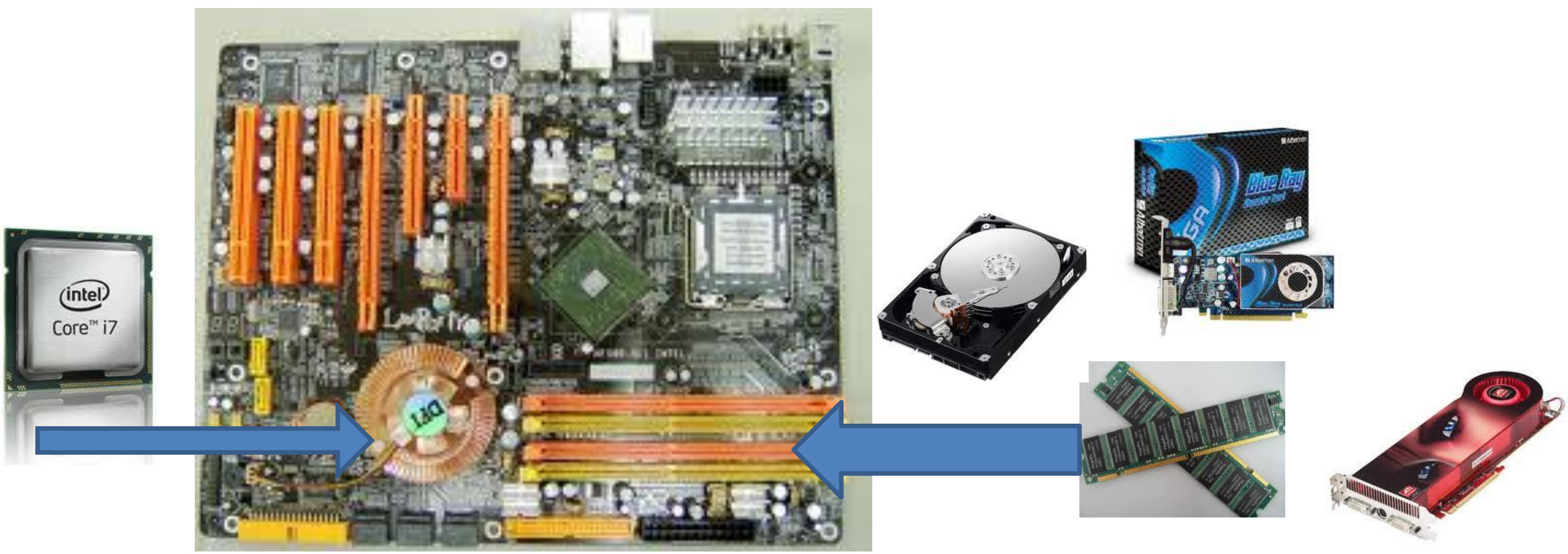
Introduction

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

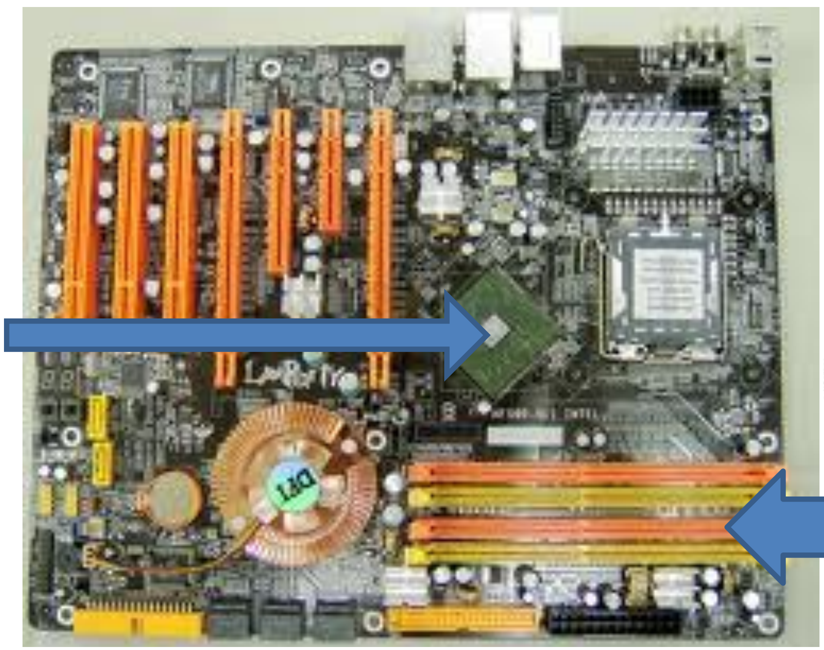


Media Player

emails

Games

Word Processing



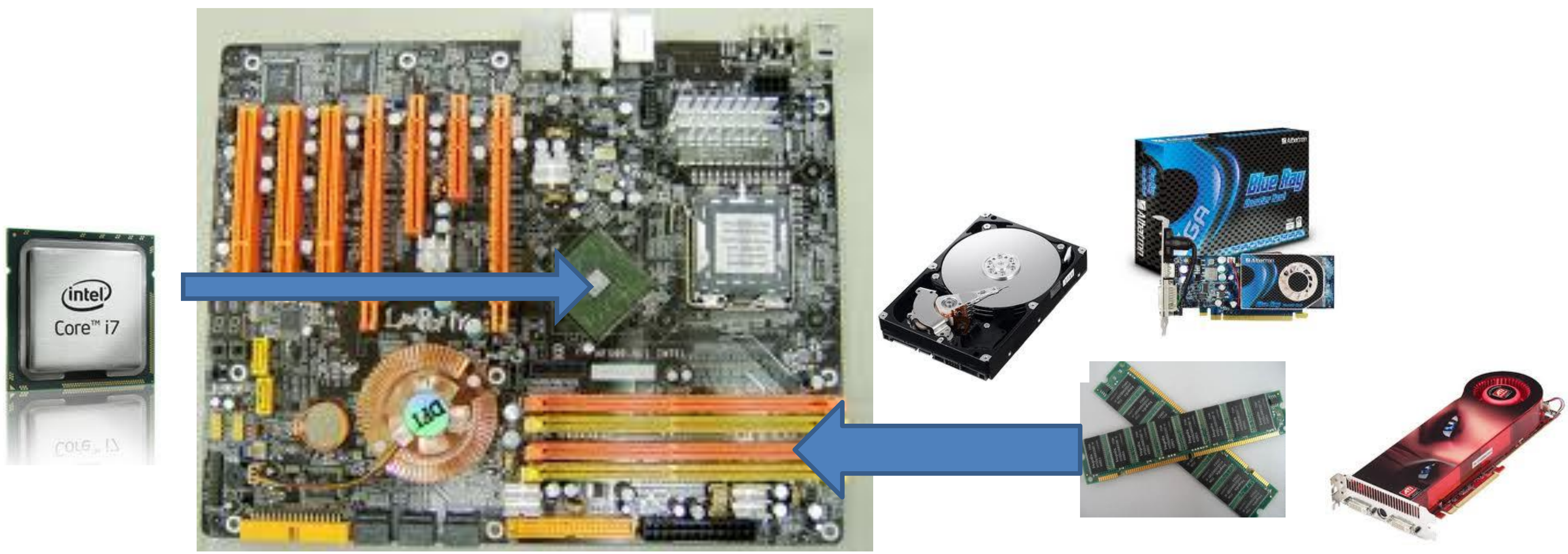
Media Player

emails

Games

Word Processing

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



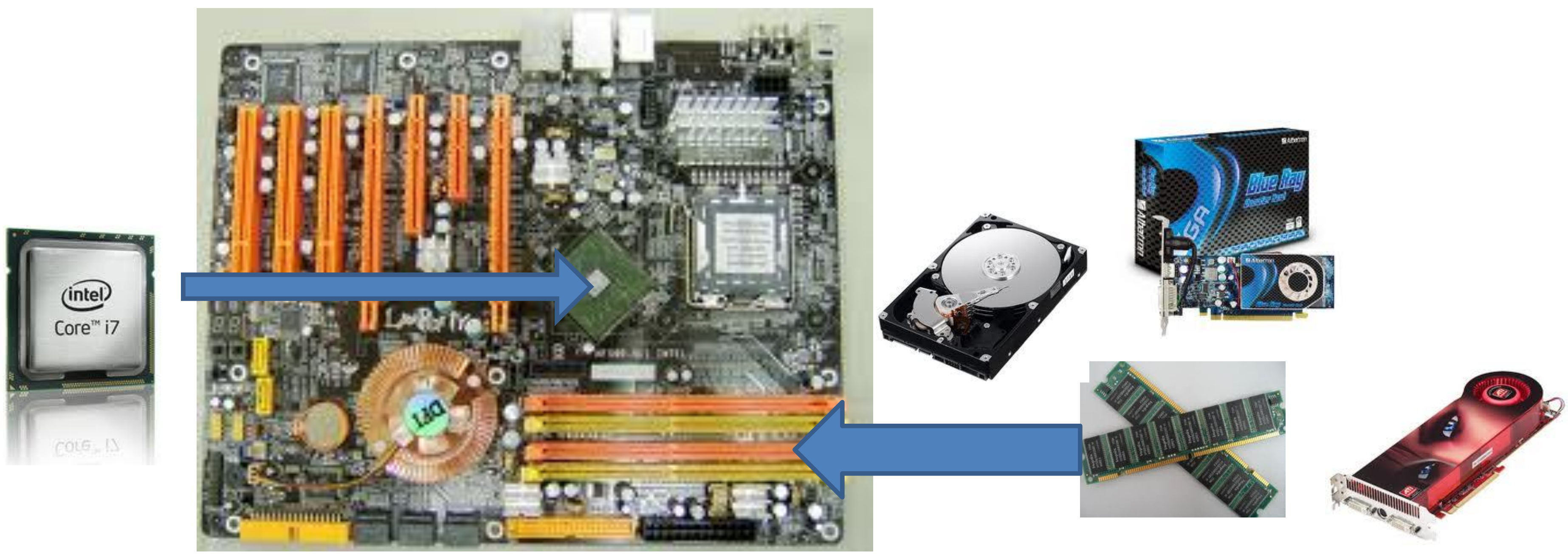
Media Player

emails

Games

Word Processing

Operating System



Components of a Modern Computer

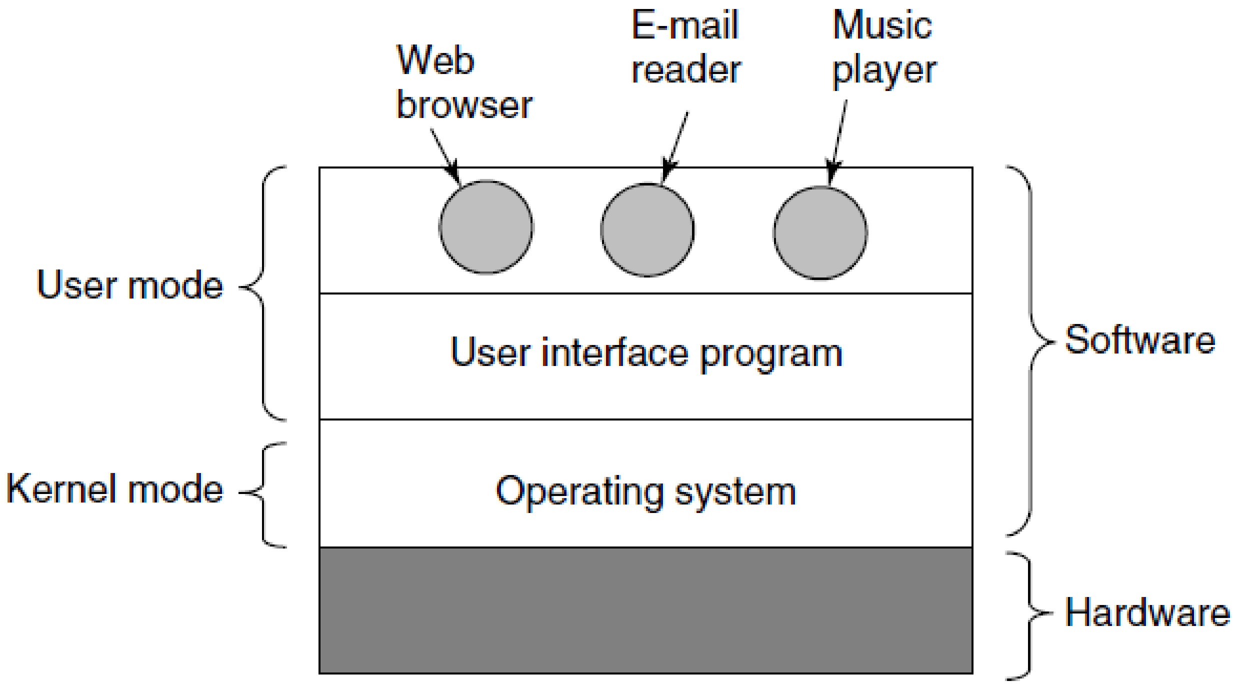
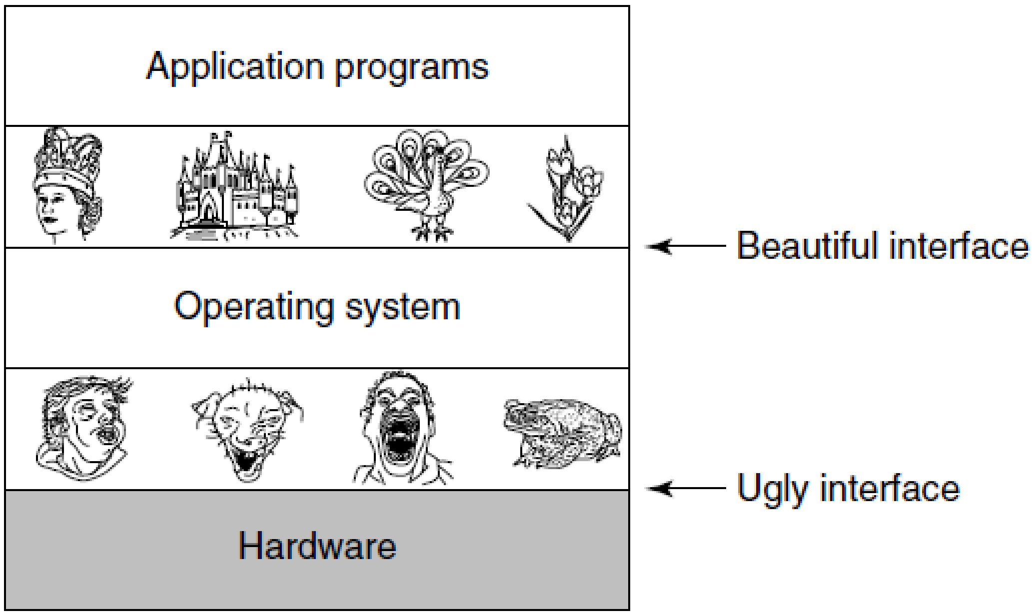


Figure 1-1. Where the operating system fits in.

The Operating System as an Extended Machine



APIs
Documented
Man(ual)-pages

Assembler
Bitmapping
Hardware

Operating systems turn ugly hardware into beautiful abstractions (arguable).

The Operating System as a Resource Manager

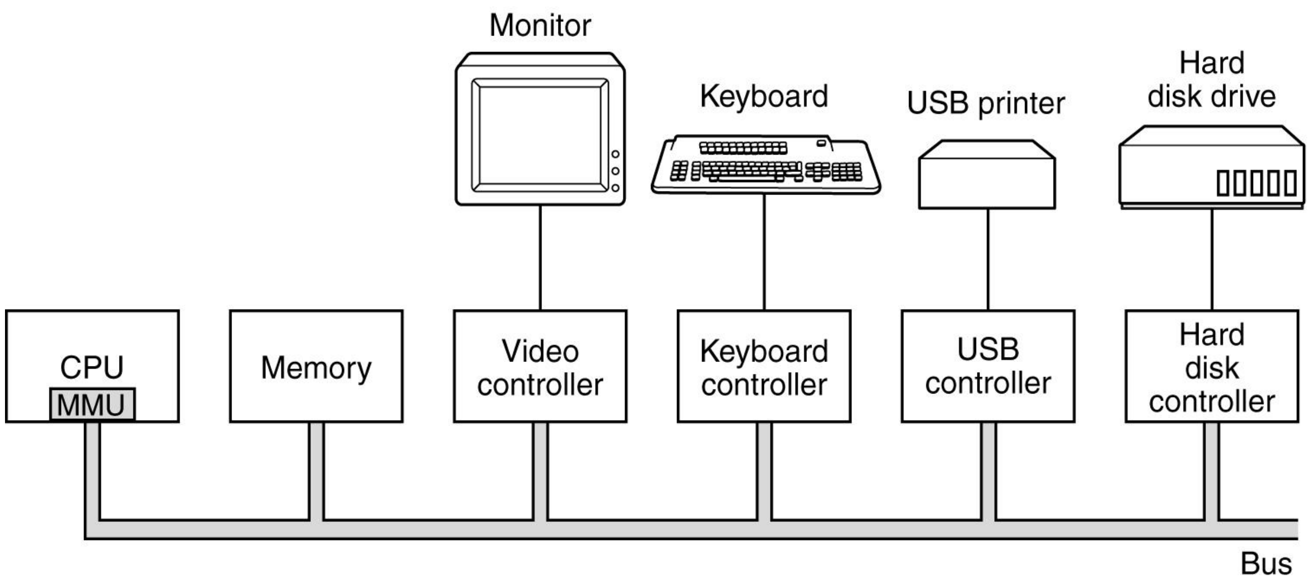
- Top down view
  - Provide abstractions to application programs
- Bottom up view
  - Manage pieces of complex systems (hardware and events)
- Alternative view
  - Provide orderly, controlled allocation of resources

The Two Main Tasks of OS

- Provide programmers (and programs) a clean set of abstract resources and services to manipulate these resources

- Manage the hardware resources

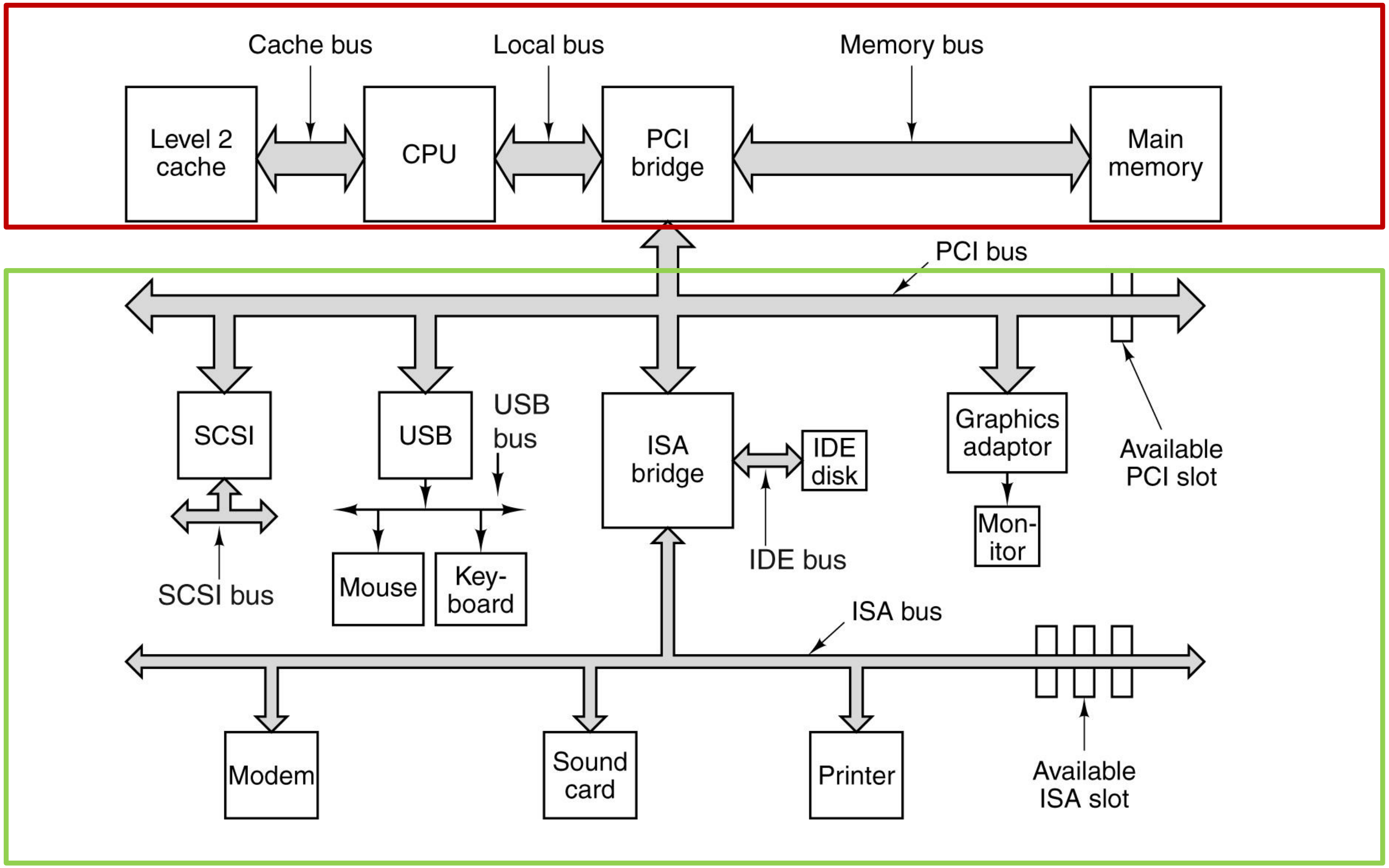
A Glimpse on Hardware



Simplified view: System Components are interlinked through a shared bus and communicate over that bus.

This is done through bus transactions
(e.g. load/store to/from memroy, interprocessor notifications, device accesses, ...)

A Glimpse on Hardware



Reality Check:

In reality there are many buses between the components one set relates to memory accesses (load/store) and one to I/O subsystems

A few conventions

2^0	**=**	**1**	**2^{16}**	**=**	**65,536**
2^1	=	2	2^{17}	=	131,072
2^2	=	4	2^{18}	=	262,144
2^3	=	8	2^{19}	=	524,288
2^4	**=**	**16**	**2^{20}**	**=**	**1,048,576**
2^5	=	32	2^{21}	=	2,097,152
2^6	=	64	2^{22}	=	4,194,304
2^7	=	128	2^{23}	=	8,388,608
2^8	**=**	**256**	**2^{24}**	**=**	**16,777,216**
2^9	=	512	2^{25}	=	33,554,432
2^{10}	=	1,024	2^{26}	=	67,108,864
2^{11}	=	2,048	2^{27}	=	134,217,728
2^{12}	**=**	**4,096**	**2^{28}**	**=**	**268,435,456**
2^{13}	=	8,192	2^{29}	=	536,870,912
2^{14}	=	16,384	2^{30}	=	1,073,741,824
2^{15}	=	32,768	2^{31}	=	2,147,483,648
			2^{32}	**=**	**4,294,967,296**

- Computer Scientists think in 2^N
- Remember a few tricks
- Know:
  2^0 .. 2^9
  2^10 ~10^3 = 1KB

- and the rest is easy (e.g.):
  - 2^20 ~10^6 = 1MB
  - 2^30 ~10^9 = 1GB
  - 2^32 = 2^(30+2) = 4GB
  - 2^16 = 2^(10+6) = 64KB

Expected you can do 2^N math on the spot

A few assembler conventions used

- Occasionally we need to use some assembler notation, I utilize a general fictious notation that should be easily understood
- Loads and stores:
  - basic means to obtain a data item from/to memory
  - ldw r3,addr   for loading a word of memory (addr) into a register (r3) (will cover that in a bit)
  - stw r3,addr   for storing a value in register (r3) back to memory (addr)
- Arithmetic operations:
  - takes operands and performs basic operations
  - add r3, r4, r5 (r3 = r4+r5)

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

Services

- Abstraction
- Simplification
- Convenience
- Standardization

CONTAINER

Makes computer usage simpler

What Is an OS?

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

Finite resources

Competing demands

Examples:

- CPU
- Memory
- Disk
- Network

Government

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?

Government

Social security

Resources

- Allocation
- Protection
- Reclamation
- Virtualization

illusion of infinite private resources

Memory versus disk

Timeshared CPU

More extreme cases possible (& exist)

Operating System

- OS (kernel) is really just a program that runs with special privileges to implement the features of allocation, protection, reclamation and virtualization

  <u>and</u>

  the services that are structured on top of it.

Booting Sequence

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

POST (Power On Self Test)

- BIOS determines boot Device

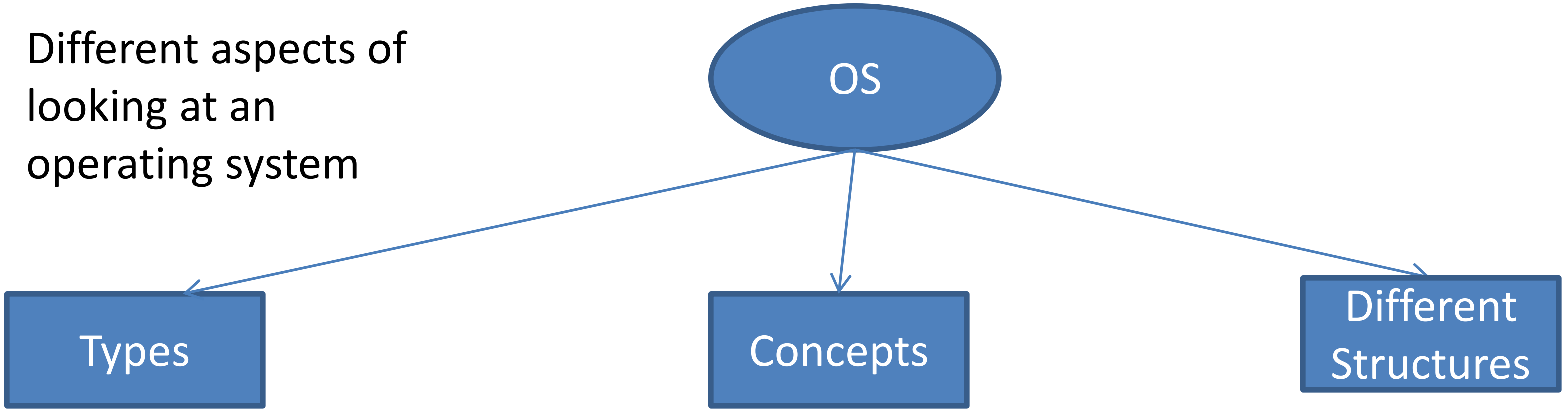
- The first sector in boot device is read into memory and executed to determine active partition

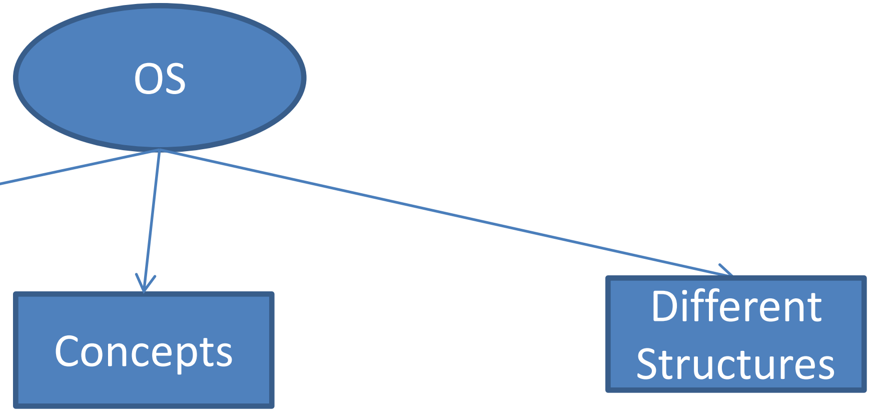
- Secondary boot loader is loaded from that partition.

- This loaders loads the OS from the active partition and starts it.

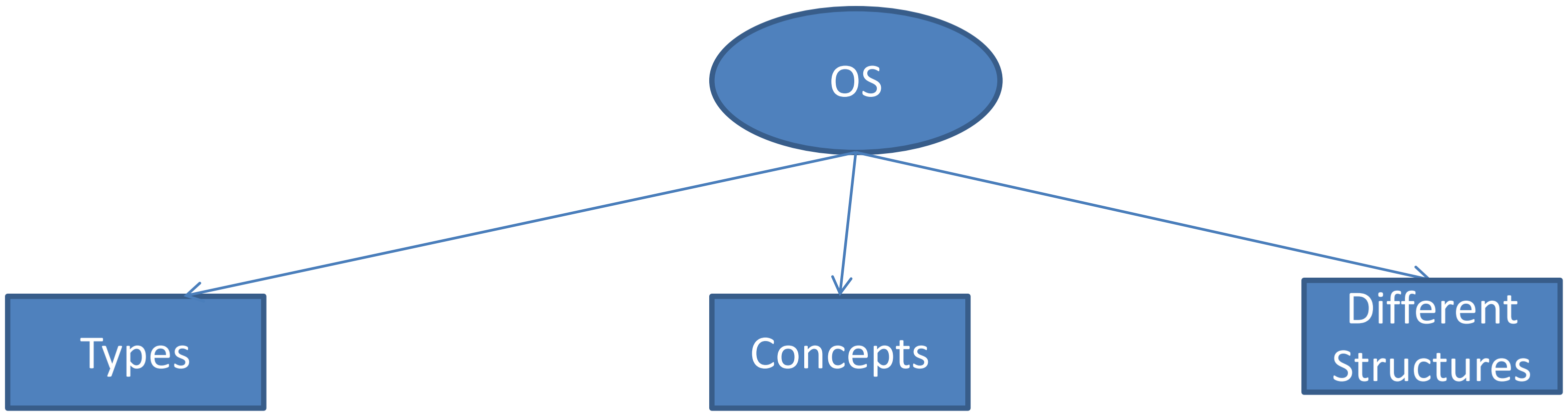
- BIOS: Basic Input/Output System (till ~2010)
- e.g. UEFI: Unified Extensible Firmware Interface (**UEFI**) is a specification for a software program that connects a computer's firmware to its operating system (OS) and functions as nowadays BIOS

Different aspects of looking at an operating system



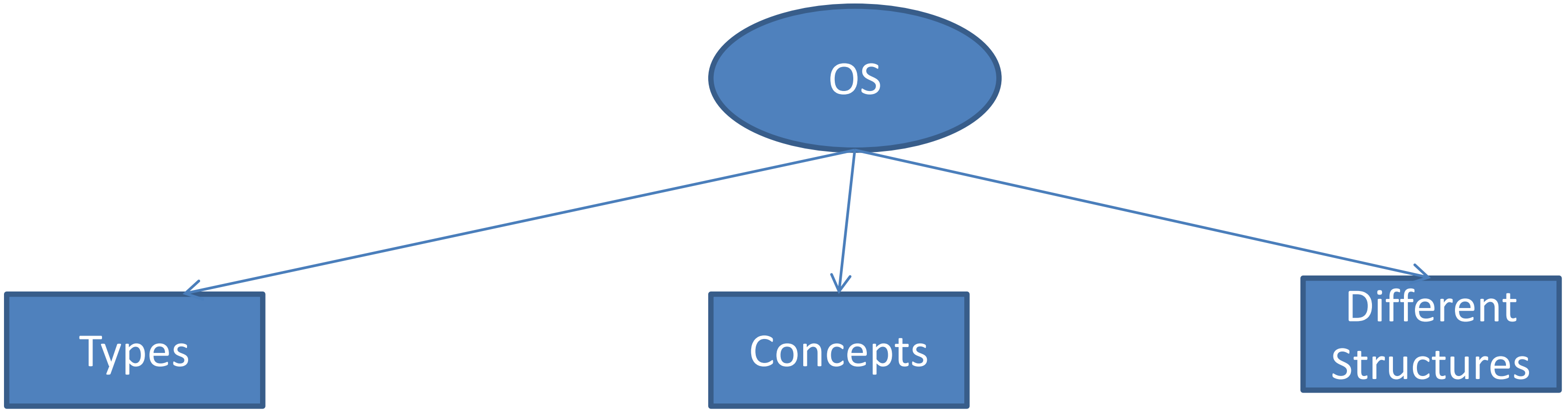


- **Mainframe/supercomputer OS**
    - **batch**
    - **transaction processing**
    - **timesharing**
    - **e.g. OS/390**
- **Server OS**
- **Multiprocessor OS**
- **PC OS**
- **Embedded OS**
- **Sensor node OS**
- **RTOS**
- **Smart card OS**



- 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

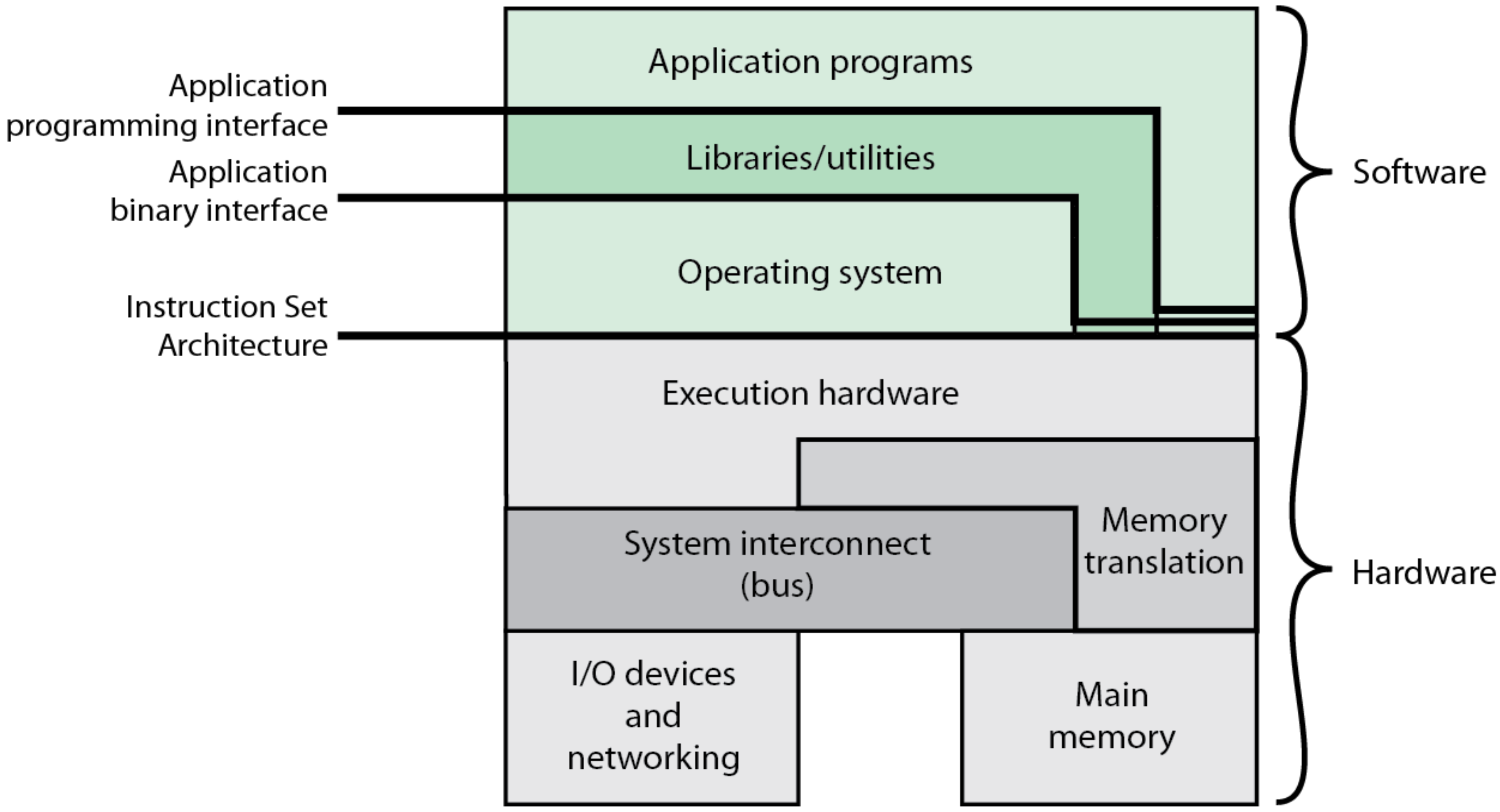
Main objectives of an OS:

- 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.*”
  - **Adlai 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
- Fifth generation 2005 – present
  - hand-held devices, sensors

History of Operating Systems (1945-55)

- Programming and Control 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

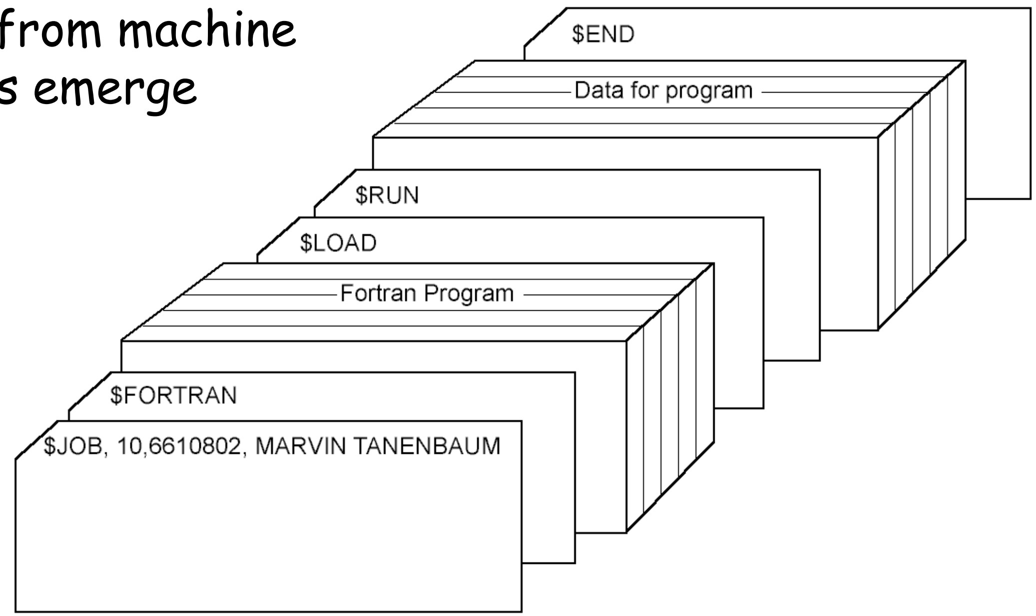
Source: wikipedia

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?

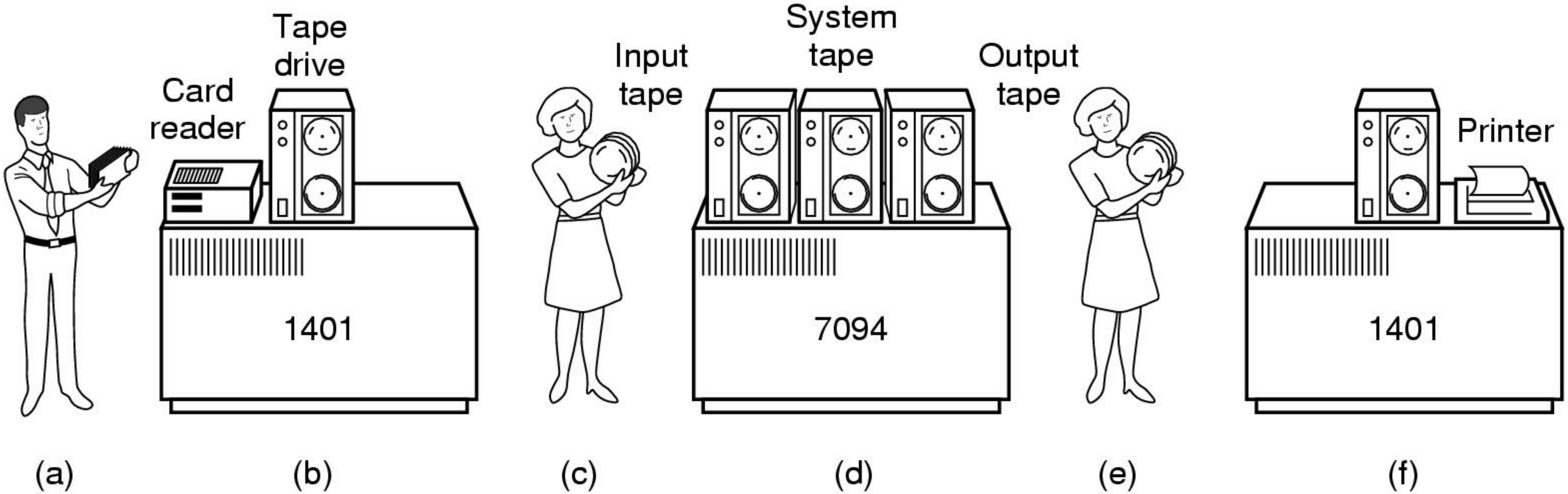
History of Operating Systems (1955-65)

- 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

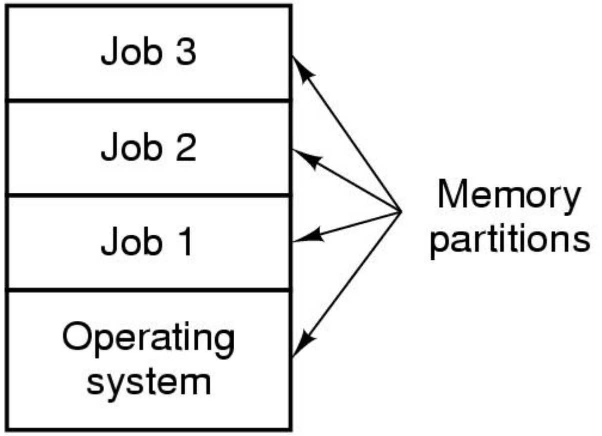
History of Operating System (1955-65)



Early batch system
- bring cards to 1401
- read cards to tape
- put tape on 7094 which does computing
- put tape on 1401 which prints output

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

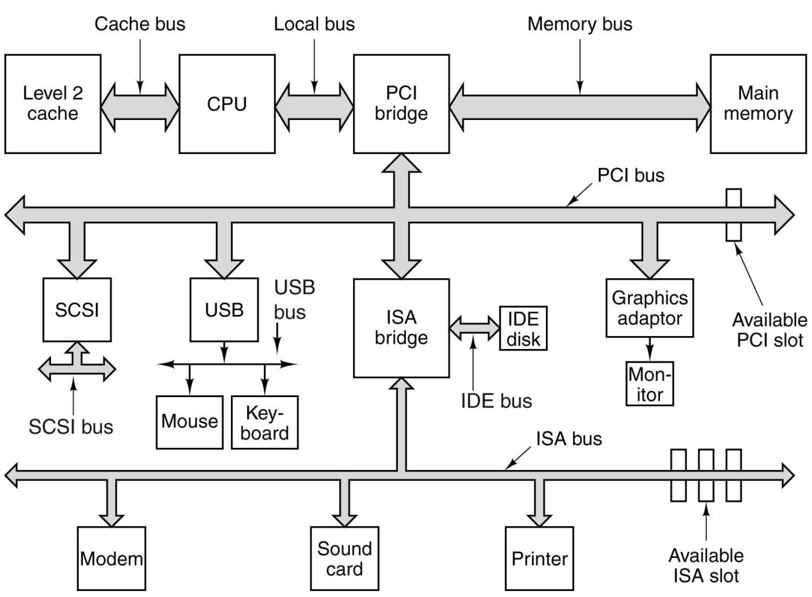
The Operating System Jungle / Zoo (1980-present)

- 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



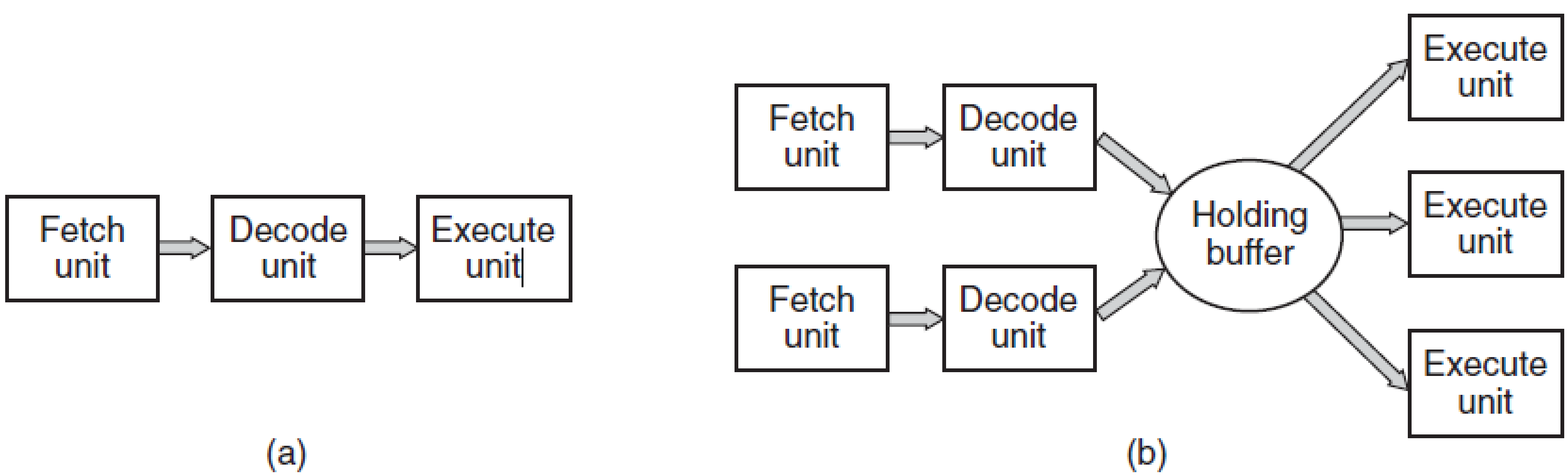
Processors

- Each CPU has a specific set of instructions
  - ISA (Instruction Set Architecture) largely epitomized in the assembler
    - 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) A three-stage pipeline. (b) A superscalar CPU.

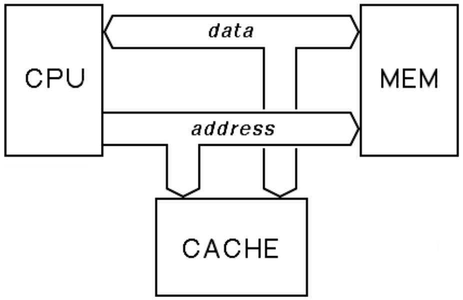
Memory-Storage Hierarchy

Latency		Capacity
1ns	Registers	32+32
10ns	Cache	8KB – 2MB (L1 – L3)
100ns	Main memory	GBs - TBs
10msec	Magnetic disk	10s * TBs
10secs	Magnetic tape	500s * TBs

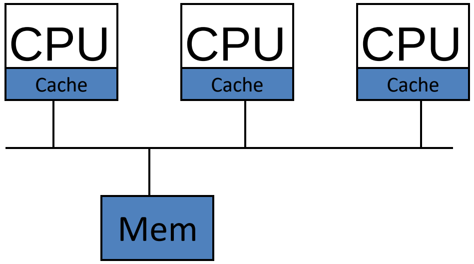
- Other metrics:
  - Bandwidth (e.g. MemBandwidth 30GB/s → 200GB/s, Disk ~70-200MB/s)
- What can an OS do to increase its "performance"
  - Active management where to place data !!!

CPU Caches

- Principle:
  - Data/Instruction that were recently used are "likely" used again in short period
  - Caching is principle used in "many" subsystems (I/O, filesystems, …) [hardware and software]

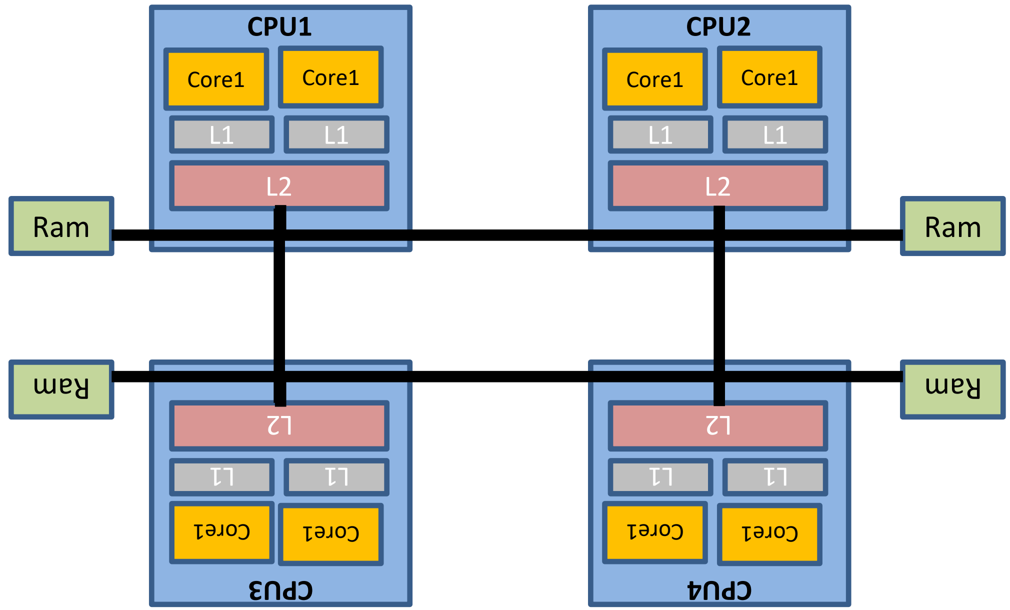


- Cache hit:
  - no need to access memory
- Cache miss:
  - data obtained from mem, possibly update cache
- Issues
  - Operation MUST be correct (see coherency)
  - Cache management and Coherency for Memory done in hardware
  - Data can be in read state in multiple caches but only in one cache when in write state



Example of real cache/memory access times

- Modern systems have multiple CPUs with their own attached memory and multiple level of caches.
- Non-uniform memory access.

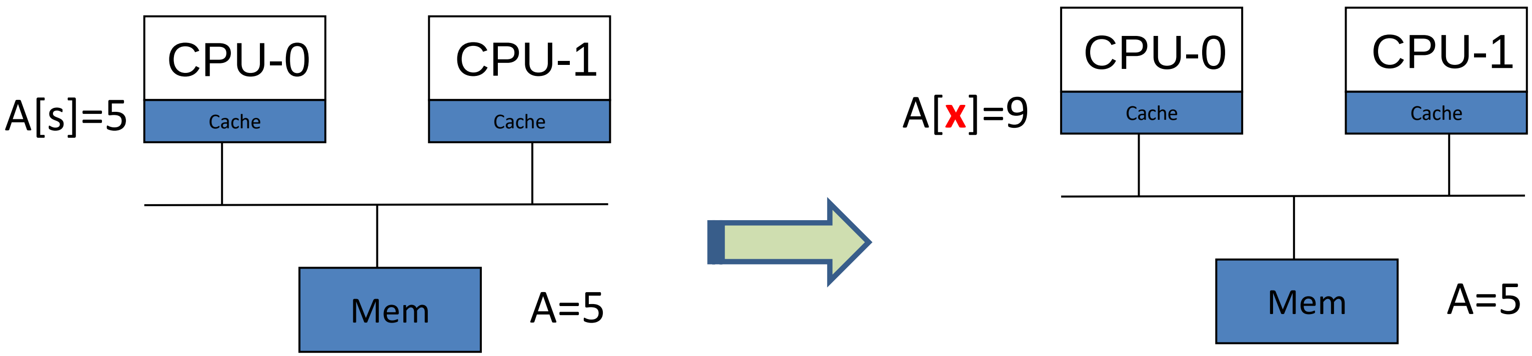



```
Core i7 Xeon 5500 Series Data Source Latency (approximate)               [Pg. 22]

local  L1 CACHE hit,                              ~4 cycles (   2.1 -   1.2 ns )
local  L2 CACHE hit,                             ~10 cycles (   5.3 -   3.0 ns )
local  L3 CACHE hit, line unshared               ~40 cycles (  21.4 - 12.0 ns )
local  L3 CACHE hit, shared line in another core ~65 cycles (  34.8 - 19.5 ns )
local  L3 CACHE hit, modified in another core    ~75 cycles (  40.2 - 22.5 ns )

remote L3 CACHE (Ref: Fig.1 [Pg. 5])        ~100-300 cycles ( 160.7 - 30.0 ns )

local  DRAM                                                   ~60 ns
remote DRAM                                                  ~100 ns
```

http://software.intel.com/en-us/forums/showthread.php?t=77966

Cache Coherency

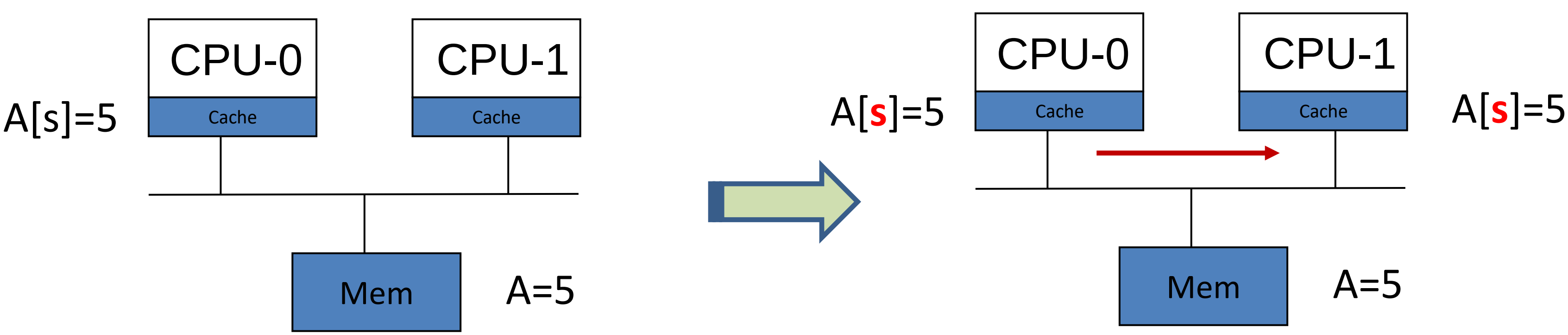
Scenario 1



C0 loads A
(A on C0 in shared mode [s])

C0 stores A=9
(A on C0 is now in exclusive mode [x])

Scenario 2

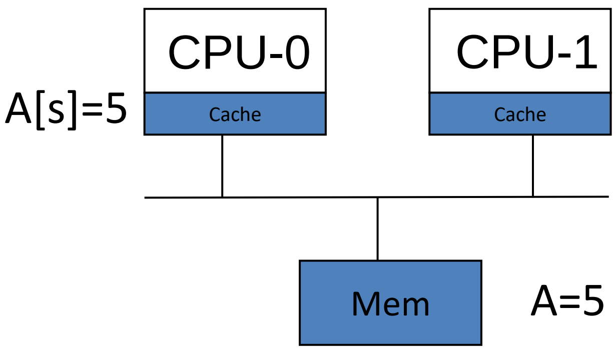


C0 loads A
(A on C0 in shared mode [s])

C1 now loads A (cache line is copied)
(A on C0 and C1 in shared mode [s])

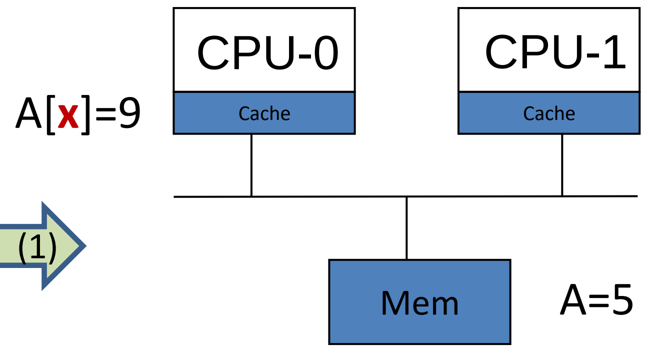
Cache Coherency

Scenario 3



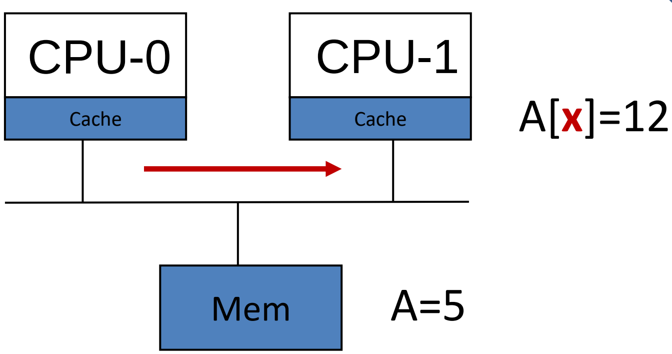
C0 loads A
(A on C0 in shared mode [s])

(1)



C0 stores A=9
(A on C0 is now in exclusive mode [x])

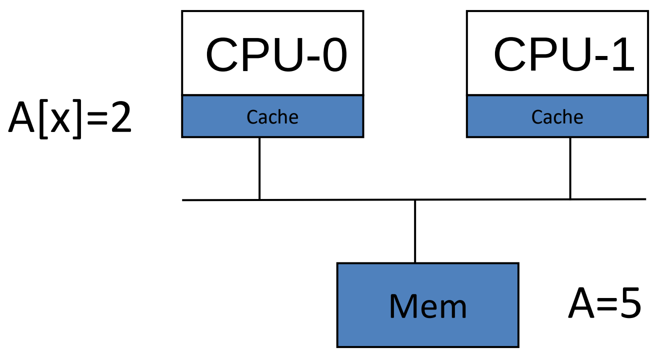
(2)



C1 stores A=12
(A on C1 in exclusive mode)
cache line is just moved

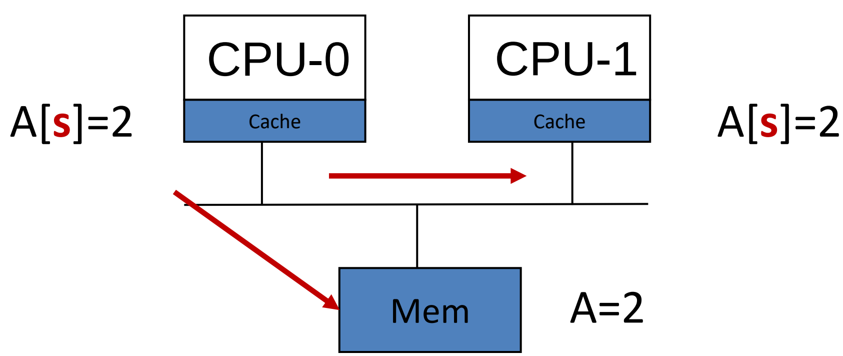
Cache Coherency

Scenario 4



C1 loads A

Forces C0 to write back A to mem
(or lowest level of shared cache)
and put A into shared mode for C0/C1

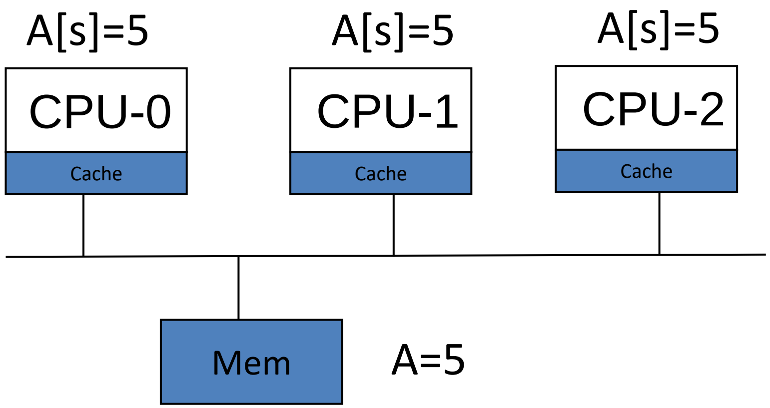


Imperative at all times

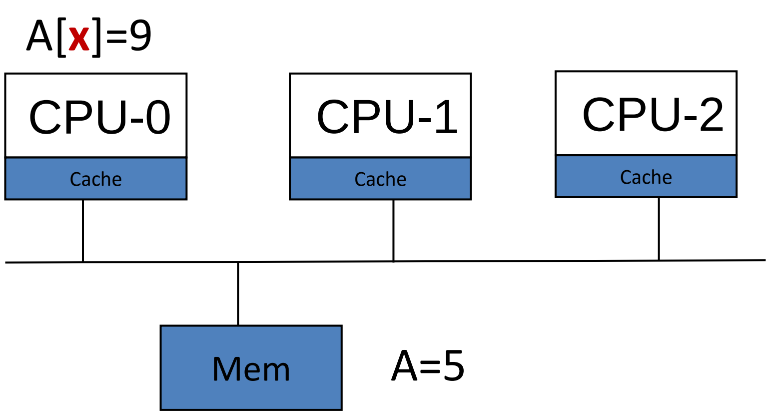
- **A can be in [s] across several CPUs**
- **A can be at most in ONE CPU in [x] and nowhere else in [s]**

Cache Coherency

Scenario 5



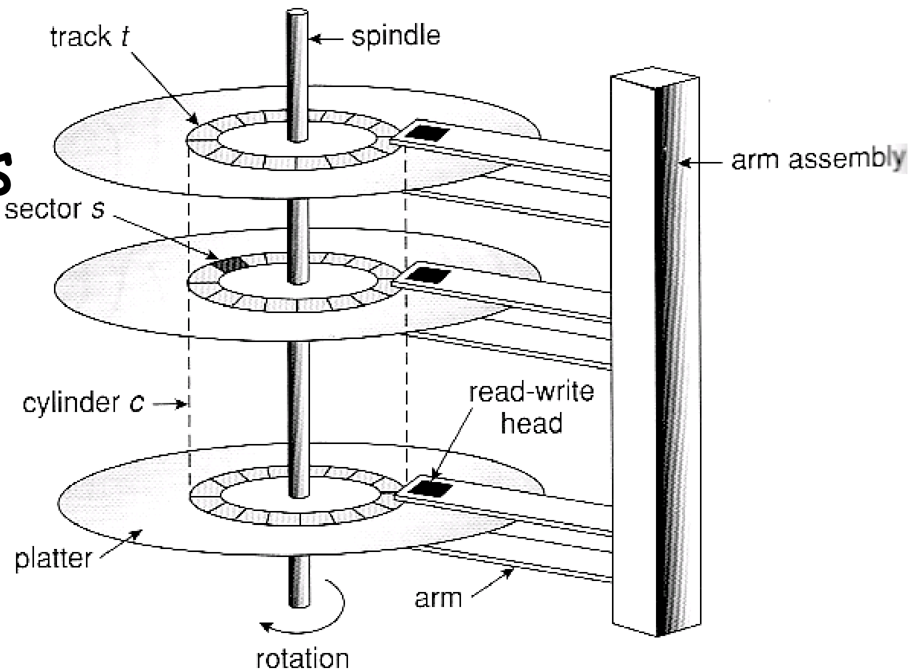
Both thread C0, C1, C2 load A
(A on C0, C1, C2 in shared mode)



Now C0 sets A=9
All cache lines on other cpus must be invalidated and A on C0 in [x]

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

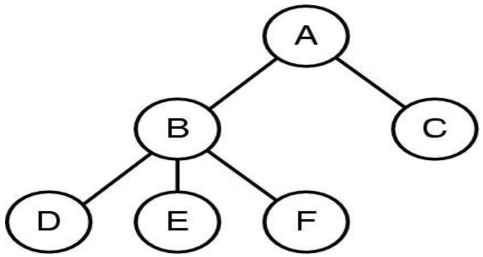
  - Power Management

OS Major Components

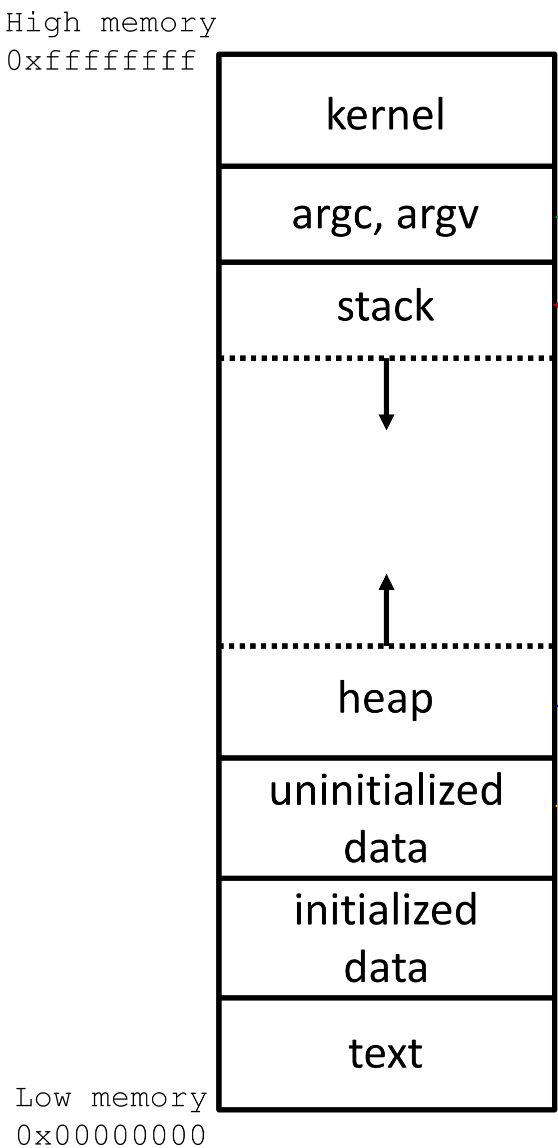
- Process and thread management
- Resource management
  - CPU
  - Memory
  - Device (I/O)
- File system
- Bootstrapping

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



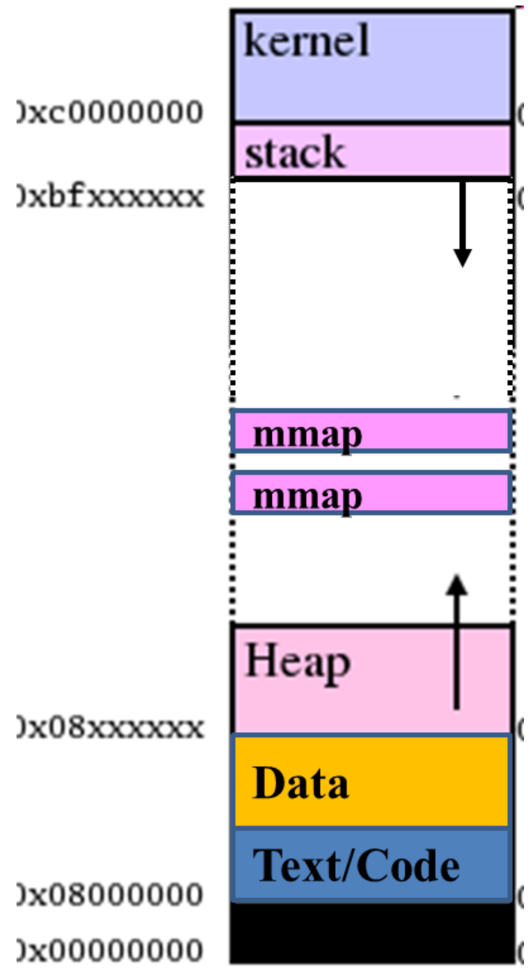
Address Space:
A view of program layout




```c
#include <stdio.h>
#include <stdlib.h>

#define NUMS (4)

int a;
int b = 2;
int x;
int y = 3;


int main(int argc, char* argv[])
{
    int *values;
    int i;

    values = (int*) malloc(NUMS*sizeof(int));

    for (i=0 ; i<NUMS; i++)
        values[i] = i;

    return 0;
}
```

Address Space

- Processes don't ld/st with physical addresses
- Address space defines a "private" addressing concept
  - ➔ requires form of address virtualization (will be covered in memory management)
  - ➔ We distinguish virtual address vs physical address
- 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
- Even the kernel has its own address space but typically maps the virt-address -> phys-addr but still goes through address translation



CPU Execution Modes

- Two common modes of CPU to enable protection
  - <u>User mode</u>

    - aka <u>unprivileged</u> or <u>problem mode</u>

    - only capable to running a subset of instructions (almost all)
      limits (~excludes) user from accessing critical resources

  - <u>Kernel mode</u>

    - aka <u>privileged</u> or <u>supervisor</u>

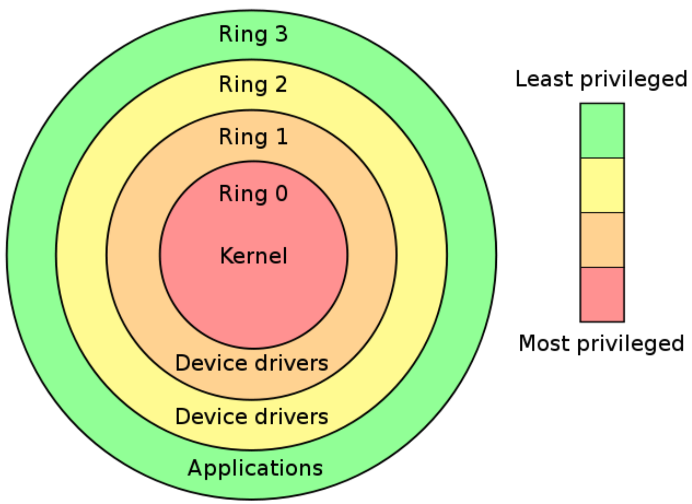
    - access to all instruction

    - enables the kernel to provide core services in a protected way

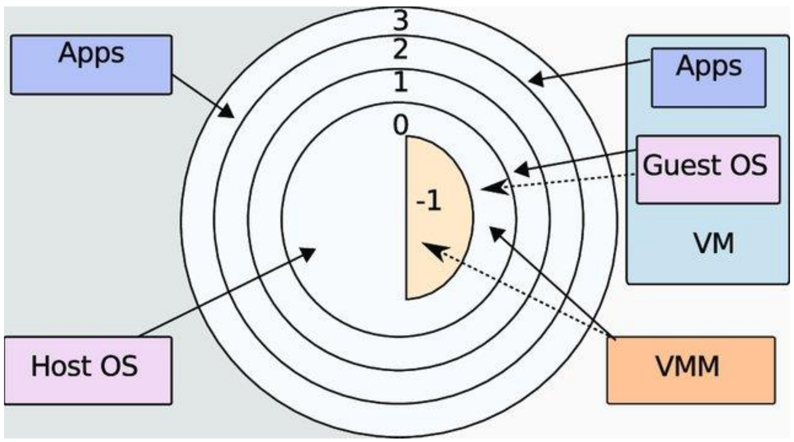
CPU Execution Modes (more general)

- Architectures sometimes provide more extended modes.
- Often referred to as "protection rings"
- Similar concept of privilege limitation on instructions and resource access resulting in traps

x86 protection rings



x86 protection rings with virtualization



CPU Execution Modes Switches

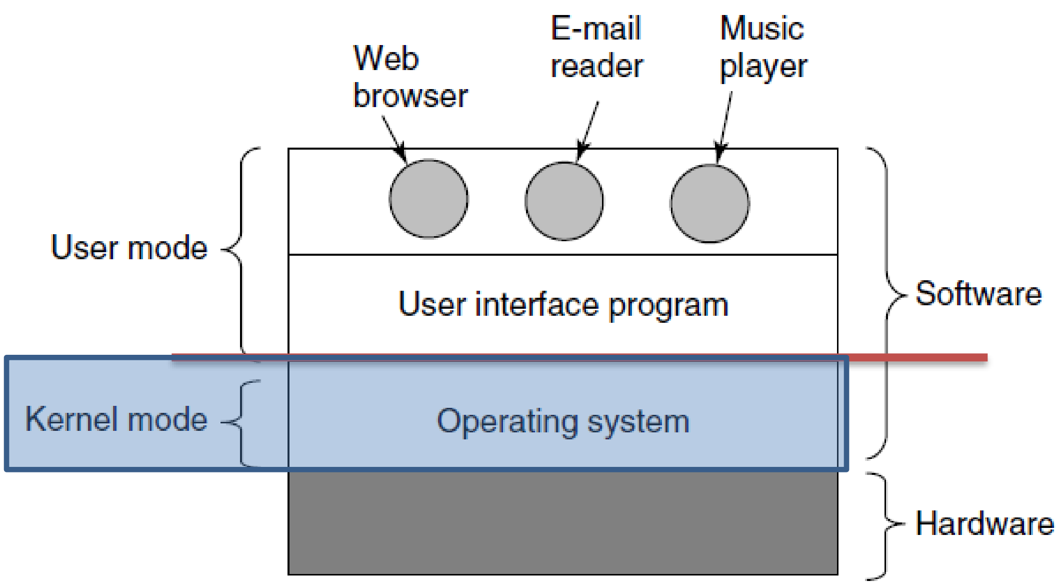
How to switch between the two modes:

UserMode -> KernelMode

- Trap
- Interrupt (also Kernel2Kernel)
- Execption (also Kernel2Kernel)

KernelMode -> UserMode

- rfi (return from interrupt)
  (also kernel to kernel)



Exception / Traps / Interrupt

- *Exceptions*:
  - triggered by a "fault condition" of an instruction condition
  - *Synchronous*
- Traps
  - special kind of exception ➔ instruction, aka sc [system call] , side effect is intended
  - *Synchronous:* triggered by "trap instruction" for syscall
- Interrupts:
  - Triggered by a hardware event from a "device" (device needs attention)
  - *Asynchronous*

Understand similarities and differences between these 3 "events"

Where do Exceptions/Traps/Interrupts switch to ?

- They all end up in the so called "interrupt handler":

  (1) Protected Hardware register is initialized in OS bootstrap with the address of kernel entry code (aka __entry) , so the hardware knows where to jump to when an or Exception or Trap or Interrupt is raised.

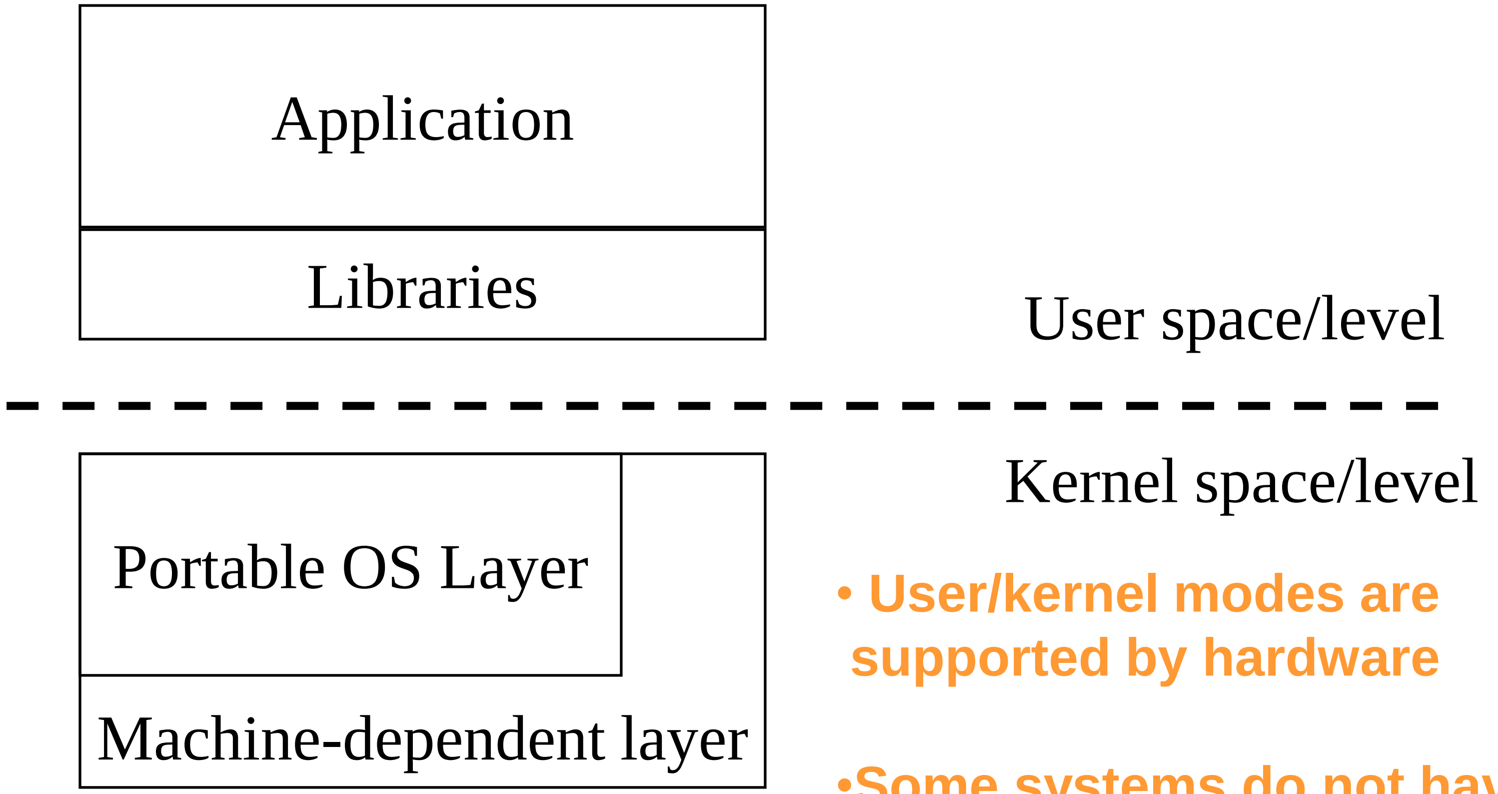
  (2) kernel entry code (__entry) is typically written in assembler because all registers must be accessed and a stack frame must be created to get as quickly as possible into "C" code.

  (2) on exception / trap / interrupt the CPU changes mode to kernel and starts execution to __entry in the kernel while the storing the failing/interrupted instruction.

  (3) from there the assembler identifies whether an interrupt, execption, or trap and jumps to their respective handlers.

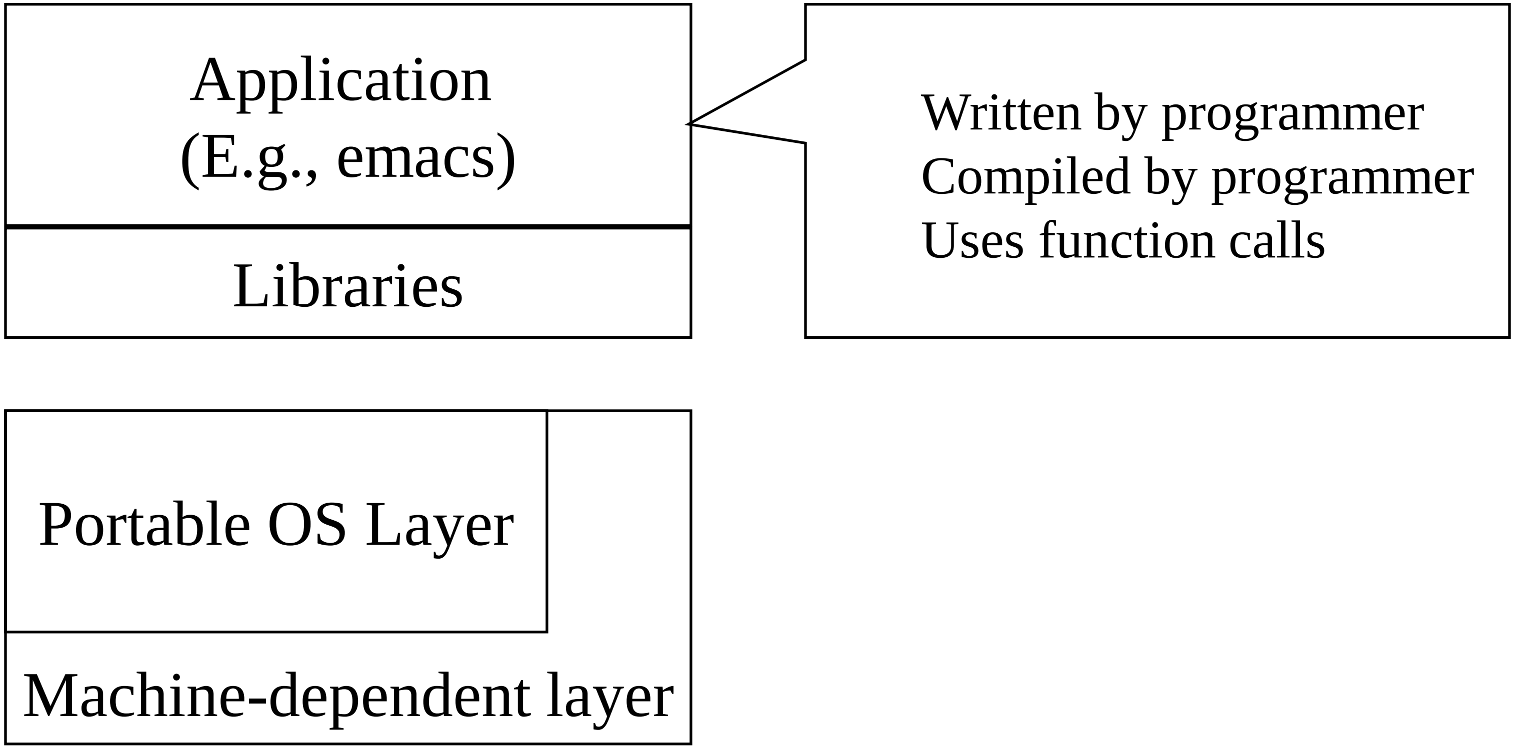
- __entry is the ONLY means to start running code in the operating system kernel

A peek into Unix/Linux

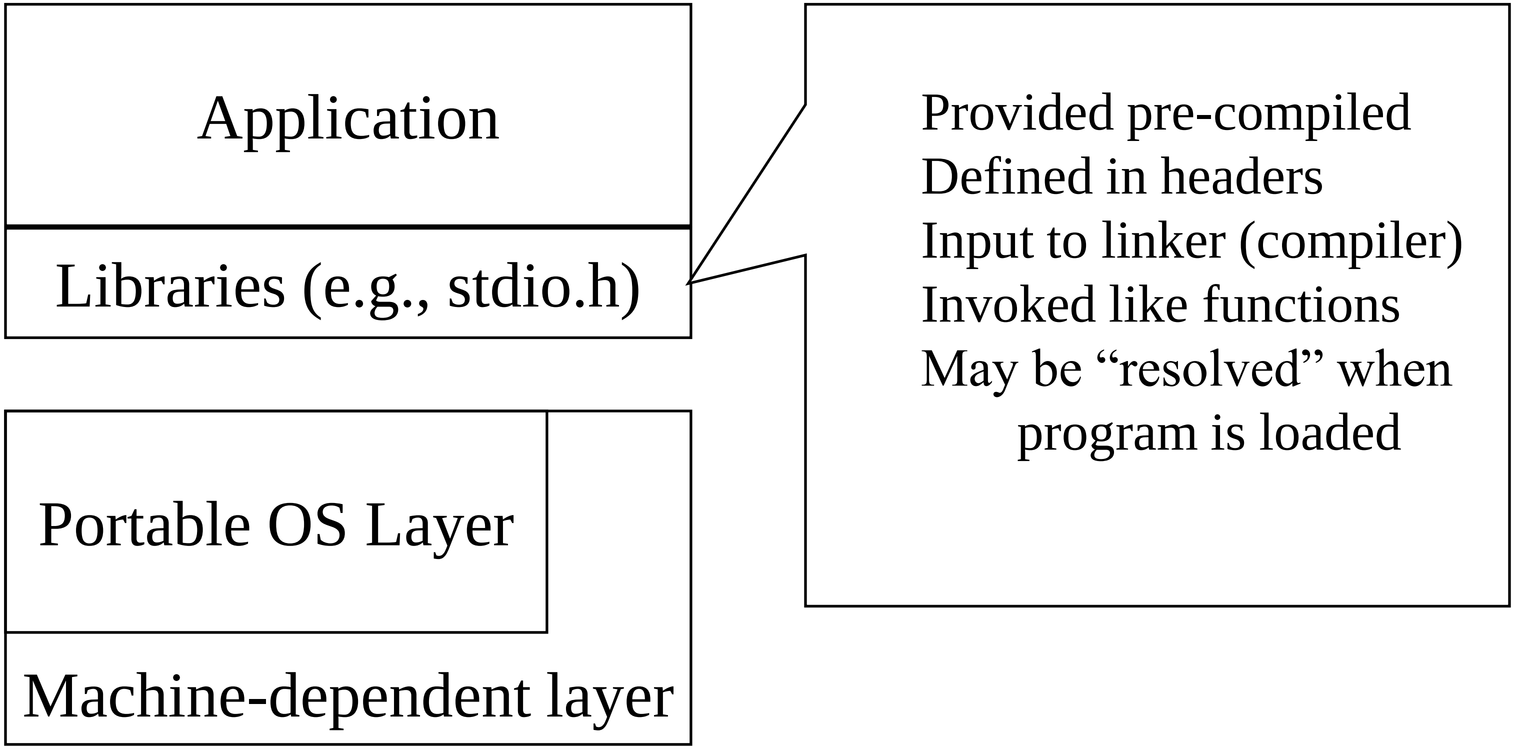


- User/kernel modes are supported by hardware
- Some systems do not have clear user-kernel boundary

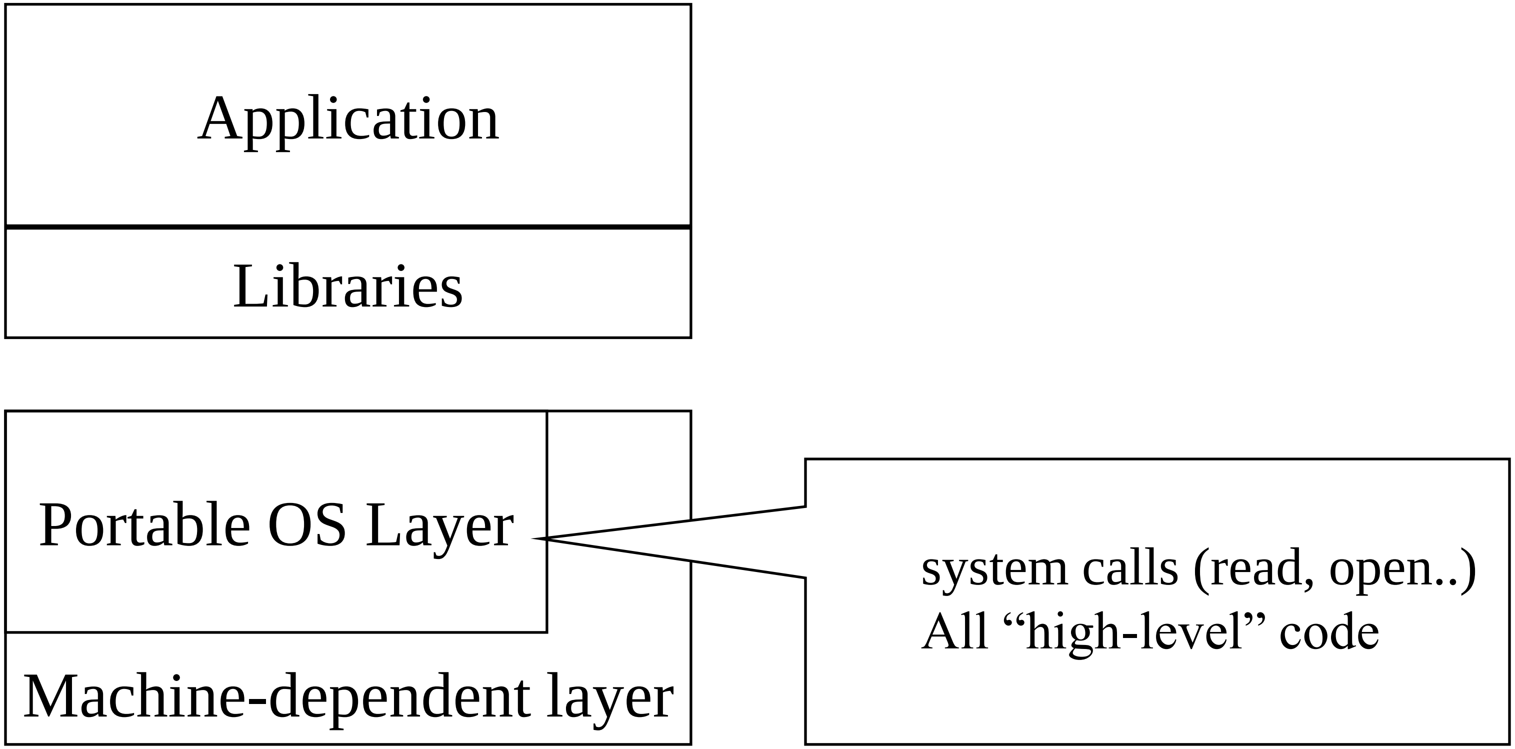
Unix: Application



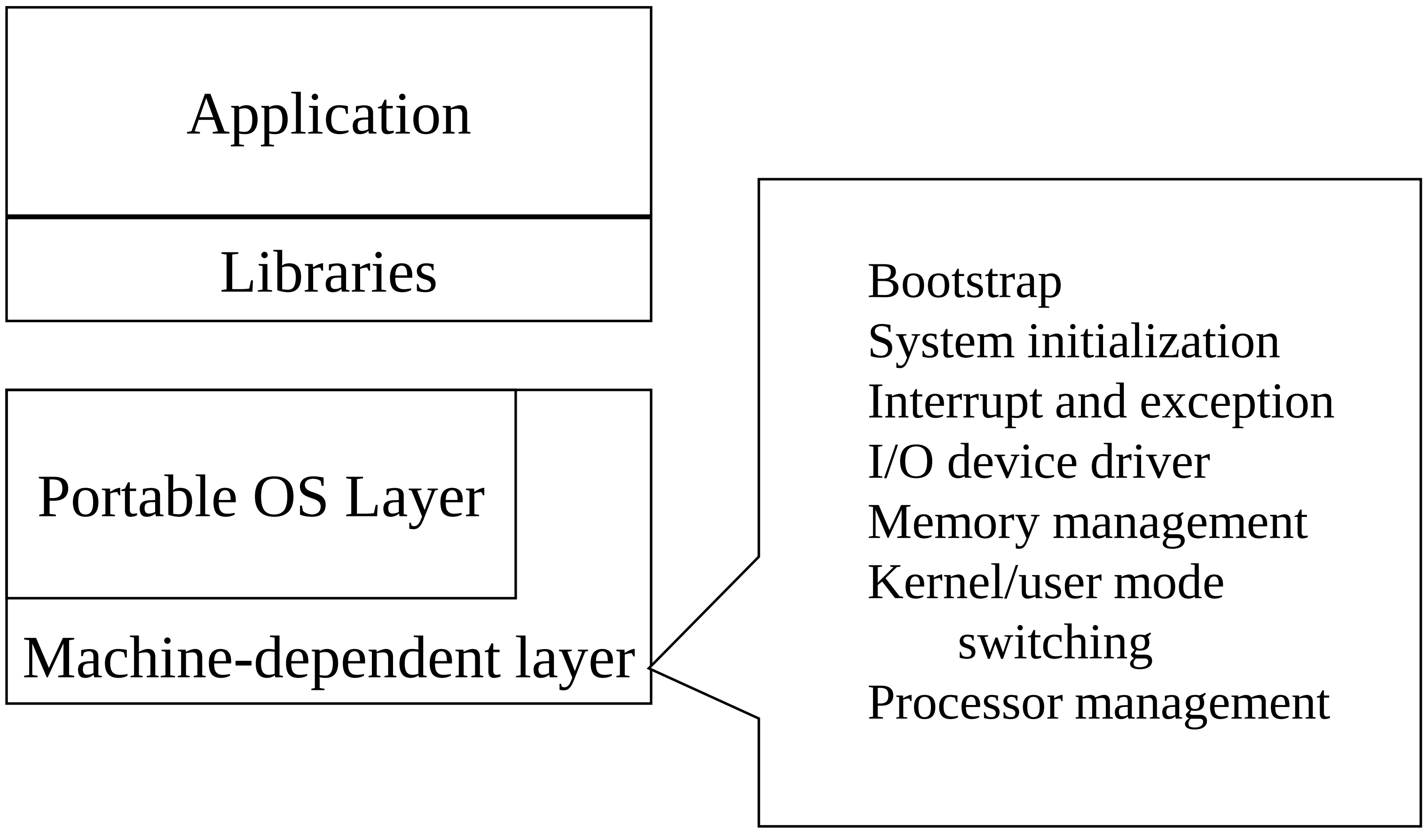
Unix: Libraries



Typical Unix OS Structure



Typical Unix OS Structure



Service Requests from user to kernel (OS)

- Basic means to request services from the operating system kernel is to make **system calls (which end up in a "trap / sc" event)**

- It's a well architected and "secure" API between kernel and userspace

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

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)
  - e = rmdir(name)

- Miscellaneous
  - e = kill(pid,signal)
  - e = chmod(name,mode)

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

System Call == OS kernel service request

- Reminder:

  Invoked via non-priviliged instruction (trap / sc)
  - Treated often like an interrupt, but its "somewhat" different

- Synchronous transfer control from user to kernel

- 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 see __entry -> syscall_handler

How are syscalls implemented

- First, one has to understand how arguments in any regular function call are passed.
- For this, a calling code convention is defined.
- Typically, arguments are passed through registers (sometimes as offsets on the stack)
- Those registers can be modified by the function called, any other registers most be saved and restored by the callee function
  - Volatile register (args,stackptr) and non-volative registers (callee must save and restore)
- Generally referred to as ABI: Application Binary Interface
- Syscalls are simply an extension on this. All compilers need to agree on ABI or code will no cooperate/work.

arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7
arm/OABI	a1	a2	a3	a4	v1	v2	v3
arm/EABI	r0	r1	r2	r3	r4	r5	r6
arm64	x0	x1	x2	x3	x4	x5	-
blackfin	R0	R1	R2	R3	R4	R5	-
i386	ebx	ecx	edx	esi	edi	ebp	-
ia64	out0	out1	out2	out3	out4	out5	-
mips/o32	a0	a1	a2	a3	-	-	-
mips/n32,64	a0	a1	a2	a3	a4	a5	-
parisc	r26	r25	r24	r23	r22	r21	-
s390	r2	r3	r4	r5	r6	r7	-
s390x	r2	r3	r4	r5	r6	r7	-
sparc/32	o0	o1	o2	o3	o4	o5	-
sparc/64	o0	o1	o2	o3	o4	o5	-
x86_64	rdi	rsi	rdx	r10	r8	r9	-
x32	rdi	rsi	rdx	r10	r8	r9	-

arch/ABI	instruction	syscall #	retval	Notes
arm/OABI	swi NR	-	a1	NR is syscall #
arm/EABI	swi 0x0	r7	r0	
arm64	svc #0	x8	x0	
blackfin	excpt 0x0	P0	R0	
i386	int $0x80	eax	eax	
ia64	break 0x100000	r15	r8	See below
mips	syscall	v0	v0	See below
parisc	ble 0x100(%sr2, %r0)	r20	r28	
s390	svc 0	r1	r2	See below
s390x	svc 0	r1	r2	See below
sparc/32	t 0x10	g1	o0	
sparc/64	t 0x6d	g1	o0	
x86_64	syscall	rax	rax	See below
x32	syscall	rax	rax	See below

Trap/SC instruction

Excellent writeup: https://lwn.net/Articles/604287/

syscall implementation (user side)

/usr/include/asm-generic/unistd.h

unistd.h:extern ssize_t pread64 (int __fd, void *__buf, size_t __nbytes)

```
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <unistd.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */

long syscall(long number, ...);
```

Definition agreed upon by libc and kernel
➔ ABI is known. Compiler assembles args

```
0000000000400596 <main>:
  400596:       55                      push   %rbp
  400597:       48 89 e5                mov    %rsp,%rbp
  40059a:       48 83 ec 70             sub    $0x70,%rsp
  40059e:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4005a5:       00 00
  4005a7:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4005ab:       31 c0                   xor    %eax,%eax
  4005ad:       48 8d 45 a0             lea    -0x60(%rbp),%rax
  4005b1:       ba 50 00 00 00          mov    $0x50,%edx
  4005b6:       48 89 c6                mov    %rax,%rsi
  4005b9:       bf 00 00 00 00          mov    $0x0,%edi
  4005be:       e8 ad fe ff ff          callq  400470 <read@plt>
  4005c3:       89 45 9c                mov    %eax,-0x64(%rbp)
  4005c6:       8b 45 9c                mov    -0x64(%rbp),%eax
  4005c9:       48 8b 4d f8             mov    -0x8(%rbp),%rcx
  4005cd:       64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
  4005d4:       00 00
  4005d6:       74 05                   je     4005dd <main+0x47>
  4005d8:       e8 83 fe ff ff          callq  400460 <__stack_chk_fail@plt>
  4005dd:       c9                      leaveq
  4005de:       c3                      retq
  4005df:       90                      nop
```

ABI: Application Binary Interface

```
/* fs/read_write.c */
#define __NR3264_lseek 62
__SC_3264(__NR3264_lseek, sys_llseek, sys_lseek)
#define __NR_read 63
__SYSCALL(__NR_read, sys_read)
#define __NR_write 64
__SYSCALL(__NR_write, sys_write)
#define __NR_readv 65
__SC_COMP(__NR_readv, sys_readv, compat_sys_readv)
#define __NR_writev 66
__SC_COMP(__NR_writev, sys_writev, compat_sys_writev)
#define __NR_pread64 67
__SC_COMP(__NR_pread64, sys_pread64, compat_sys_pread64)
#define __NR_pwrite64 68
__SC_COMP(__NR_pwrite64, sys_pwrite64, compat_sys_pwrite64)
#define __NR_preadv 69
__SC_COMP(__NR_preadv, sys_preadv, compat_sys_preadv)
#define __NR_pwritev 70
__SC_COMP(__NR_pwritev, sys_pwritev, compat_sys_pwritev)

/* fs/sendfile.c */
#define __NR3264_sendfile 71
__SYSCALL(__NR3264_sendfile, sys_sendfile64)
```

#define __SYSCALL(number) syscall(number...)

syscall is implemented as assembler largely taking the arguments already in the right registers and TRAP-ing into the kernel.

syscall implementation (kernel side)

- Kernel defines a table (using the compiler help)

```
int syscall_table [ __NR_SYSCALL_MAX ] = {
      :
      [ __NR_READ]  = sys_read,
      [ __NR_WRITE] = sys_write,
      :
}
```

The compiler does the magic and associates the syscall number with the kernel internal function

- On system trap, architecture automatically and immediately enters kernel mode and runs a small piece of assembler code that is stored at a machine register address set by the OS at boot time.

- Said trap assembler code (aka interrupt handler) does the following:
  - Checks the syscall number in well known register (see ABI) to be in range
  - Assembler equivalent:
    - Change stack to kernel (more on this in a bit)
    - All arguments are already in right place thanks to the ABI and the compiler's help
    - "Call/jmp" to syscall_table[registers.syscall_number];   // see ABI definition
    - After return from ^^^^, switch back from kernel stack to user stack and RFI (return from kernel mode).

Putting it all together (based on ARM/EABI)

Your app.c

```
char buf[128];
int  fd;
myfunct()
{
    read(fd,buf,64);
}
```

Regular function call

assembler based on compiler

```
ldr r0, =fd
ldr r1, =buf
ldr r2, =0x40
bl  read
bx  lr
```

```
r0 = fd
r1 = buf
r2 = 64
```

libc.so

```
#define syscall(sysnum) \
    asm("ldr r7,%d; swi 0x0", sysnum)

int read(int fd, char* buf, in nbytes)
{
    return syscall(__NR_READ);
}
```

```
ldr r7, =0x3F
swi 0x0
bx  lr
```

```
r7 = __NR_READ
```

User

kernel

```
__entry: // all assembler code,
   // save require non-volatile
   // figure out what type (intr,trap,exception

   call syscall_table[r7]
   return from interrupt
```

All argument registers must stay intact:
r0,r1,r2,…

```
shift r7,2
add r20,r7,=syscall_table
blr r20
bx  lr
```

Regular function call

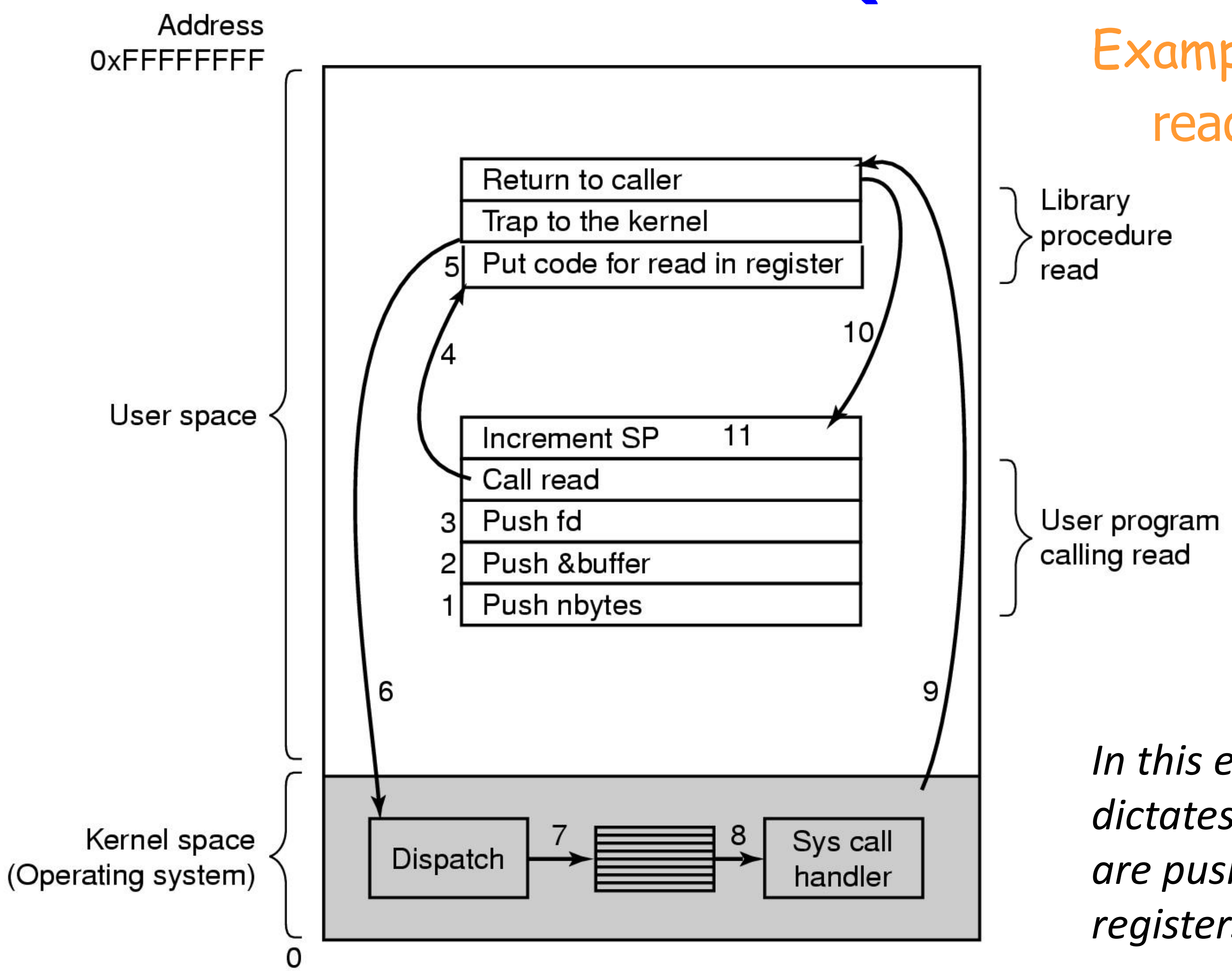
Function expects args in r0,r1,r2,…

```
int sys_read(int fd, char* buf, in nbytes)
{
    // some sophisticated kernel code
}
```

System Call with args passed over the stack (different ABI)

Example:

read (fd, buffer,nbytes)



In this example the ABI dictates that function arguments are pushed on the stack not through registers

(older architectures)

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

Other implicit/explicit OS Services Examples

- Services that can be provided at user level (because they only read unprotected data)
  - Read time of the day
- 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

Is Any OS Complete? (Criteria to Evaluate OS)

Portability

Security

Fairness

Robustness

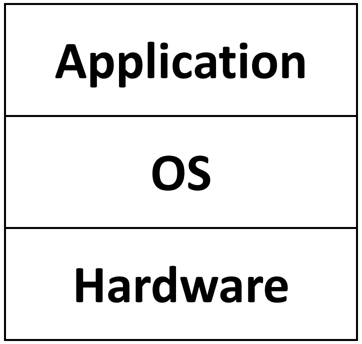
Efficiency

Interfaces

When you look at these criteria you should see that they can't all be satisfied at the same time.

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

Operating Systems Structure (Chapter 1)

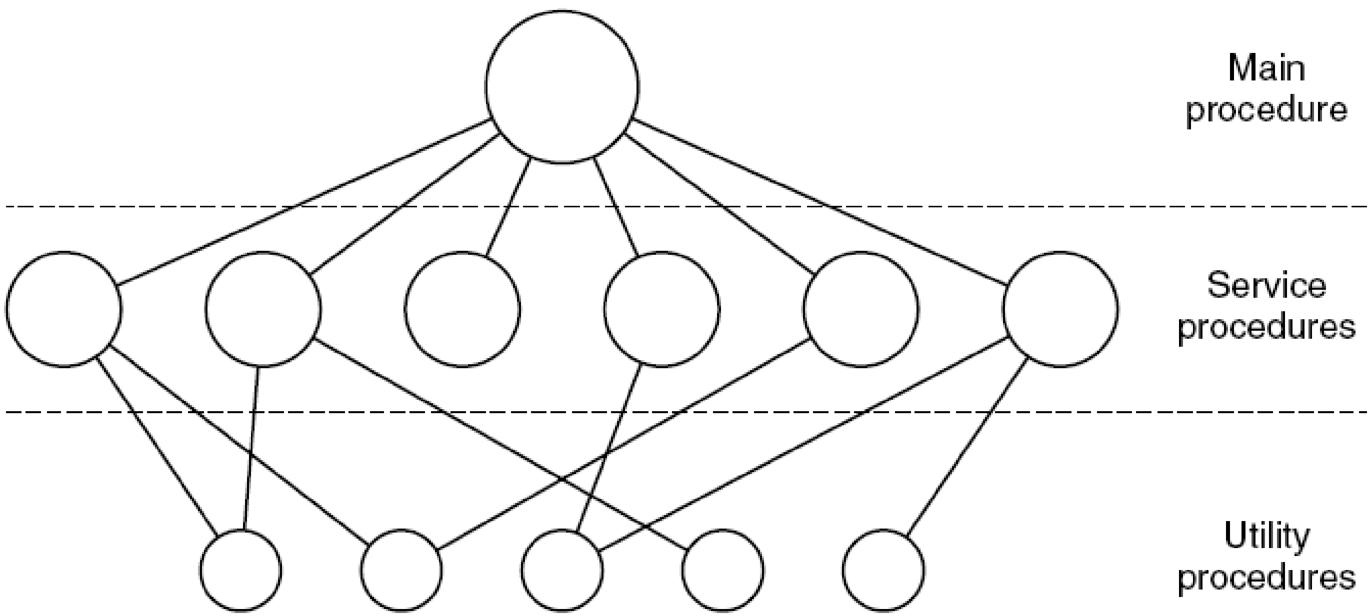
Monolithic systems – basic structure

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

Monolithic Systems

By far the most common OS organization

A simple structuring model for a monolithic system.



Layered Systems

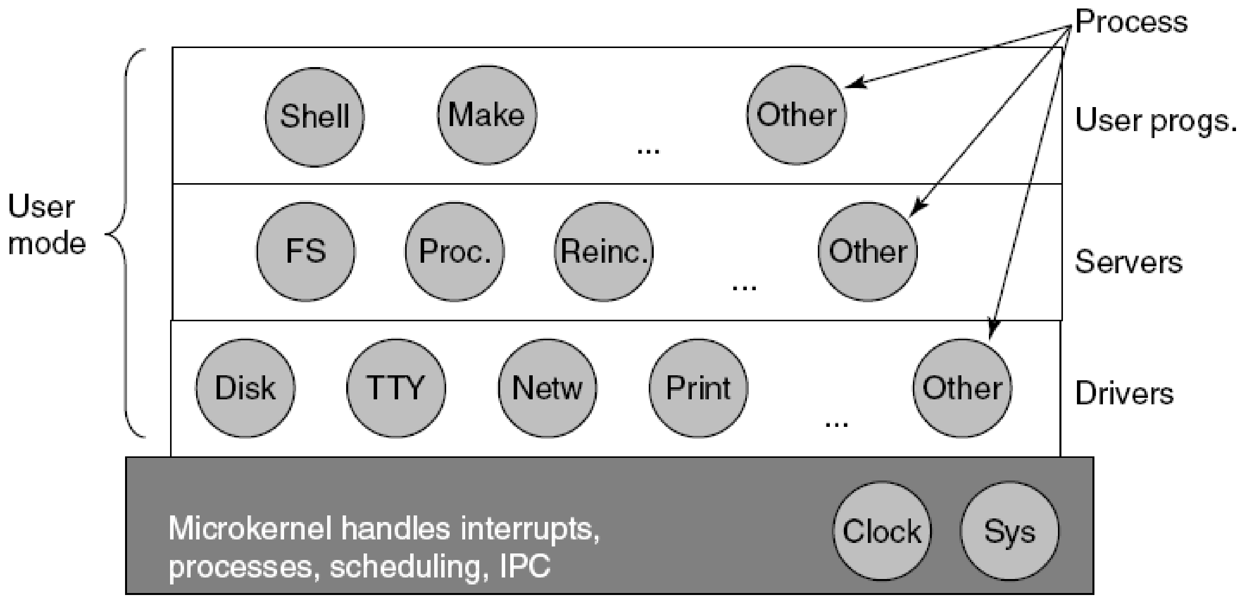
- Layer-n services are comprised of services provided by Layer-(n-1)..
- Structure of the THE operating system (Dijkstra 1968)

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- THE used this approach as a design AID
- Multics Operating System relied on Hardware Protection to enforce layering

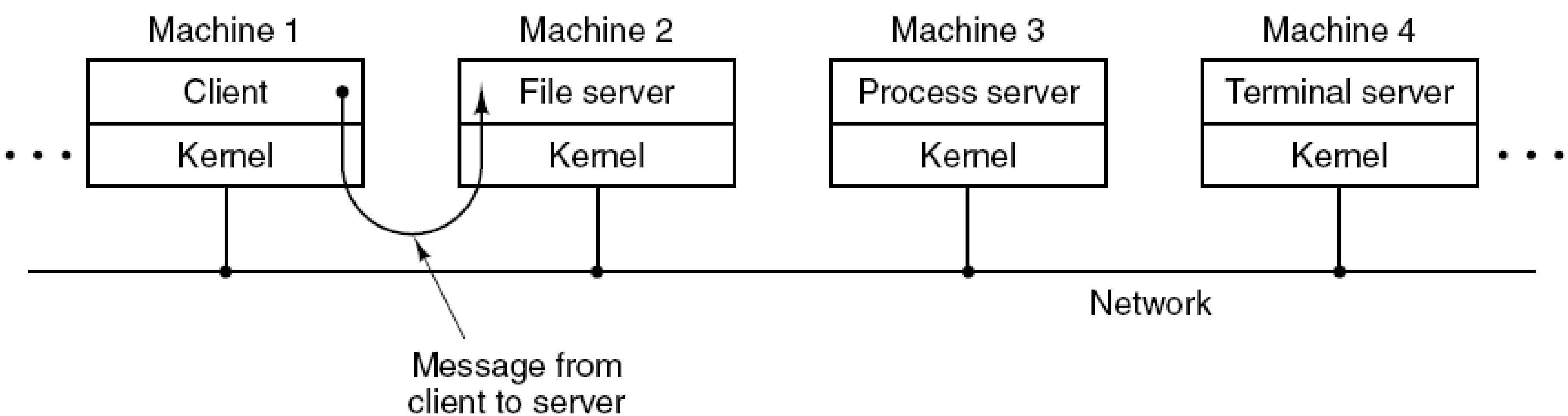
Microkernels

- Microkernels move the layering boundaries between kernel and userspace
- Move only most rudimentary services to kernel
- Move other services to Userspace
- Higher Overhead, but more flexibility, higher robustness
  when one OS non-micro kernel component fails it is being restarted.
  - Minix: is only 3200 lines of C and 800 lines assembler)
  - L4, K42,



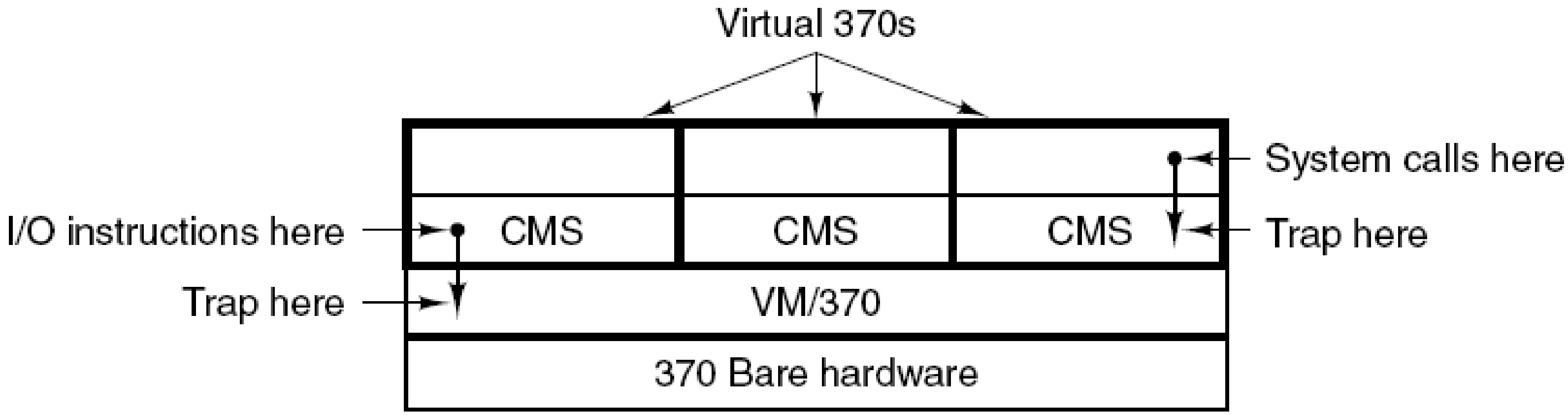
Client-Server Model

- Assumes generic network model (network, bus)
- Communication via message passing



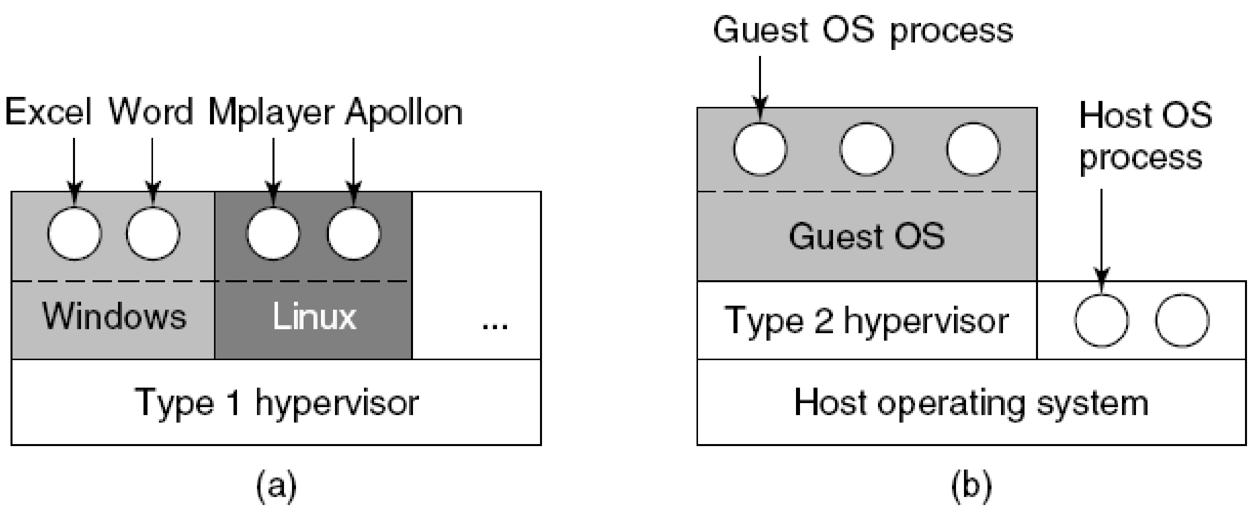
Virtual Machines (1)

- VM/370: Timesharing system should be comprised of:
  - Multiprogramming
  - extended machine with more interface than bare HW
  - Completely separate these two functions
- Provides ability to “self-virtualize”
- Beginning of “modern day” virtualization technology (1970)



Virtual Machines (2)

- A type 1 hypervisor
  (like virtual machine monitor)
  - Priviliged instructions are trapped and "emulated"
- A type 2 hypervisor (runs on top of a host OS)
  - Unmodified (trapped)
  - Modified (paravirtualization)



Other areas of virtual machines usage

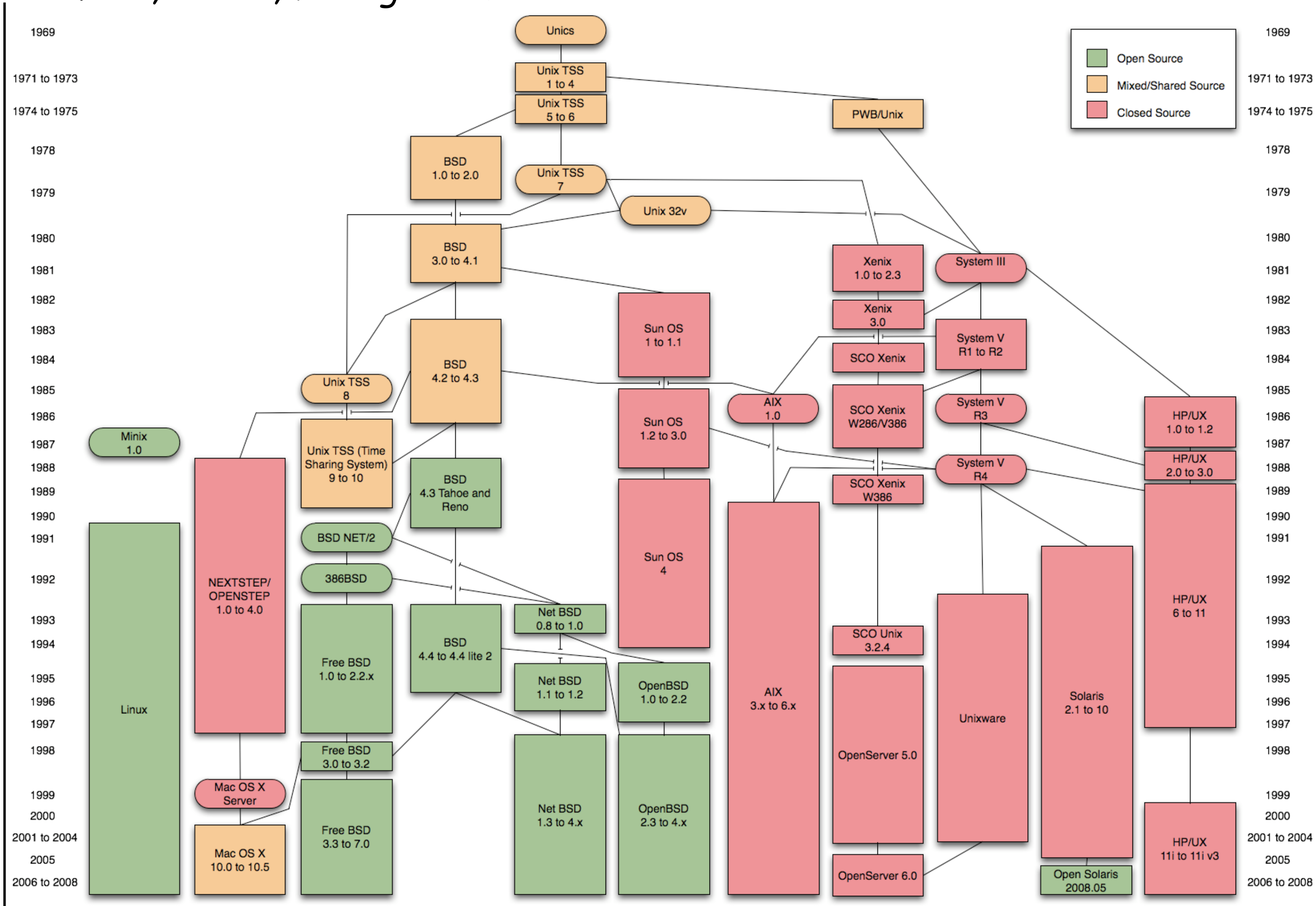
- Java virtual machines
- Dynamic scripting languages (e.g. Python)

- Typically define a instruction set that is "interpreted" by the associate virtual machine
  - JVM, PVM
  - Modern system then JIT (Just In Time) compile the VM instructions into native code.

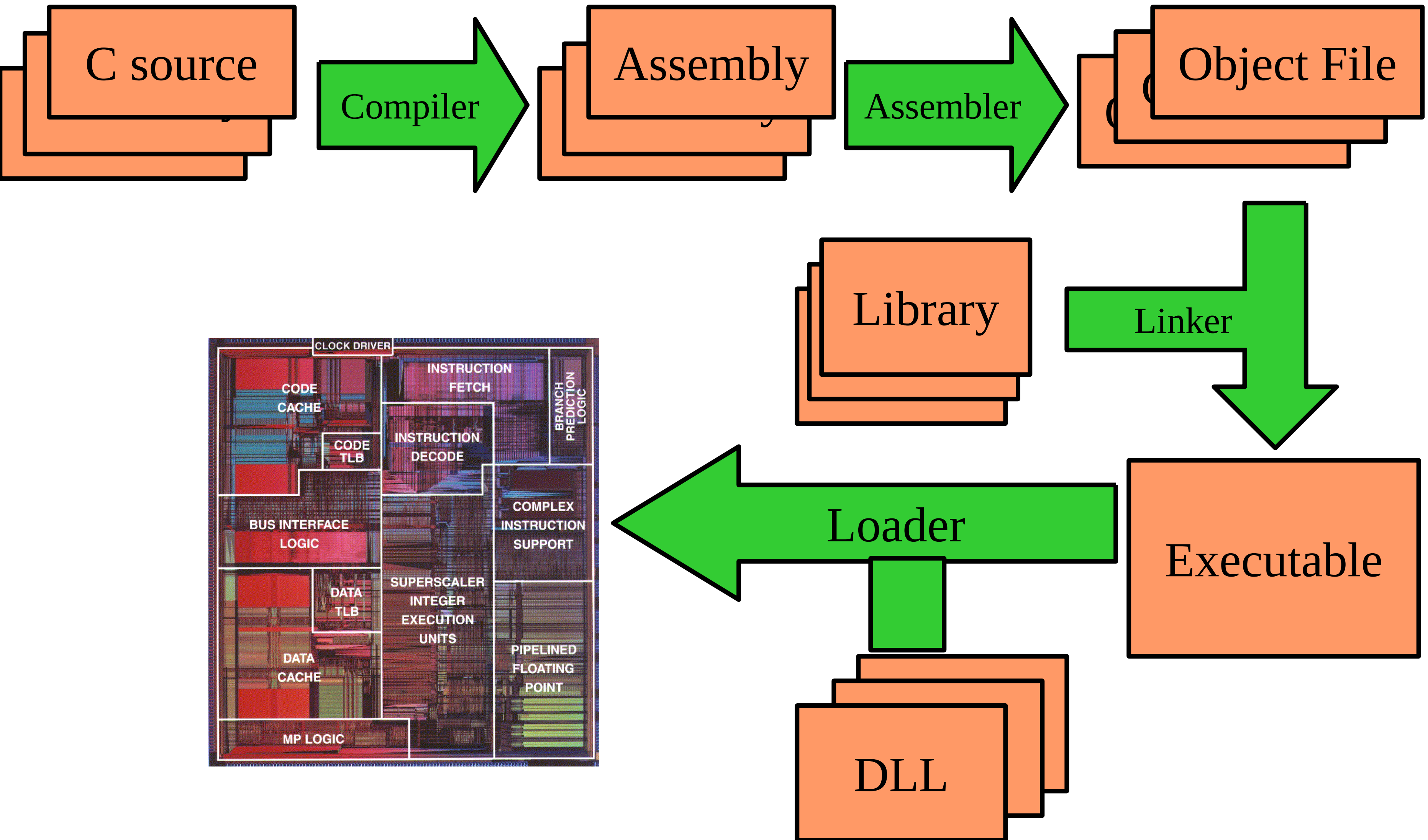
History of the UNIX Operating System

(source: wikipedia)

Bell Labs: *Thomson, Richie, Kernigan et.al*

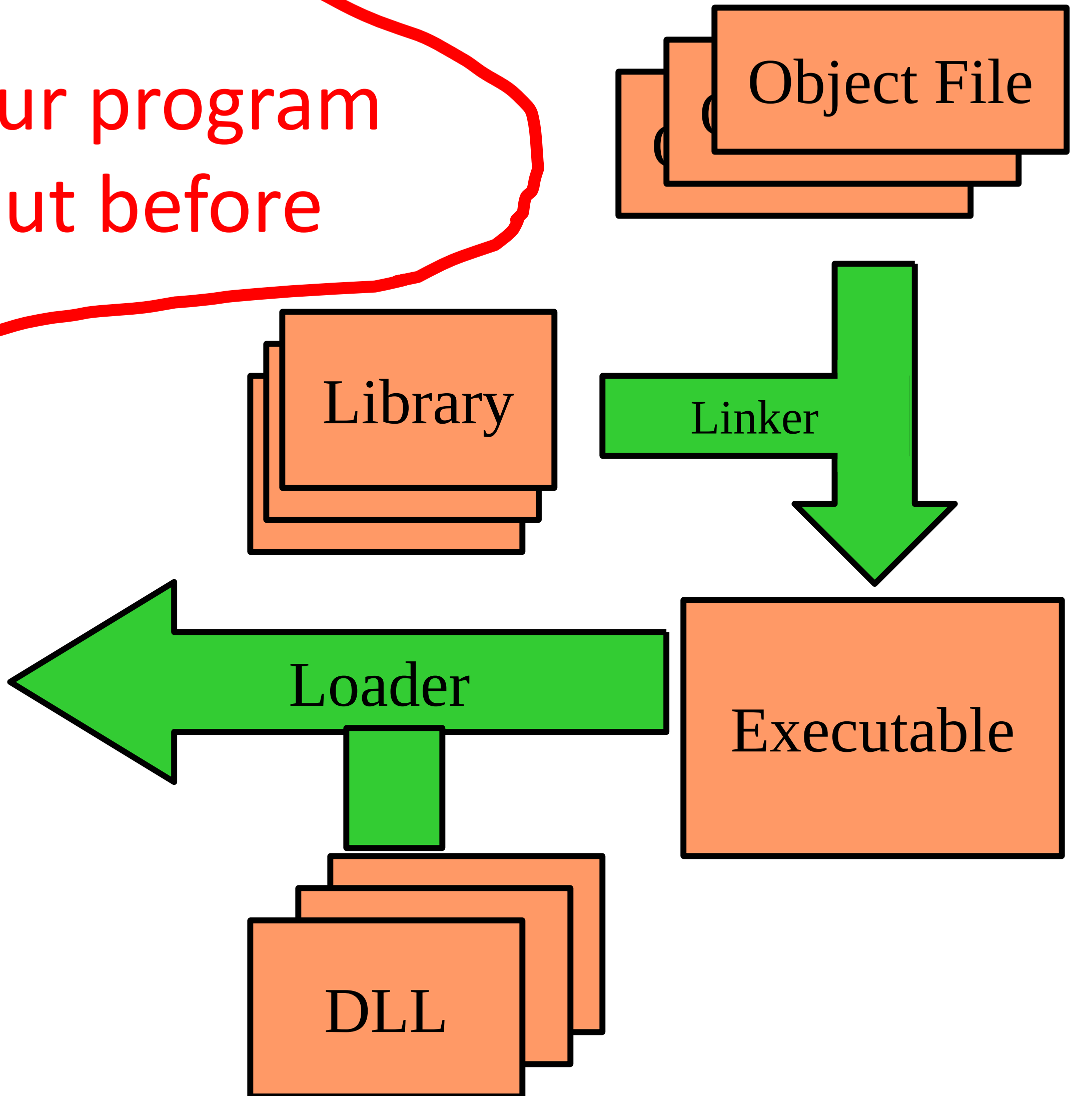


Source Code to Execution



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 memory addresses
- Resolves names of dynamic library items
- schedule program as a new process

Libraries (I)

- Programmers are expensive.

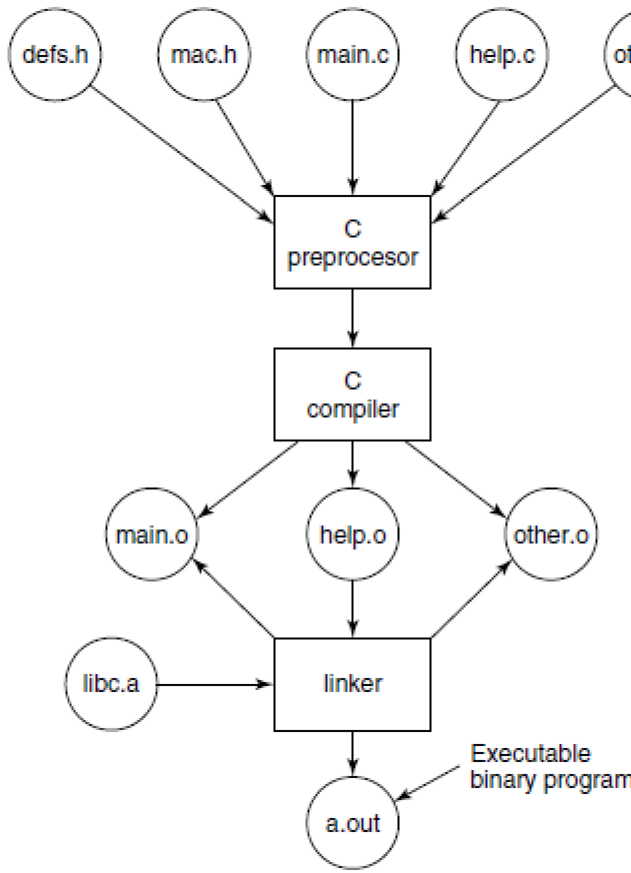
- Applications are more sophisticated.
  - Pop-down menus, streaming video, etc

- 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)

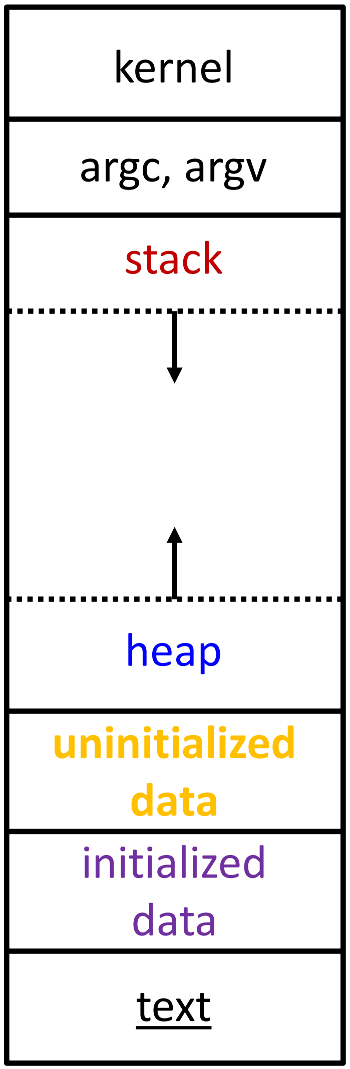
- A collection of subprograms
- Libraries are distinguished from executables in that they are not independent programs
- Libraries are "helper" code that provides services to some other programs
- Main advantages: reusability and modularity

Large Programming Projects



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

Linker:
A view of program / object module layout



Object-Module-1

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

#define NUMS (4)

int a;
int b = 2;
int x;
int y = 3;

extern int myfunc(int);

int main(int argc, char* argv[])
{
    int *values;
    int i;

    values = (int*)
          malloc(NUMS*sizeof(int));

    for (i=0 ; i<NUMS; i++)
        values[i] = myfunc(i);

    return 0;
}
```

Object-Module-2

```
int M;
int N = 2;


int myfunc(int val)
{
    return N*val;
}
```

Linker collocates code and text of same type together into the address space layout

Object Module 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)

Object Module 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

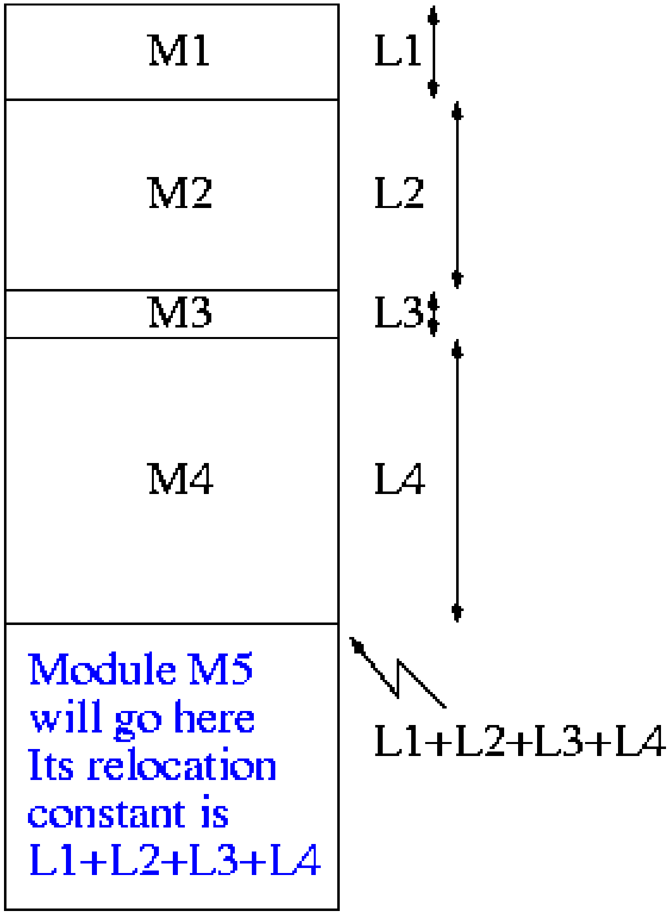
The linker changes this instruction to

jump 2420

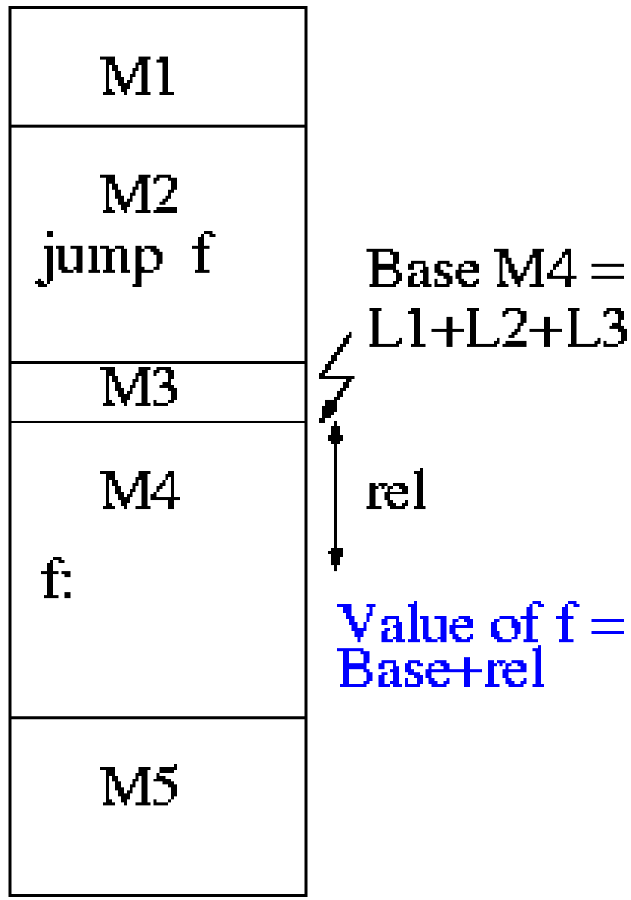
Object Module Relocation

- 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.

Relocation


M1
L1
M2
L2
M3
L3
M4
L4
Module M5
will go here
Its relocation
constant is
L1+L2+L3+L4
L1+L2+L3+L4

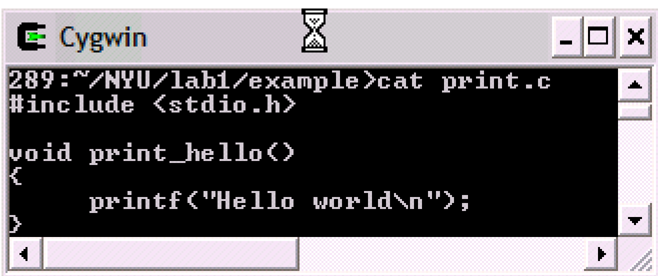
Relocation

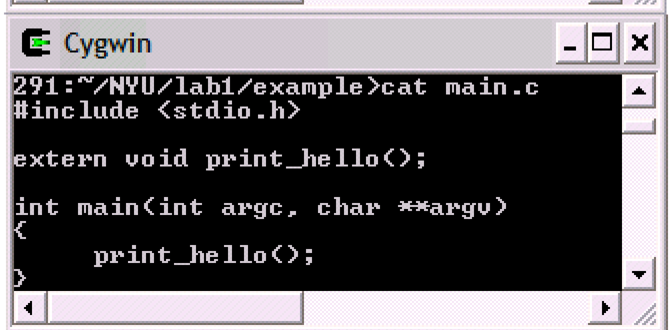


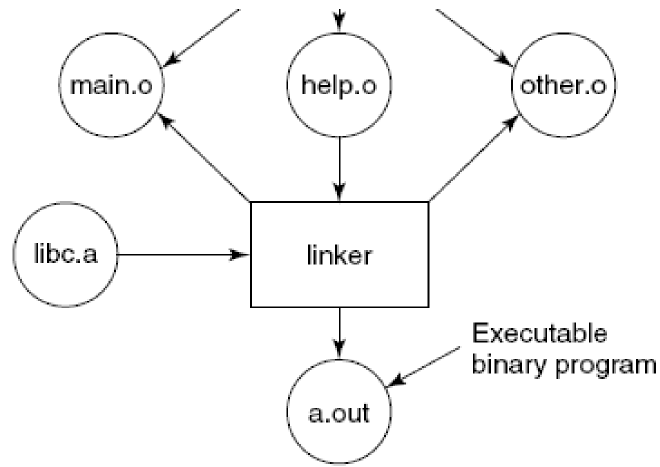
LAB assignment #1

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

```
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
```

Module-3

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
```