Operating System Overview

Otto J. Anshus

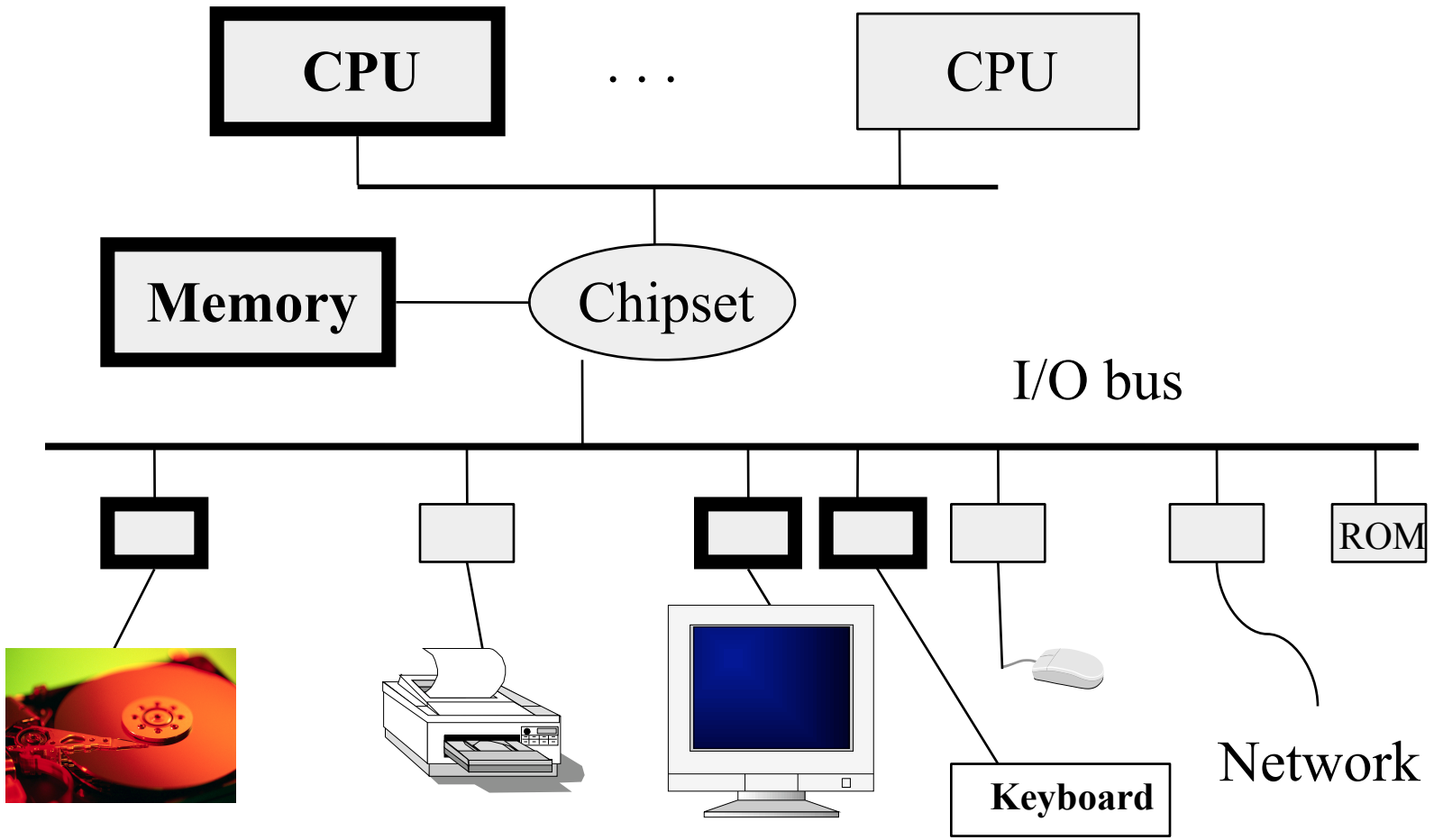
What Do Operating Systems DO?

- System construction
  - raw hardware devices are not usable as is
  - how to make HW usable
  - how to make unreliable components reliable
- Protection
  - simple OS is inefficient, need to do more, overlap
  - how to run multiple applications (and do it safely)
  - how to prevent applications from crashing a system
- Resource management
  - resources are always limited
  - how to make finite CPU, memory, and I/O go around
  - how to make resource allocation fair

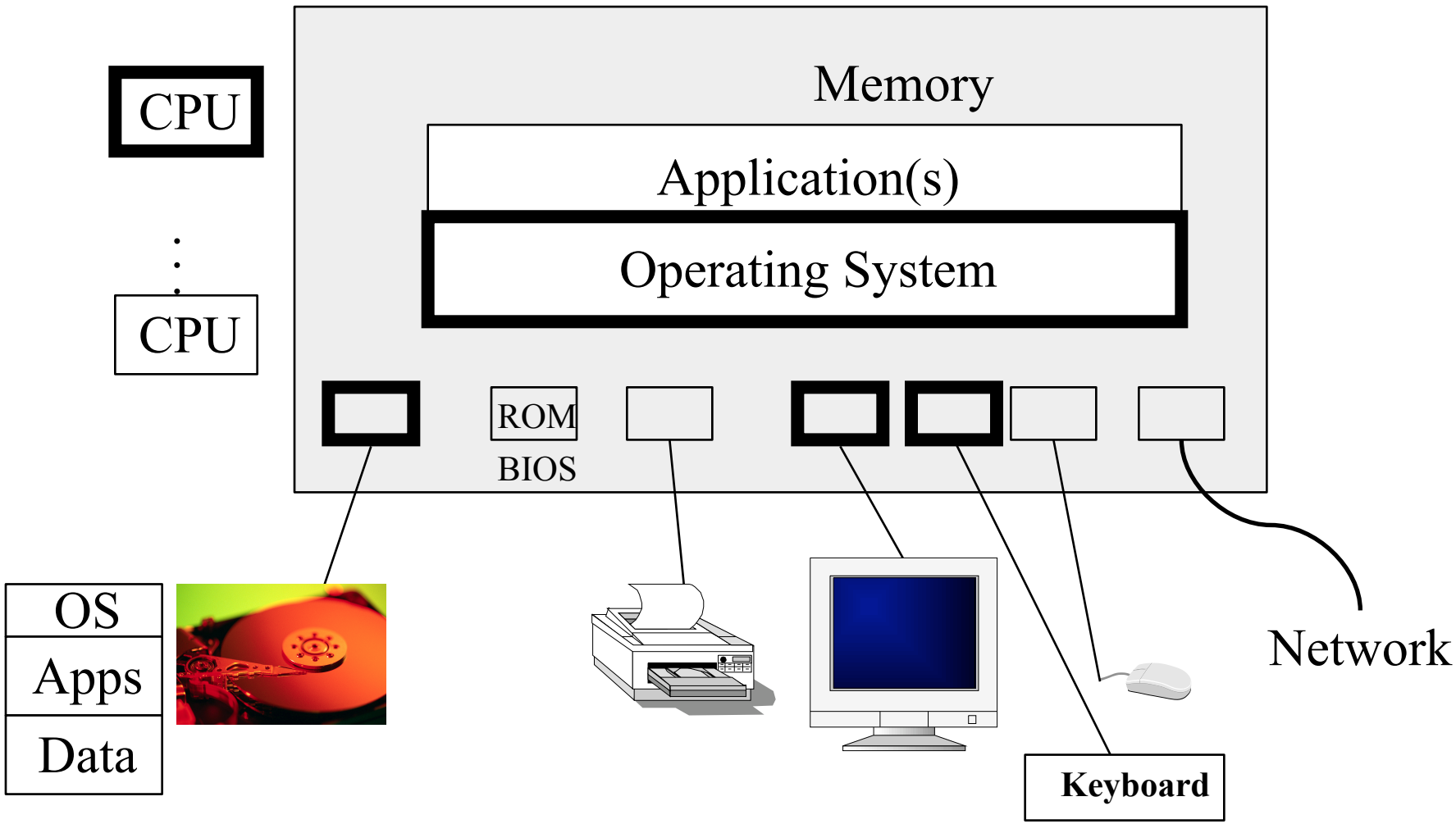
Topics

- **Protection & System Calls**
- **OS Structures, Process, Threads**
- **Non-preemptive Scheduling**
- **Synchronization**
  - **Threads & Mutex**
  - **Preemptive Scheduling & Mutex**
  - **Semaphores, Eventcounts, Monitors**
- **Thread packages**
- **CPU scheduling**
- **Deadlocks**
- **Message Passing**
- **I/O devices & Drivers**
- **Address Translation**
- **Paging**
- **VM Design Issues**
- **Disks & Files**
- **File systems & FS implementation**

A Simple Computer



A Simple Computer System



Why OS is not Trivial

- Simple "do one thing at a time" OS is inefficient
  - If the single process is waiting for something, whole computer sits idle ("Idleness is the Root of All Evil")
- Obvious Idea
  - Run more than one process "at once" (interleaved vs. overlapped)
  - When one process is delayed, switch to another
- Obvious Potential Problems
  - What if a program runs an infinite loop
  - …or starts to randomly access memory
- OS must carefully share resources while protecting resources, users and applications (from each other)

Why OS is not Trivial

- A too simple OS is expensive
  - One user occupy one computer
  - One application occupy whole computer
- Obvious Idea
  - Allow **N** users and **M** applications "at once"
    - Does machine now run about **N*M** times slower?
- Obvious Potential Problems
  - What if users or apps are evil (even if they "did not mean it")
  - … or they are too many
- Obvious (?) solution
  - OS must carefully share resources while **protecting** both itself, and resources, users and applications (from each other)

Protection at 10,000 feet

- Protection is to isolate “bad” programs and users
  - Preemption, interposition, and privileged operations
- Preemption
  - Give apps and users something, but can take it away
- Interposition
  - OS between app and HW resources
  - Track/watch resources given to apps and users
  - On every access, check that access is legal
- Privileged/unprivileged mode
  - Apps and users are unprivileged
  - OS is master and commander

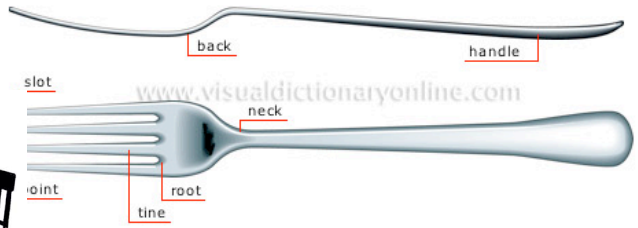
Successful Protection Examples

- Protecting CPU by using **preemption**
  - Clock interrupt: HW periodically suspends (app), and give control to OS (or some special code at known places)
  - OS decides whether to take CPU away from app or not, and who to give it too
- Protecting memory by using **address translation**
  - Every **load** and **store** instruction checked for legality
  - Typically use this machinery to translate to another value

Real Systems have Problems

- Most protect some things, but ignore others
- Can have trouble running

```
int main(){
        while(1)
                fork();
}
```



  - Common response: *freeze*. Reboot to unfreeze
  - Assume "stupid", but not malicious users
- Duality
  - **Technical**: *force* processor & memory *quotas*
  - **Social**: *help* users to learn more, *yell (or worse?)* at malicious users

Fixed Pie but Infinite Demand

- How to make the cake go further?
  - Resource usage is *bursty*
    - Give to others when idle/waiting
- But, to have higher utilization => more complexity (more can be bad)
  - One road/car vs. freeway
  - How to manage
    - Abstractions (different lanes),
    - Synchronization (traffic lights),
    - Increase capacity (build more roads)
- But, higher utilization => more contention
  - What to do when illusion breaks?
    - Refuse service (busy signal), give up (VM swapping), backoff and retry (Ethernet)

Finite -> “Infinite”

- Method 1: Exploit bursty apps
  - Take resources from idle code (process, thread) and give to code with something to do right now
- Method 2: Exploit skew
  - 80% of time taken by 20% of code
  - 10% of memory absorbs 90% of references
    - Cache 10% in fast memory, 90% in slow
- Method 3: Past predicts the future
  - Assume future == past
  - What’s the best cache entry to replace?
  - Good for weather forecasting? Stock exchange? Car driving?

More Examples

- System example
  - what if a user tries to access disk blocks directly?
- Protection example
  - what if a program starts to access memory randomly?
- Resource management example
  - what if user tries to push the system limits?

    int main(){while(1) fork();}
  - what if many programs are running infinite loops?

    while (1);

60’s vs. 00’s

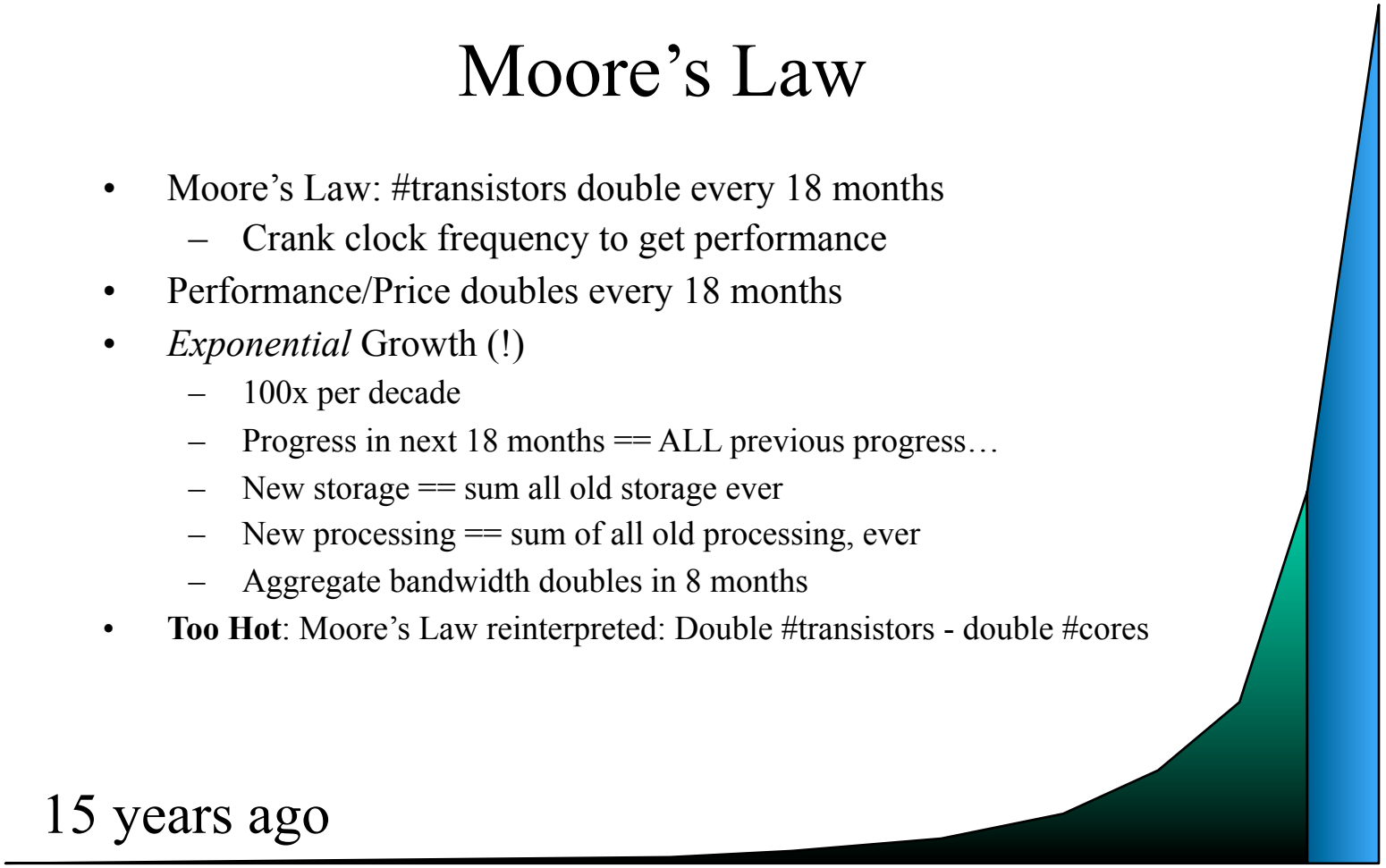
- Today is like in the late 60s. OS's are enormous
  - small OS: 100K lines
  - big OS: 10M lines
  - 100-1000 people years
- But ~90% is device drivers
- Project 5 (“post files”): ~6500 lines ☺

A Typical "Academic" Computer 1985 : 2005 : 2010

	1985	**2005**	**2010**	**Ratio**
CPU clock	3MHz	3GHz	3GHz (but multi-core)	1:1000
$/machine	$80K	$800		100:1
DRAM	0.5MB	0.5GB	5GB	1:1000:10 000
Disk	30MB	300GB		1:10,000
Network bandwidth	10Mbits/sec	1Gbits/sec		1:100
Address bits	16-32	32-64		1:2
Users/machine	10s	<1		>10:1
$/performance	$80K	<$800/1000		100,000+:1

Moore's Law

- Moore's Law: #transistors double every 18 months
  - Crank clock frequency to get performance
- Performance/Price doubles every 18 months
- *Exponential* Growth (!)
  - 100x per decade
  - Progress in next 18 months == ALL previous progress…
  - New storage == sum all old storage ever
  - New processing == sum of all old processing, ever
  - Aggregate bandwidth doubles in 8 months
- **Too Hot**: Moore's Law reinterpreted: Double #transistors - double #cores



You can GREP 1 MB in a second
You can GREP 1 GB in a minute
You can GREP 1 TB in 2 days
You can GREP 1 PB in 3 years

You can FTP 1 MB in 1 sec
You can FTP 1 GB / min (= 1 $/GB)
… 2 days and 1K$
… 3 years and 1M$

The rate of the overall computing power has been amazingly growing for more than one hundred years

Computing efficiency in ops/s/$ had 3 growth curves:

Combination of Hans Moravac + Larry Roberts + Gordon Bell
WordSize*ops/s/sysprice

1890-1945

Mechanical

Relay

7-year doubling

1945-1985

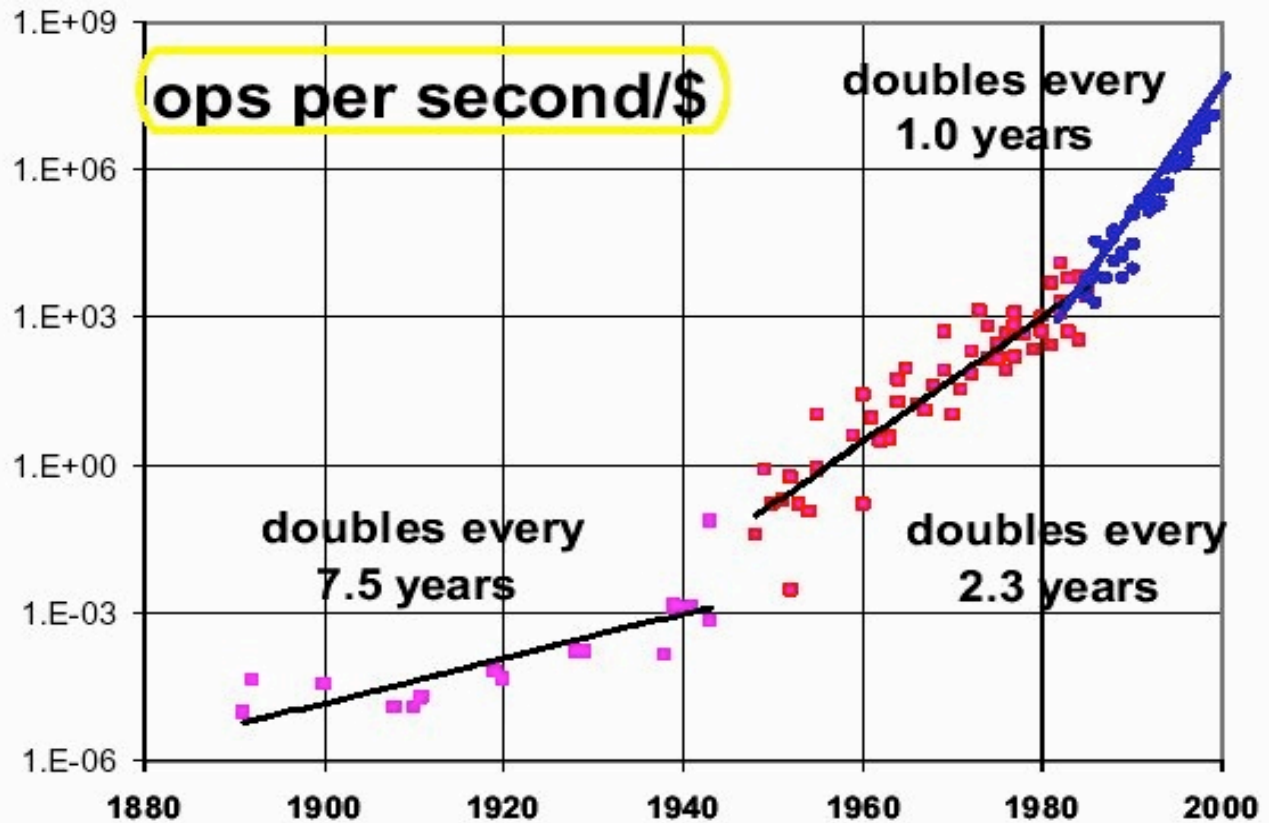
Tube, transistor,..

2.3 year doubling

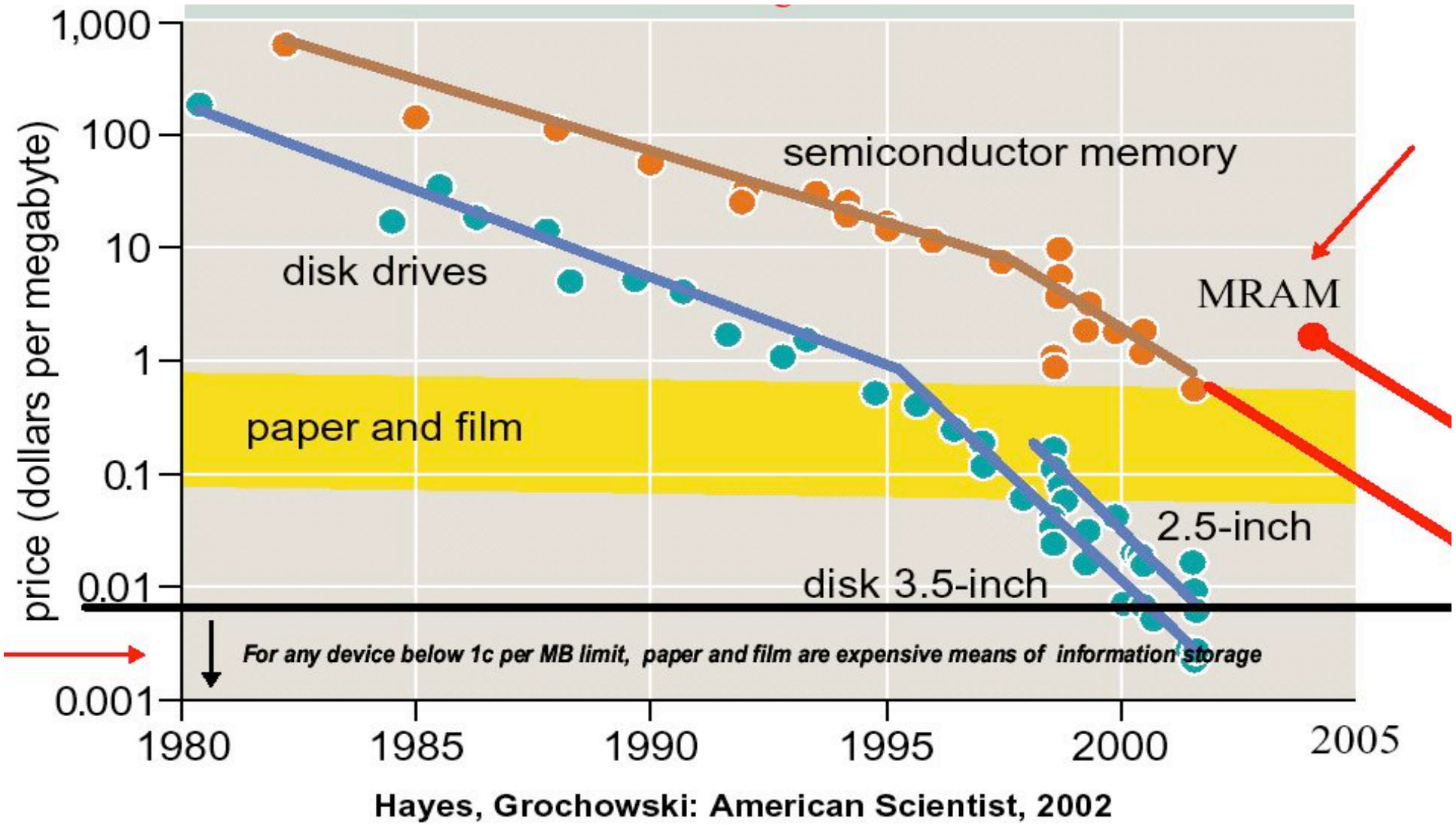
1985-2000

Microprocessor

1.0 year doubling



Exponentially Declining Cost of Data Storage



Hayes, Grochowski: American Scientist, 2002

Moving Data is Slow!

How long does it take to move a Terabyte?

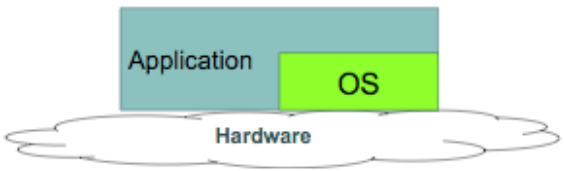
(how about a Petabyte?)

Context	Speed Mbps	Rent $/month	$/Mbps	$/TB Sent	Time/TB
Home phone	**0.04**	**40**	**1,000**	**3,086**	**6 years**
Home DSL	**0.6**	**50**	**117**	**360**	**5 months**
T1	**1.5**	**1,200**	**800**	**2,469**	**2 months**
T3	**43**	**28,000**	**651**	**2,010**	**2 days**
OC3	**155**	**49,000**	**316**	**976**	**14 hours**
OC 192	**9600**	**1,920,000**	**200**	**617**	**14 minutes**
100 Mpbs	**100**				**1 day**
Gbps	**1000**				**2.2 hours**

Source: TeraScale Sneakernet, Microsoft Research, Jim Gray et al.

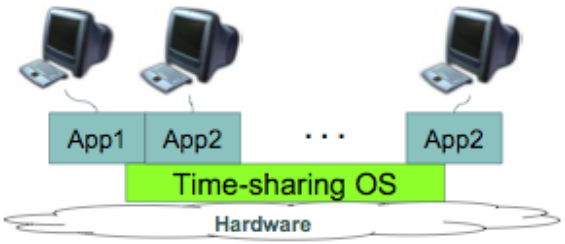
Phase 1: HW Expensive, Human Cheap

- User at console, OS as subroutine library
- Batch monitor (no protection): load, run, print
- Developments
  - Data channels, interrupts: overlap I/O and CPU
  - Direct Memory Access (DMA)
  - Memory protection: keep bugs to individual programs
- Exceptions to the rule
  - Multics: www.multicians.org OS designed 1963 and ran in 1969,
  - Multics, the foundation of Unix www.multicians.org/general.html#tag14
  - IBM 360 OS released with 1000 bugs



Phase 2: HW Cheap, Human Expensive

- Use cheap terminals to let users share a computer
- Unix enters the mainstream
- Problems: thrashing as the number of users increases



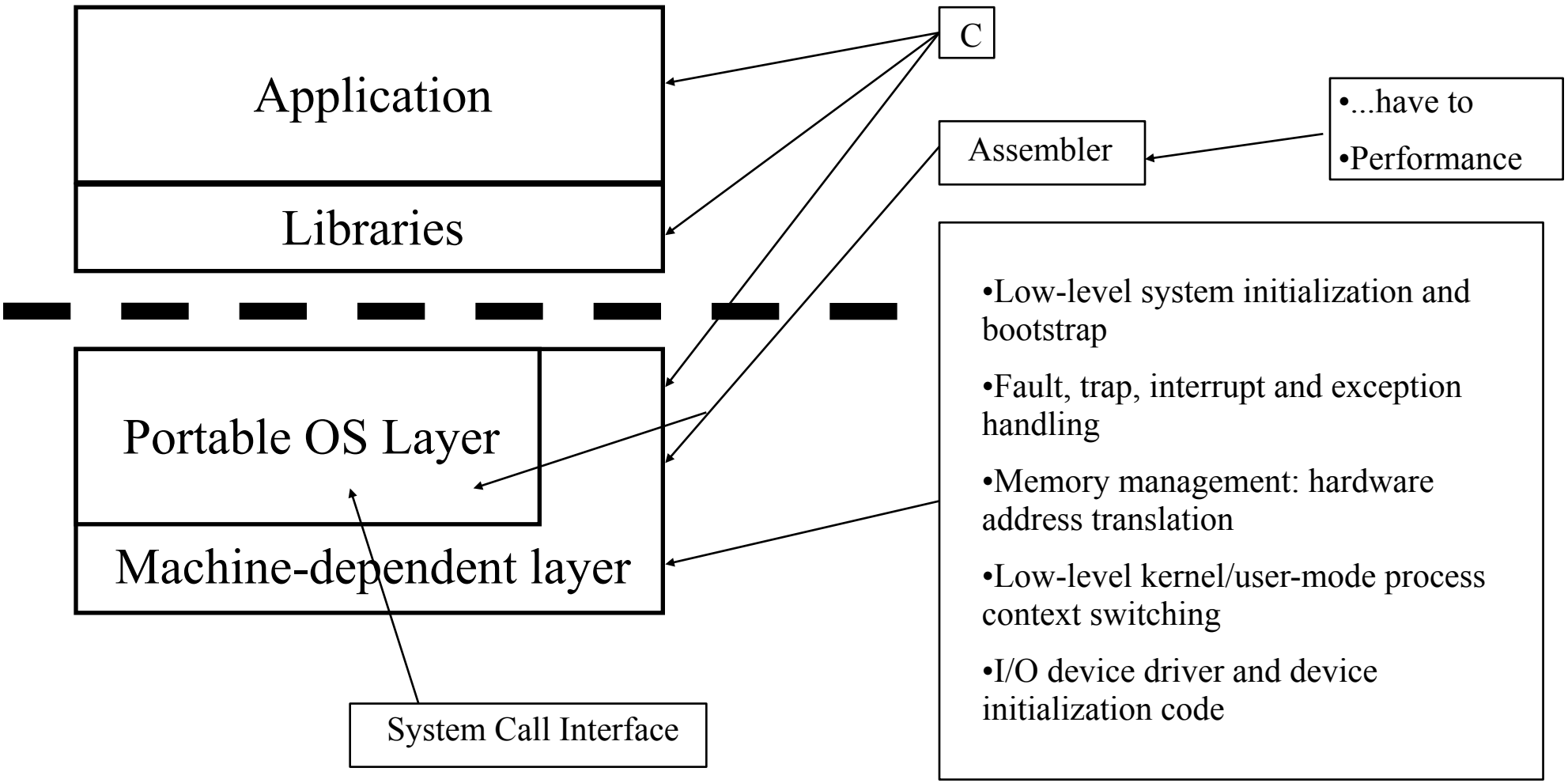
Phase 3: HW Cheaper, Human More Expensive

- Personal Computer
  - Altos OS: Ethernet, Bitmap display, laser printer
  - Pop-up menu window interface, e-mail, publishing software, ftp, telnet, …
  - Eventually >100M units/year
- PC OS
  - memory protection
  - multiprogramming
  - networking

Phase 4: >1 Computer per User

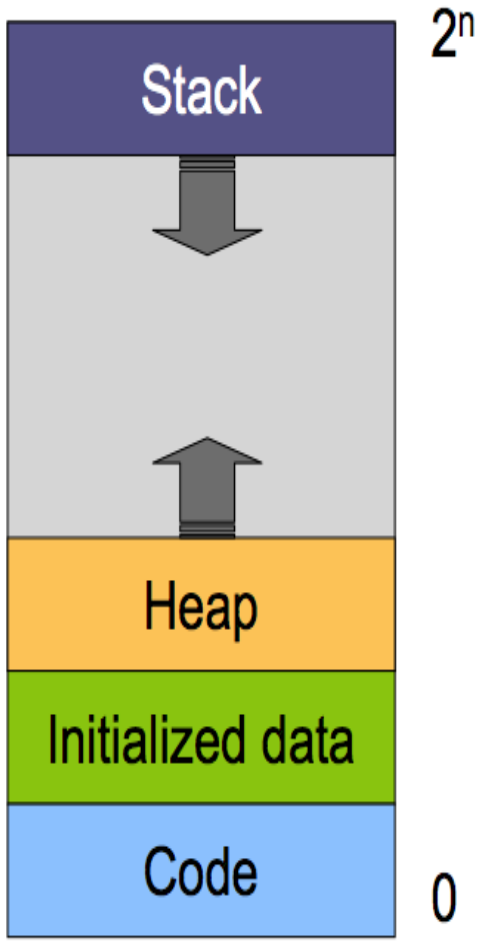
- Parallel and distributed systems
  - Parallel machine
  - Clusters
    - Uni-processors (single and multi-core)
    - Multi-processors (single and multi-core)
    - Grid
      - Multiple computers
      - Multiple clusters of computers
  - Clouds
  - Pervasive computers
    - Wearable computers
    - Computers everywhere, invisible (?)
- OS both specialized and general

Typical Unix OS Structure



What is an Application?

- Four segments
  - Code/text: instructions
  - Data: variables
  - Stack
  - Heap
- Why?
  - Separate code and data
  - Stack and heap grow toward each other



- **Stack**
  - Layout by compiler
  - Allocate at process creation (fork)
  - Deallocate at process termination
- **Heap**
  - Linker and loader specify the starting address
  - Allocate/deallocate by library calls such as **malloc()** and **free()** called by application
- **Data**
  - Compiler allocate statically
  - Compiler specify names and symbolic references
- **Linker** translate references and relocate addresses
- **Loader** finally lay them out in memory

OS Service Examples

- Examples of services **not** provided by code running at user level
  - System calls **(calls from apps to the OS kernel)**
    - File open, close, read and write **(implemented inside the OS kernel)**
  - OS control **(implemented inside OS kernel)**
    - Control the CPU so that users can't take over by doing
      - `while ( 1 ) ;`
    - Protection:
      - Keep user programs from crashing OS
      - Keep user programs from crashing each other
- Example of service we can allow to be running at user level
  - Read time of the day

User level vs. Kernel level

- Kernel (a.k.a. supervisory or privileged) level
  - All instructions are available
  - Total control possible so OS must say “Mine, all mine” Daffy Duck
- User level
  - Some instructions are not available any more
  - Programs can be modified and substituted by user

What to Do When Trying to Execute an Illegal Instruction

- **Instruction Stream**
  - **Fetch instruction**
  - **Decode instruction**
  - **Fetch operands**
  - **Execute**
  - **Write back result**
  - **Next instruction**

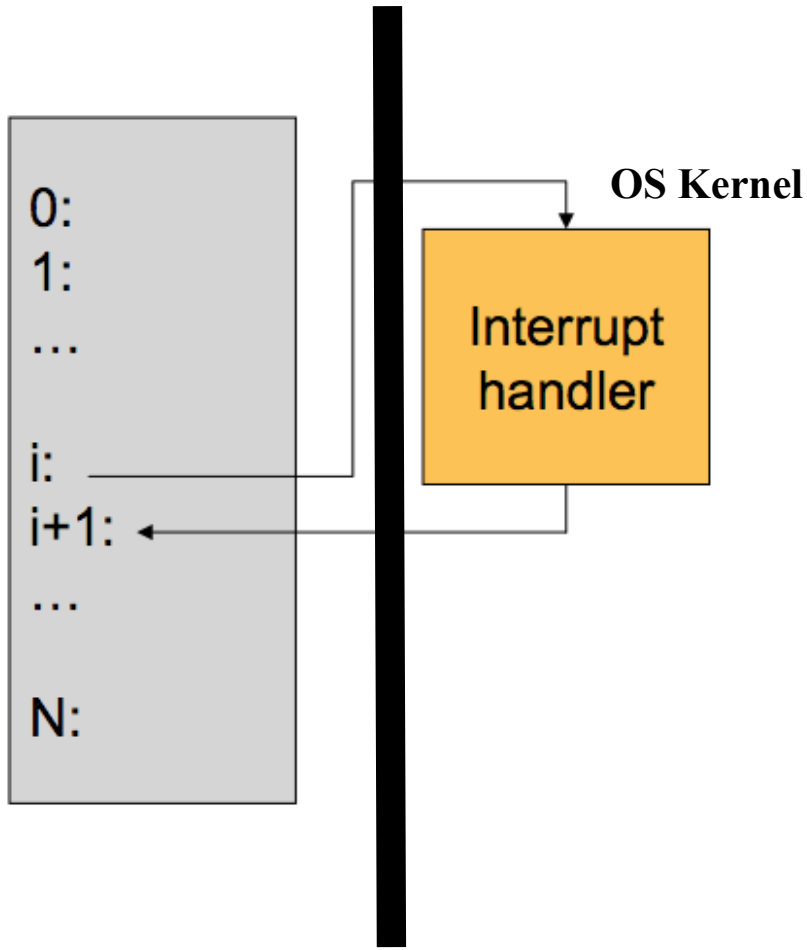
When **decoding** instruction, what to do if bit-pattern doesn't represent a (legal) instruction?

- Halt? (No, but why not?)
- Instead: "**Trap**": fetch next instruction at "predetermined" address in memory. Make sure that you have placed your (OS) code there beforehand

Interrupts and Traps

Application **IS** *program* being executed **IS** at least one *process* (with at least one *thread* each)

- Interrupts
  - Raised by external events
  - CPU resume from the interrupt handler in the kernel
    - switch to next process
      - **iret** instruction: returns by popping return address from stack, and enable interrupts (IA32 instruction set)
- Traps
  - Internal events
  - System calls (syscalls)
  - Also return by **iret**



Now we know how to activate the OS

- **Interrupt** signals, caused by events external to the processor, are sensed by the processor and causes a trap similar to what happens when decoding an ”illegal” instruction
- A program under execution (a process) can cause an **interrupt** by intent
- OS must make sure that **interrupts** are enabled/disabled so they can take place/are discovered and are locked out.
  - When interrupts happen, the OS should be activated (and can then do whatever it wants to do as a result of the interrupt)
- **Interrupts** are *ensured to happen* by requesting recurring wake-up signals from a ”timer” external to the processor
- *Lots of finesse has to be added, but these are some of the basic mechanisms*

Example: Boot OS and Run

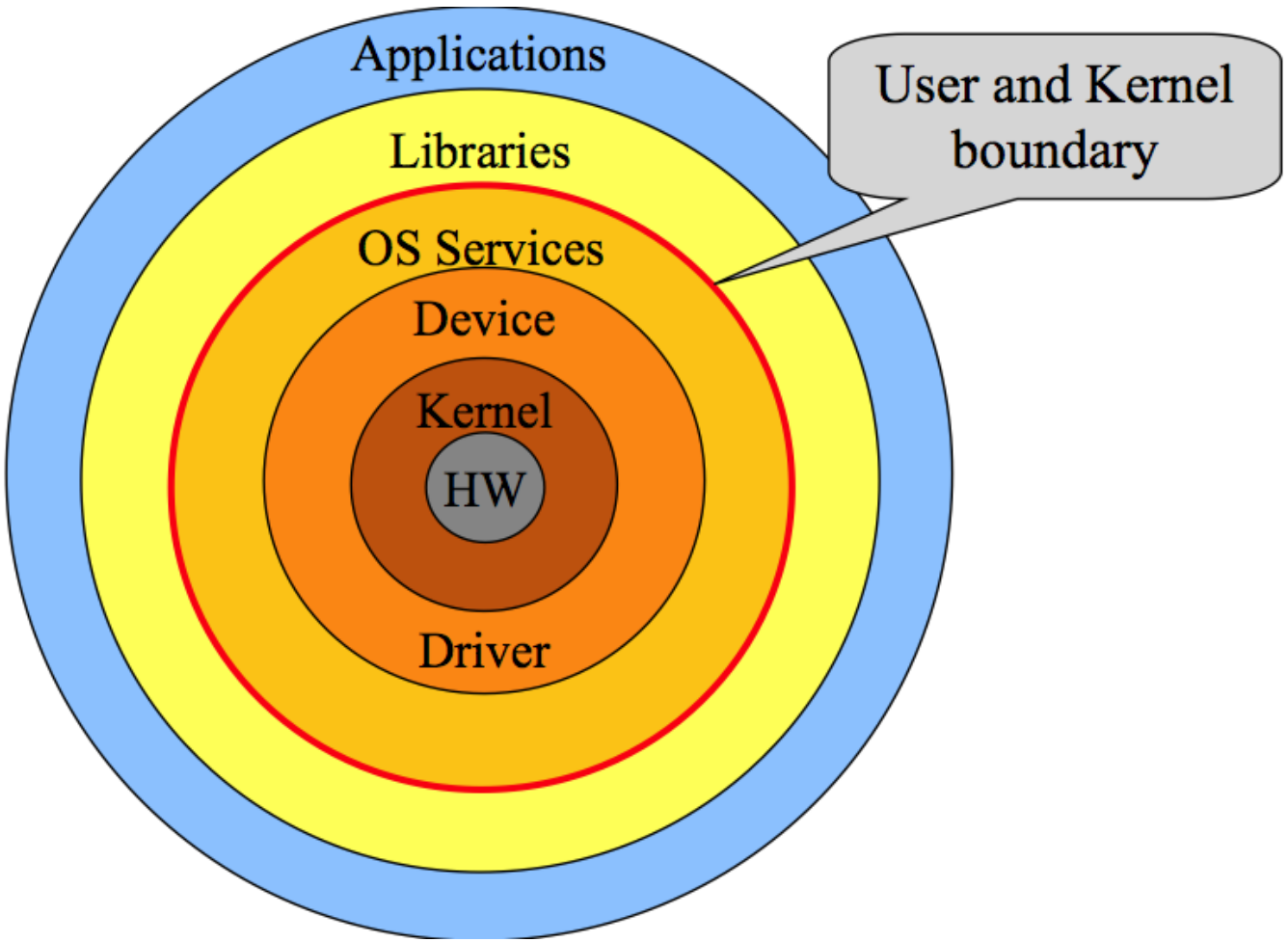
- We assume that the OS **has** been "booted" already
  - power on/start up code reads boot block from HD to memory
  - boot block code reads OS from HD to memory
  - OS is given control

- OS starting user program (first time)
  - Read (already compiled and readied program) image from disk
  - Initialize OS kernel data structures with info about the program
  - ATOMICALLY DO {<Set privilege level "user">; <Load instruction register from start of program>} **% why 'atomically?**

- User program requesting OS service
  - Make a mark "somewhere" (memory, stack, registers) indicating which service is requested and where to find the parameters
  - Execute instruction that is bound to trap in decode

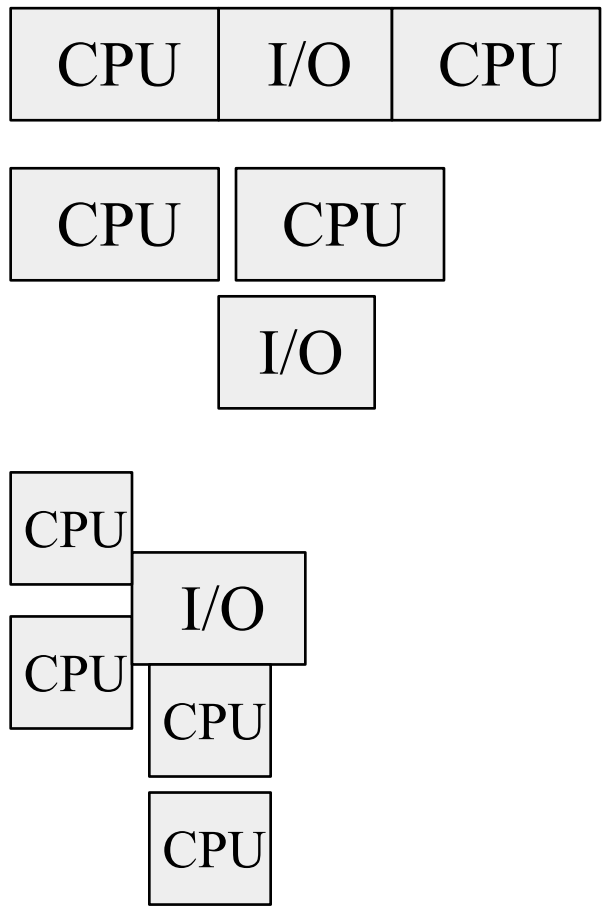
- *Still need mechanism that allows OS to "preempt" user process, OS needs to be activated independently of running program. That can only be achieved "external" to the running program's instruction stream.*

Software "Onion" - but do not take it too seriously



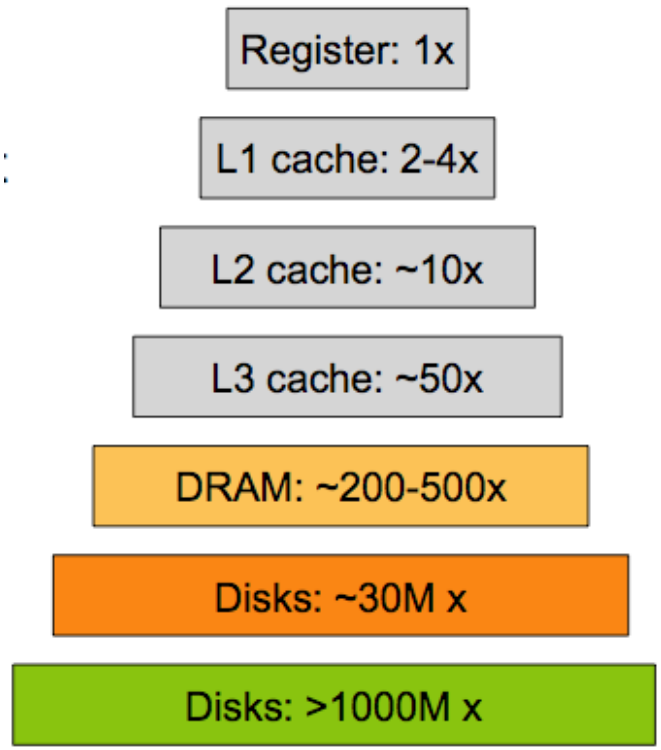
Processor Management

- Goals
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations
- Issues
  - Do not waste CPU resources
  - Synchronization and mutual exclusion
  - Fairness and deadlock free

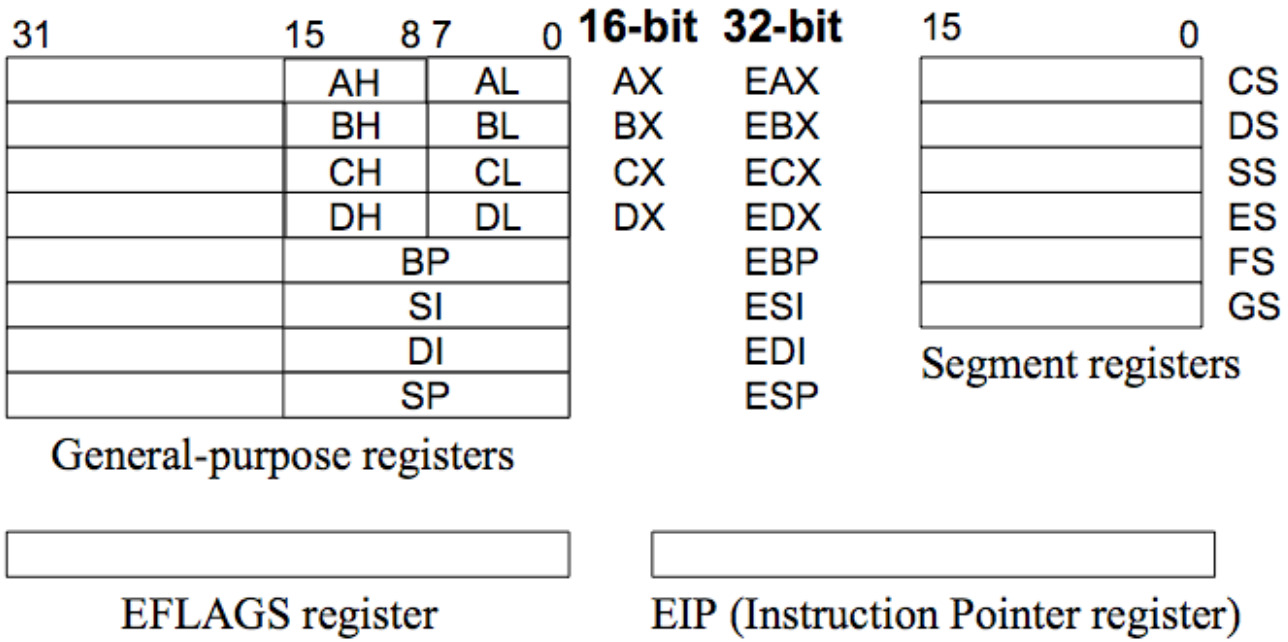


Memory Management

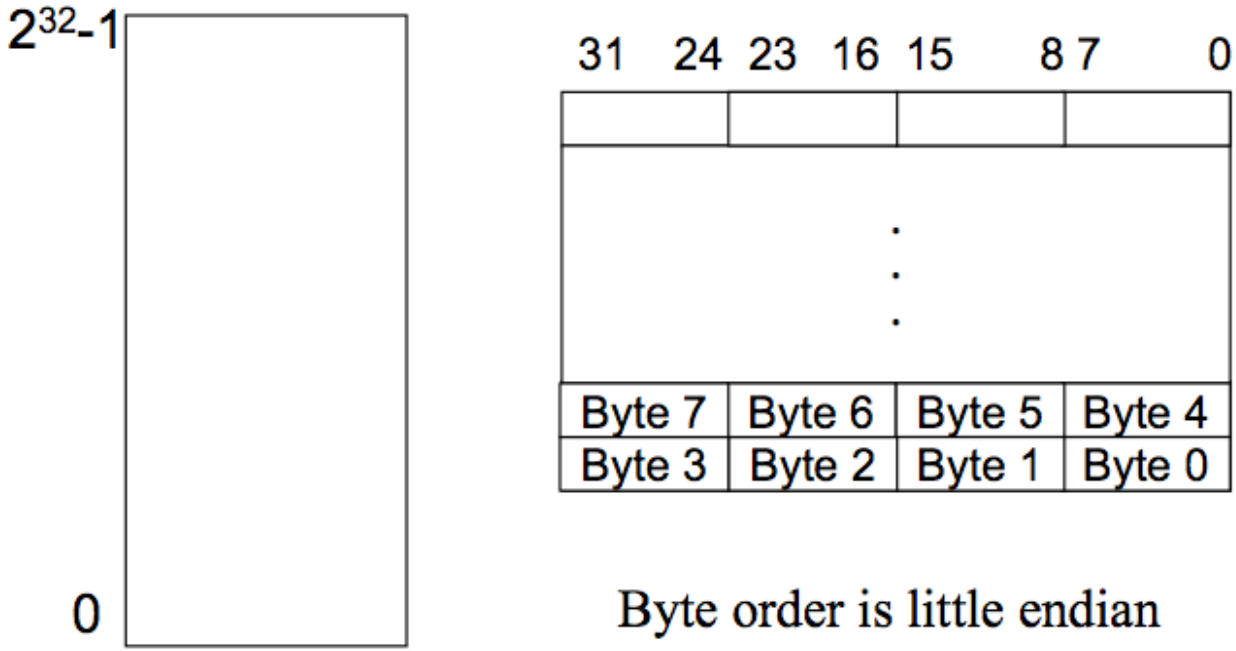
- Goals
  - Support programs to run
  - Allocation and management
  - Transfers from and to secondary storage
- Issues
  - Efficiency & convenience
  - Fairness
  - Protection



IA32 Architecture Registers



IA32 Memory



Intel architecture is “little endian”: little end in first

Power PC (and Sun SPARC) is “***bi***endian”, but Apple is using it as a “big endian”

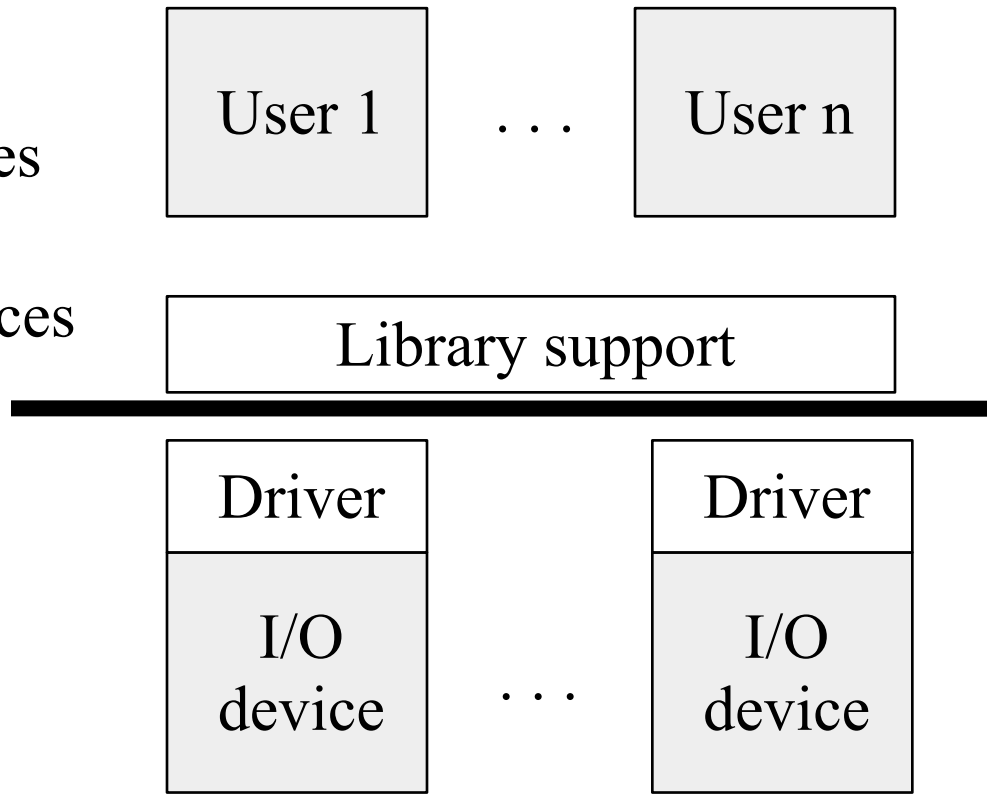
Java: big endian (most significant byte)

Hexadecimal

- **16 decimal is base**
  - 0, 1, 2,…,9, A, B, C, D, E, F
- **C4AFh=50351d**
  - $C*16^3+ 4*16^2+A*16^{1+} F*16^0$
  - $12*16^3+ 4*16^2+10*16^{1+} 15*16^0 = 50351d$
- **2^8-1=11111111b =255d =FFh**
- **2^{16}-1=1111111111111111b=65535d =FFFFh**
- **2^{32}-1=1111111111111111…1b =4294967295d =FFFFFFFFh**

I/O Device Management

- Goals
  - Interactions between devices and applications
  - Ability to plug in new devices
- Issues
  - Efficiency
  - Fairness
  - Protection and sharing



Window Systems

- All in the kernel (Windows)
  - Pro: efficient
  - Con: difficult to develop new services
- All at user level
  - Pro: easy to develop new services
  - Con: protection
- Split between user and kernel (Unix)
  - Kernel: display driver and mouse driver
  - User: the rest

File System

- A typical file system
  - Open a file with authentication
  - Read/write data in files
  - Close a file
- Can the services be moved to user level?