

Architecture, Systems, and Networking in 80 minutes



CSE 490h, Autumn 2008



Father Guido Sarducci

The Five Minute University



- <http://www.youtube.com/watch?v=kO8x8eoU3L4>
- <http://www.cs.washington.edu/info/videos/asx/5minuteU.asx>

Architecture, Systems, and Networking in ~~80~~ 75 minutes: Architecture in 5 minutes

CSE 490h, Autumn 2008



Macro-architectural trends are changing what we can do

■ Moore's Law

- Transistor density doubles every 18 months
 - Greater power
 - Lower cost

■ Processing power, traditionally

- Doubles every 18 months
- 60% improvement each year
- A factor of 100 every decade

- 1980: 1 MHz Apple II+, \$2,000
 - 1980 also 1 MIPS VAX-11/780, \$120,000
- 2006: 3.0 GHz Pentium D, \$800



■ Processing power, recently

- Additional transistors => more cores of the same speed, rather than higher speed
- 2008: Intel Core2 Quad-Core 2.4 GHz, \$800

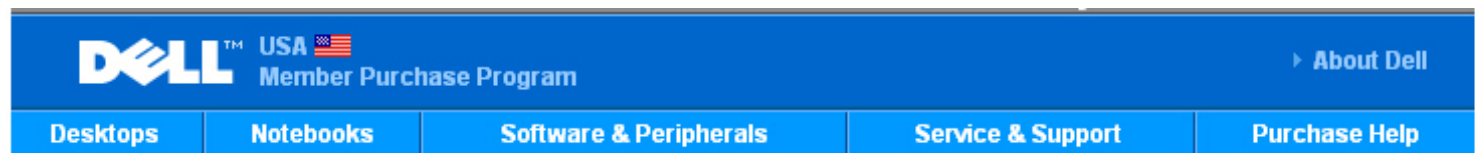
The screenshot shows the Dell website's configuration interface for an XPS 420. At the top, there is a navigation bar with the Dell logo, contact information (1-888-799-3355), a shopping cart icon, and a search bar. Below the navigation bar, a banner recommends Windows Vista Home Premium. The main content area is divided into several sections:

- Navigation:** A progress bar with four steps: 1. Build My Dell, 2. Add My Software & Accessories, 3. Protect My Investment, and 4. Review & Continue. A "SWITCH TO LIST VIEW" link is also present.
- PROCESSING POWER:** A section titled "PROCESSING POWER" with the text "Experience pure genius – the most powerful Intel processor lets you run multiple intense applications at top speeds." It features an image of an Intel processor and a "Genius" badge indicating "Leading quad-core performance".
- SELECT MY PROCESSOR:** A section titled "SELECT MY PROCESSOR" with a "Help Me Choose" link. It lists several processor options with their specifications and prices:
 - Intel® Core™ 2 Duo Processor E8200 (6MB L2 Cache, 2.66GHz, 1333FSB) add \$0
 - Intel® Core™ 2 Q6600 Quad-Core (8MB L2 cache, 2.4GHz, 1066FSB) [Included in Price]**
 - Intel® Core™ 2 Duo Processor E8400 (6MB L2 Cache, 3.0GHz, 1333FSB) add \$0
 - Intel® Core™ 2 Duo Processor E8500 (6MB L2 Cache, 3.16GHz, 1333FSB) [add \$100 or \$3/month¹]
 - Intel® Core™ 2 Extreme QX9650 (3.0GHz, 1333FSB, 12MB L2 Cache) [add \$1,150 or \$35/month¹]
 - Intel® Core™ 2 Q9400 Quad-Core (6MB L2 cache, 2.66GHz, 1333FSB)
- XPS 420 Specifications:** A section titled "XPS 420" showing the starting price of \$799, a financing option of "As low as \$24/month*", and a preliminary ship date of 10/9/2008¹. It includes a "Print Summary" link and a list of system specifications:
 - 800MHz - 4 DIMMs
 - 500GB - 7200RPM, SATA 3.0Gb/s, 16MB Cache
 - Single Drive: 16X CD/DVD burner (DVD+/-RW) w/double layer write capability
 - No Monitor
 - ATI Radeon HD 2400 PRO 128MB
 - Integrated 7.1 Channel Audio
 - No speakers (Speakers are required to hear audio from your system)
 - Dell USB Keyboard
 - Dell Optical USB Mouse
 - No Floppy Drive or Media Reader

Primary memory capacity

Same story, same reason (transistor density)

- 1972: 1MB, \$1,000,000
- 1982: I remember pulling all kinds of strings to get a special deal: 512K of VAX-11/780 memory for \$30,000
- 2005:



The screenshot shows the top navigation bar of the Dell Member Purchase Program website. It features the Dell logo on the left, the text "USA" with a small American flag icon, and "Member Purchase Program" below it. On the right side, there is a link "About Dell". Below the main bar is a secondary navigation bar with five tabs: "Desktops", "Notebooks", "Software & Peripherals", "Service & Support", and "Purchase Help".

Standard Microsoft Windows XP and Windows 2000 operating systems can not address more than 4GB of memory. Large memory configurations for the Dell Precision 470 and 670 are only supported with Red Hat Enterprise WS v3 for Intel EM64T.

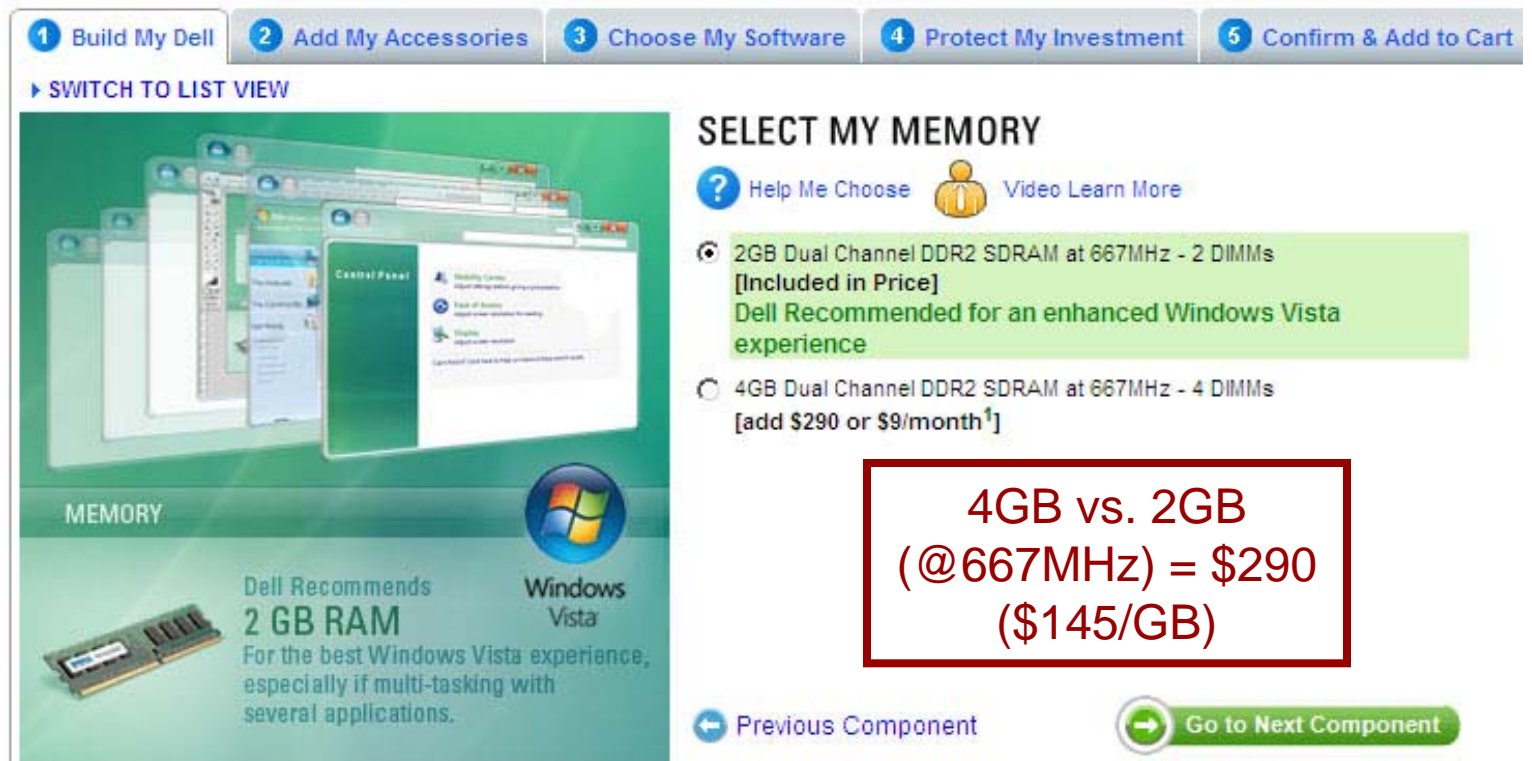
- 512MB, DDR2 SDRAM Memory, 400MHz, ECC (2 DIMMS)
- 1GB, DDR2 SDRAM Memory, 400MHz, ECC (2 DIMMS) [Add \$124.10]
- 1.5GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add \$313.10]
- 2GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add \$466.10]
- 2GB, DDR2 SDRAM Memory, 400MHz, ECC (2 DIMMS) [Add \$700.10]
- 3GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add \$952.10]
- 4GB, DDR2 SDRAM Memory, 400MHz, ECC (6 DIMMS) [Add \$1,267.10]
- 4GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add \$1,537.10]
- 1GB, DDR2 SDRAM Memory, 400MHz, ECC (4 DIMMS) [Add \$124.10]

**4GB vs. 2GB
(@400MHz) = \$800
(\$400/GB)**

| 2007:

1 Build My Dell 2 Add My Accessories 3 Choose My Software 4 Protect My Investment 5 Confirm & Add to Cart

▶ SWITCH TO LIST VIEW



SELECT MY MEMORY

[? Help Me Choose](#) [Video Learn More](#)

- 2GB Dual Channel DDR2 SDRAM at 667MHz - 2 DIMMs
[Included in Price]
Dell Recommended for an enhanced Windows Vista experience
- 4GB Dual Channel DDR2 SDRAM at 667MHz - 4 DIMMs
[add \$290 or \$9/month¹]

4GB vs. 2GB (@667MHz) = \$290 (\$145/GB)

← Previous Component [Go to Next Component](#) →

MEMORY

Dell Recommends **2 GB RAM**
For the best Windows Vista experience, especially if multi-tasking with several applications.

Windows Vista

| 2008:



Chat with us or call: 1-888-799-3355

Dell recommends Windows Vista® Home Premium.

You are here: USA > Home & Home Office

To compare multiple configurations, click the review and continue tab below then click 'Add to Wish List'.

- 1 Build My Dell
- 2 Add My Software & Accessories
- 3 Protect My Investment
- 4 Review & Continue

▶ SWITCH TO LIST VIEW



MEMORY

Help improve multi-tasking, speed up gaming, and take your PC's performance even higher with increased RAM.

SELECT MY MEMORY

- 3GB Dual Channel DDR2 SDRAM at 800MHz - 4 DIMMs
[Included in Price]
- 4GB Dual Channel DDR2 SDRAM at 800MHz - 4 DIMMs
[add \$49 or \$1/month¹]
Maximize Your Memory Performance

4GB vs. 3GB
(@800MHz) = \$49
(\$49/GB)

← Previous Component

→ Go to Next Component

→ Buy Now




■ Disk capacity, 1975-1989

- doubled every 3+ years
- 25% improvement each year
- factor of 10 every decade
- Still exponential, but far less rapid than processor performance

■ Disk capacity since 1990

- doubling every 12 months
- 100% improvement each year
- factor of 1000 every decade
- 10x as fast as processor performance!

- 
- Only a few years ago, we purchased disks by the megabyte (and it hurt!)
 - Today, 1 GB (a billion bytes) costs ~~\$1~~ ~~\$0.50~~ \$0.25 from Dell (except you have to buy in increments of ~~40~~ ~~80~~ 250 GB)
 - => 1 TB costs ~~\$1K~~ ~~\$500~~ \$250, 1 PB costs ~~\$1M~~ ~~\$500K~~ \$250K



■ Optical bandwidth today

- Doubling every 9 months
- 150% improvement each year
- Factor of 10,000 every decade
- 10x as fast as disk capacity!
- 100x as fast as processor performance!!

Architecture, Systems, and Networking in ~~80~~ 75 minutes: Systems in 40 minutes

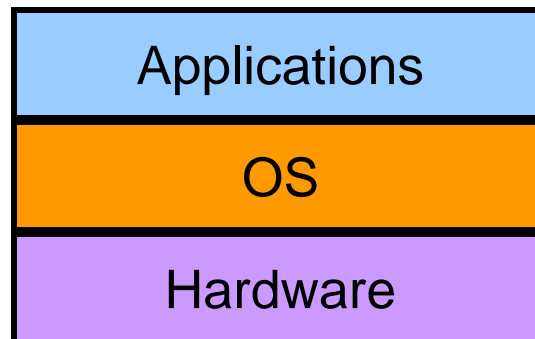
CSE 490h, Autumn 2008



What is an Operating System?

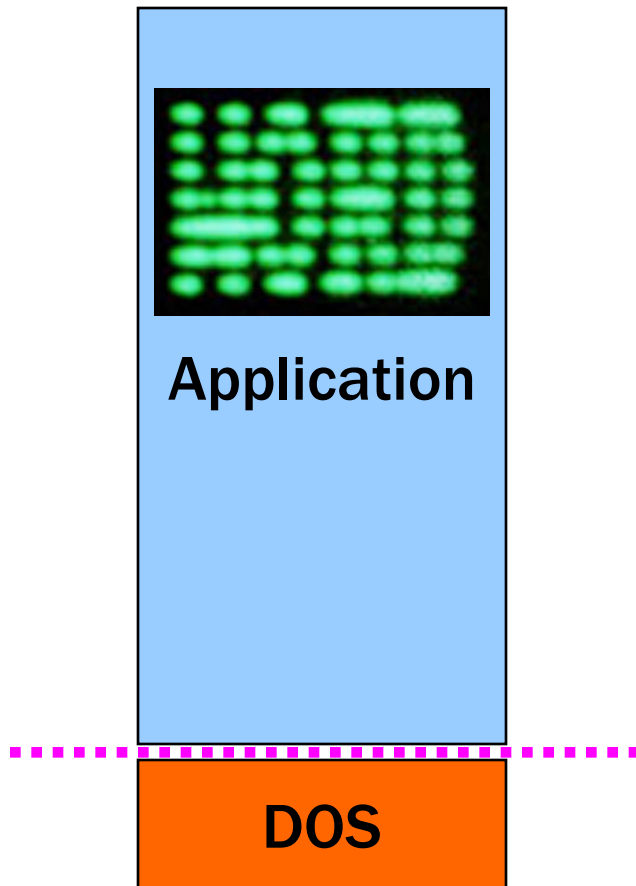
- An operating system (OS) is:

- A software layer to abstract away and manage details of hardware resources
- A set of utilities to simplify application development

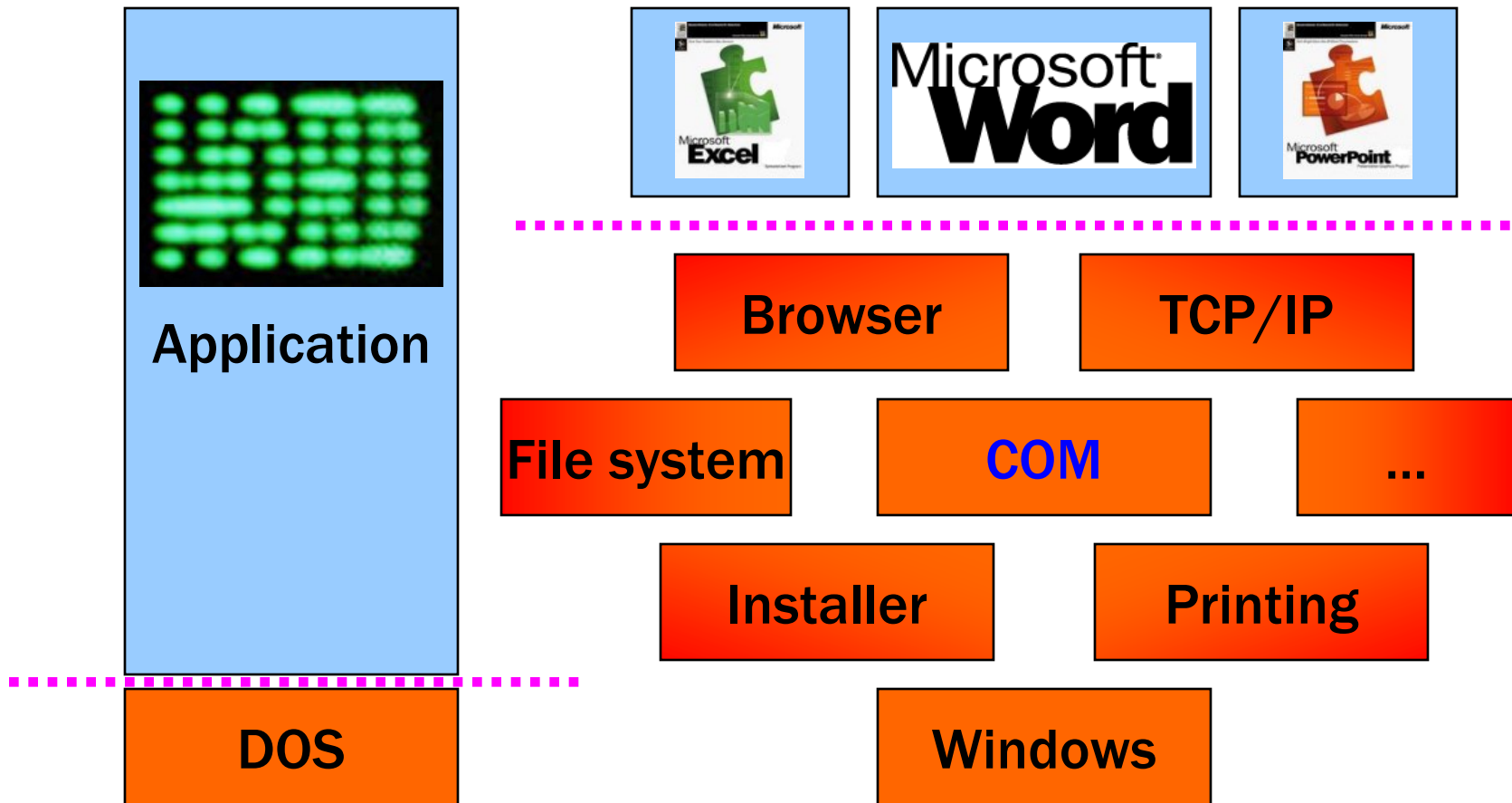


- "All the code you didn't have to write" in order to implement your application

What is Windows?



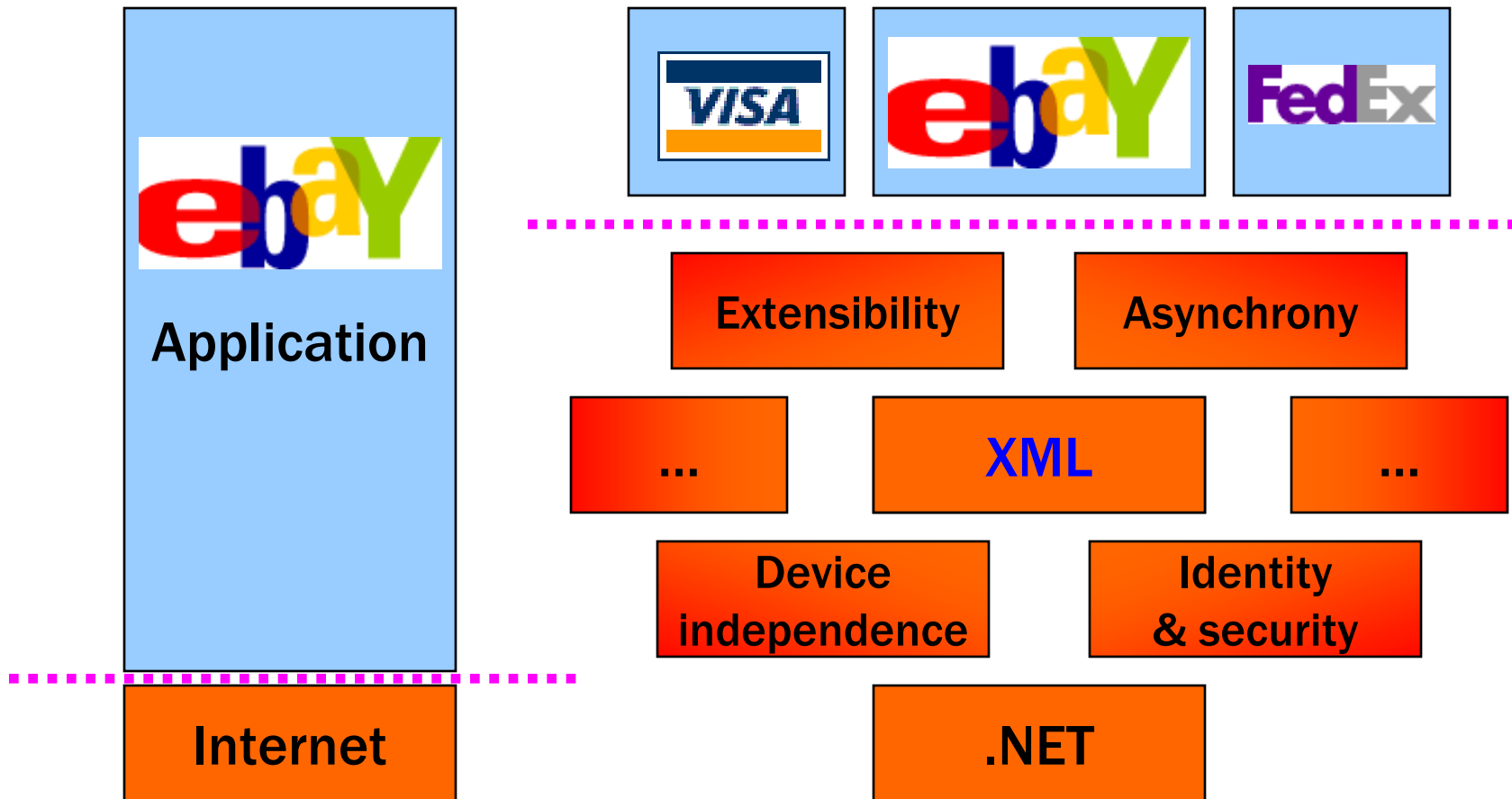
What is Windows?



What is .NET?




What is .NET?



The major OS issues



- **structure:** how is the OS organized?
- **sharing:** how are resources shared across users?
- **naming:** how are resources named (by users or programs)?
- **security:** how is the integrity of the OS and its resources ensured?
- **protection:** how is one user/program protected from another?
- **performance:** how do we make it all go fast?
- **reliability:** what happens if something goes wrong (either with hardware or with a program)?
- **extensibility:** can we add new features?
- **communication:** how do programs exchange information, including across a network?

- 
- **concurrency:** how are parallel activities (computation and I/O) created and controlled?
 - **scale:** what happens as demands or resources increase?
 - **persistence:** how do you make data last longer than program executions?
 - **distribution:** how do multiple computers interact with each other?
 - **accounting:** how do we keep track of resource usage, and perhaps charge for it?

There are *tradeoffs*, not right and wrong answers!

Architectural features supporting OS's



- timer (clock) operation
- synchronization instructions (e.g., atomic test-and-set)
- memory protection
- I/O control operations
- interrupts and exceptions
- protected modes of execution (kernel vs. user)
- privileged instructions
- system calls (and software interrupts)

Privileged instructions



- Some instructions are restricted to the OS
 - known as **privileged** instructions
- E.g., only the OS can:
 - Directly access I/O devices (disks, network cards)
 - why?
 - Manipulate memory state management
 - page table pointers, TLB loads, etc.
 - why?
 - Manipulate special 'mode bits'
 - interrupt priority level
 - why?
 - Execute the halt instruction
 - why?

Kernel and user mode



- So how does the processor know if a privileged instruction should be executed?
 - The architecture must support at least two modes of operation: **kernel mode** and **user mode**
 - VAX, x86 support 4 protection modes
 - Mode is set by status bit in a protected processor register
 - user programs execute in user mode
 - OS executes in kernel mode (OS == kernel)
- Privileged instructions can only be executed in kernel mode
 - What happens if user mode attempts to execute a privileged instruction?

Crossing protection boundaries

- So how do user programs do something privileged?
 - E.g., how can you write to a disk if you can't execute I/O instructions?
- User programs must call an OS procedure
 - OS defines a sequence of **system calls**
 - How does the user-mode to kernel-mode transition happen?
- There must be a system call instruction, which:
 - Causes an exception (throws a **software interrupt**), which vectors to a kernel handler
 - Passes a parameter indicating which system call to invoke
 - Saves caller's state (registers, mode bit) so they can be restored
 - OS must verify caller's parameters (e.g., pointers)
 - Must be a way to return to user mode once done

Other events can cause the OS to get control



- Two main types of events: interrupts and exceptions
 - Exceptions are caused by software executing instructions
 - | e.g., the x86 'int' instruction
 - | e.g., a page fault, or an attempted write to a read-only page
 - | an expected exception is a "trap", unexpected is a "fault"
 - Interrupts are caused by hardware devices
 - | e.g., device finishes I/O
 - | e.g., timer fires



- Kernel defines handlers for each event type

- Specific types are defined by the architecture
 - | e.g.: timer event, I/O interrupt, system call trap
- When the processor receives an event of a given type, it
 - | transfers control to handler within the OS
 - | handler saves program state (PC, regs, etc.)
 - | handler functionality is invoked
 - | handler restores program state, returns to program

I/O



■ Issues:

- how does the kernel start an I/O?
 - | special I/O instructions
 - | memory-mapped I/O
- how does the kernel notice an I/O has finished?
 - | polling
 - | interrupts

■ Interrupts are basis for asynchronous I/O

- device performs an operation asynchronously to CPU
- device sends an interrupt signal on bus when done
- in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
 - | who populates the vector table, and when?
- CPU switches to address indicated by vector index specified by interrupt signal

Timers



- How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
 - use a hardware timer that generates a periodic interrupt
 - before it transfers to a user program, the OS loads the timer with a time to interrupt
 - "quantum" - how big should it be set?
 - when timer fires, an interrupt transfers control back to OS
 - at which point OS must decide which program to schedule next
 - very interesting policy question: we'll dedicate a class to it
- Should the timer be privileged?
 - for reading or for writing?

Memory protection



- OS must protect user programs from each other
 - maliciousness, ineptitude
- OS must also protect itself from user programs
 - integrity and security
 - what about protecting user programs from OS?
- Paging, segmentation, virtual memory
 - page tables, page table pointers
 - translation lookaside buffers (TLBs)
 - page fault handling

More on this shortly!

Processes



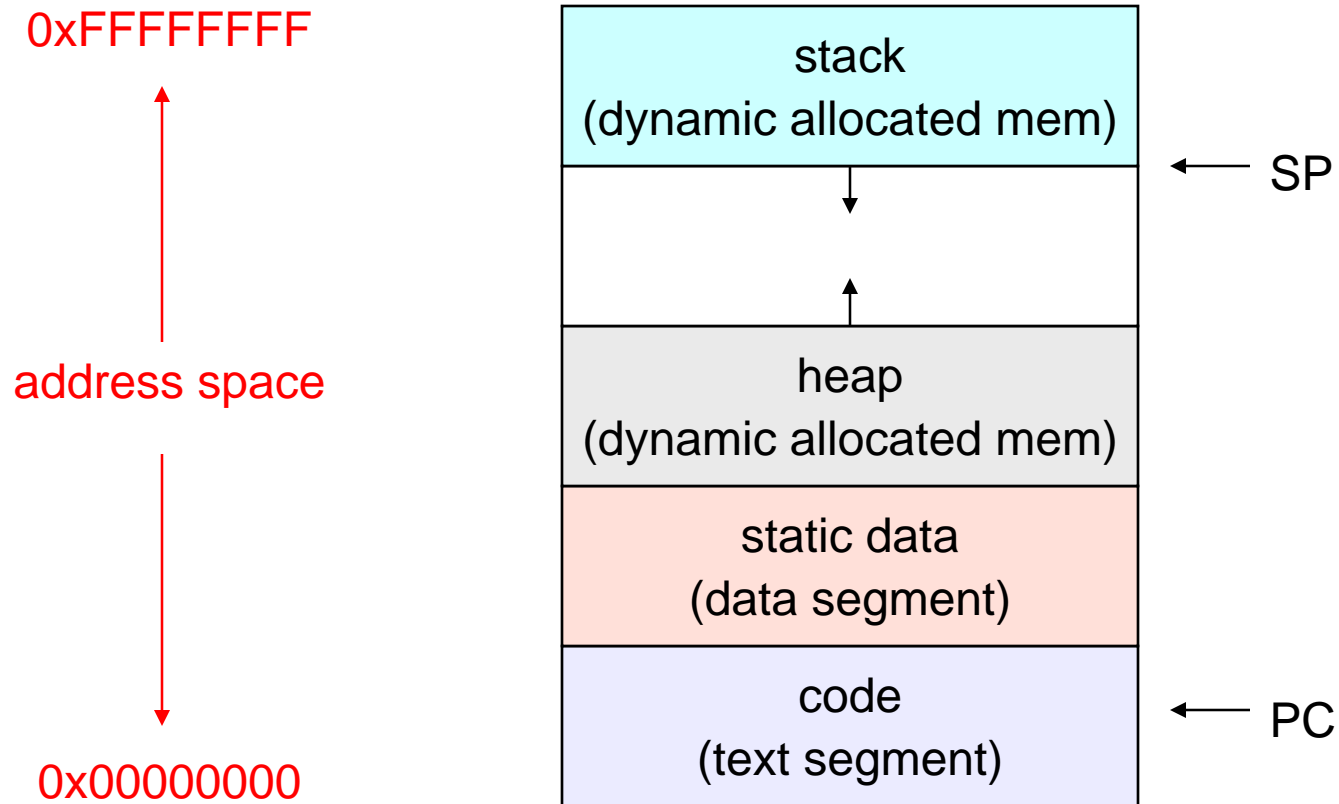
- An OS executes many kinds of activities:
 - users' programs
 - batch jobs or scripts
 - system programs
 - | print spoolers, name servers, file servers, network daemons, ...
- Each of these activities is encapsulated in a **process**
 - a process includes the execution **context**
 - | PC, registers, OS resources (e.g., open files), etc...
 - | plus the program itself (code, data, stack)
 - the OS's process module manages these processes
 - | creation, destruction, scheduling, ...

What's in a process?



- A process consists of (at least):
 - an address space
 - the code for the running program
 - the data for the running program
 - an execution stack and stack pointer (SP)
 - traces state of procedure calls made
 - the program counter (PC), indicating the next instruction
 - general-purpose processor registers and their values
 - a set of OS resources
 - open files, network connections, sound channels, ...
- In other words, it's all the stuff you need to run the program
 - or to re-start it, if it's interrupted at some point

A process's address space



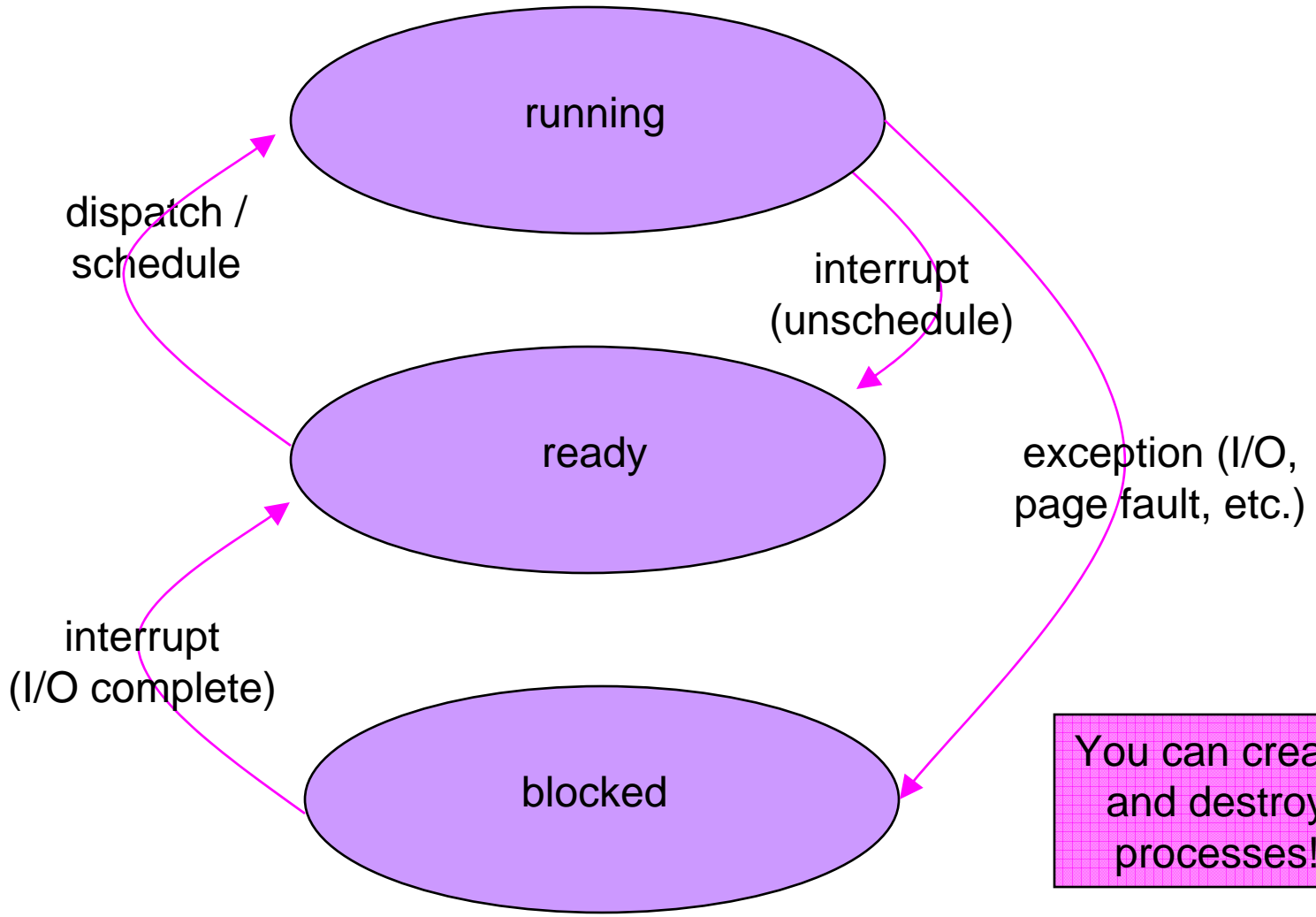
The process control block



- There's a data structure called the process control block (PCB) that holds all this stuff
 - The PCB is identified by an integer process ID (PID)
- OS keeps all of a process's hardware execution state in the PCB when the process isn't running
 - PC, SP, registers, etc.
 - when a process is unscheduled, the state is transferred out of the hardware into the PCB
- Note: It's natural to think that there must be some esoteric techniques being used
 - fancy data structures that'd you'd never think of yourself
- Wrong! It's pretty much just what you'd think of!

Process states

- Each process has an **execution state**, which indicates what it is currently doing
 - ready: waiting to be assigned to CPU
 - could run, but another process has the CPU
 - running: executing on the CPU
 - is the process that currently controls the CPU
 - pop quiz: how many processes can be running simultaneously?
 - waiting: waiting for an event, e.g., I/O
 - cannot make progress until event happens
- As a process executes, it moves from state to state
 - UNIX: run **ps**, STAT column shows current state
 - which state is a process in most of the time?



You can create and destroy processes!

Synchronization



- One process may need to wait for another
- A process may be interrupted at an inopportune moment
- Synchronization is necessary
- Sections of code must be able to be executed atomically - critical sections
- Locks, semaphores, monitors, ...

Threads



- Imagine a web server, which might like to handle multiple requests concurrently
 - While waiting for the credit card server to approve a purchase for one client, it could be retrieving the data requested by another client from disk, and assembling the response for a third client from cached information
- Imagine a web client (browser), which might like to initiate multiple requests concurrently
 - The CSE home page has 46 "src= ..." html commands, each of which is going to involve a lot of sitting around! Wouldn't it be nice to be able to launch these requests concurrently?
- Imagine a parallel program running on a multiprocessor, which might like to employ "physical concurrency"
 - For example, multiplying a large matrix - split the output matrix into k regions and compute the entries in each region concurrently using k processors

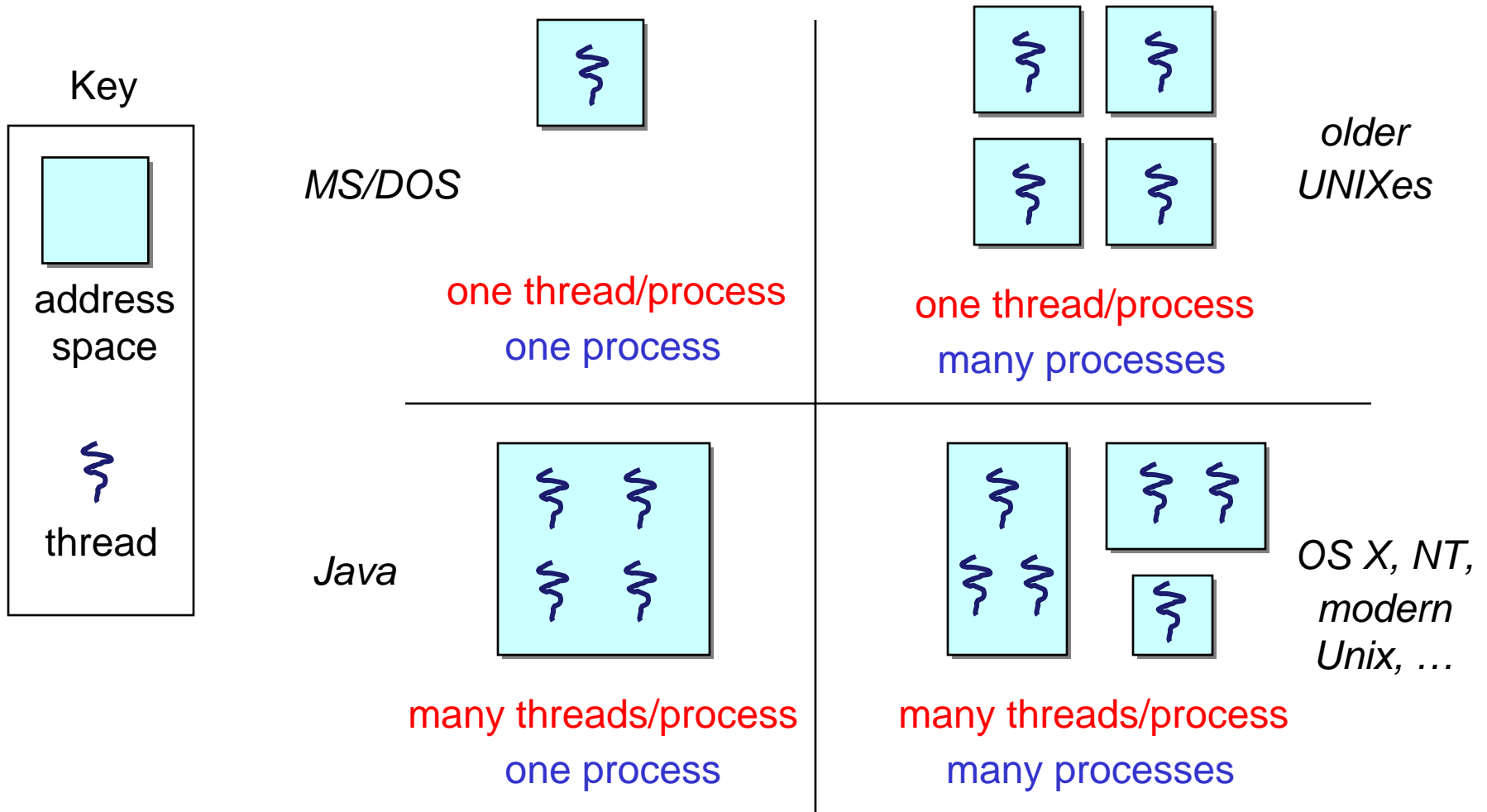
What's needed in these situations?

- In each of these examples of concurrency (web server, web client, parallel program):
 - Everybody wants to run the same code
 - Everybody wants to access the same data
 - Everybody has the same privileges
 - Everybody uses the same resources (open files, network connections, etc.)
- But you'd like to have multiple hardware execution states:
 - an execution stack and stack pointer (SP)
 - traces state of procedure calls made
 - the program counter (PC), indicating the next instruction
 - a set of general-purpose processor registers and their values

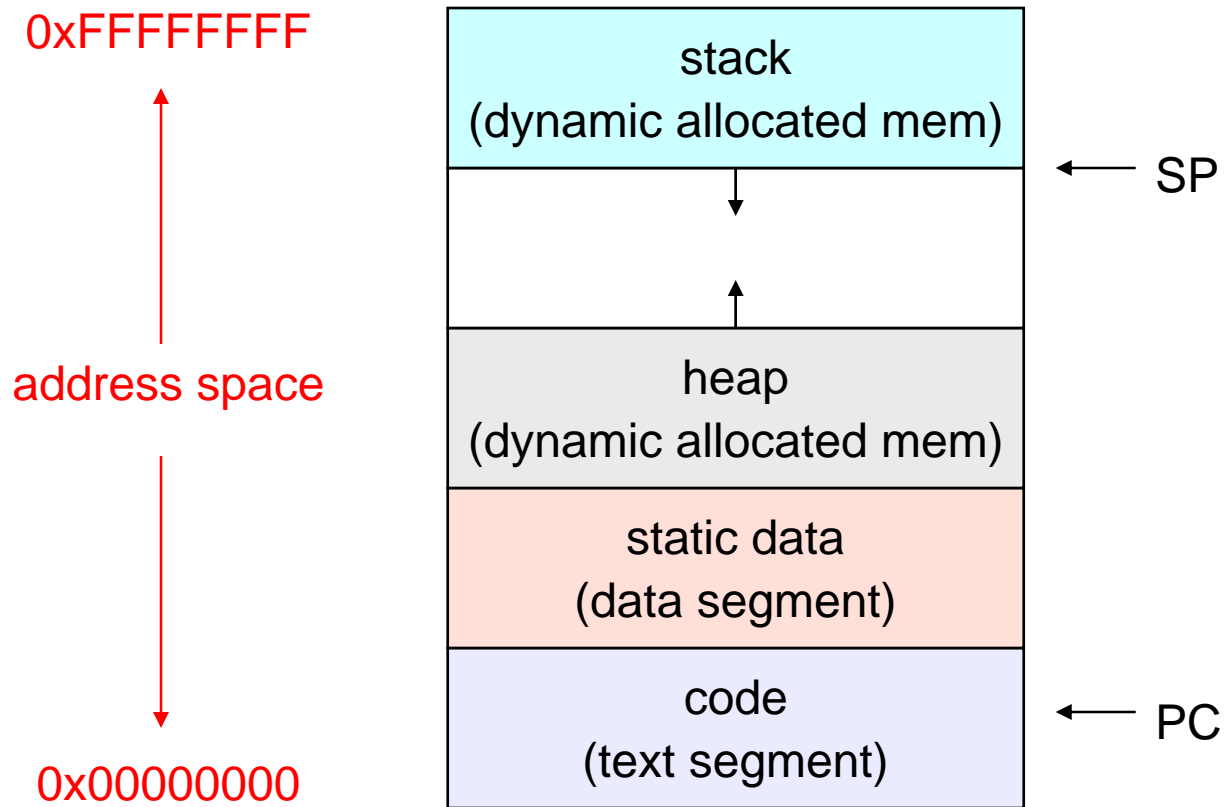
Threads and processes

- Most modern OS's (OS X, NT, modern UNIX) therefore support two entities:
 - the process, which defines the address space and general process attributes (such as open files, etc.)
 - the thread, which defines a sequential execution stream within a process
- A thread is bound to a single process / address space
 - address spaces, however, can have multiple threads executing within them
 - sharing data between threads is cheap: all see the same address space
 - creating threads is cheap too!
- Threads become the unit of scheduling
 - processes / address spaces are just containers in which threads execute

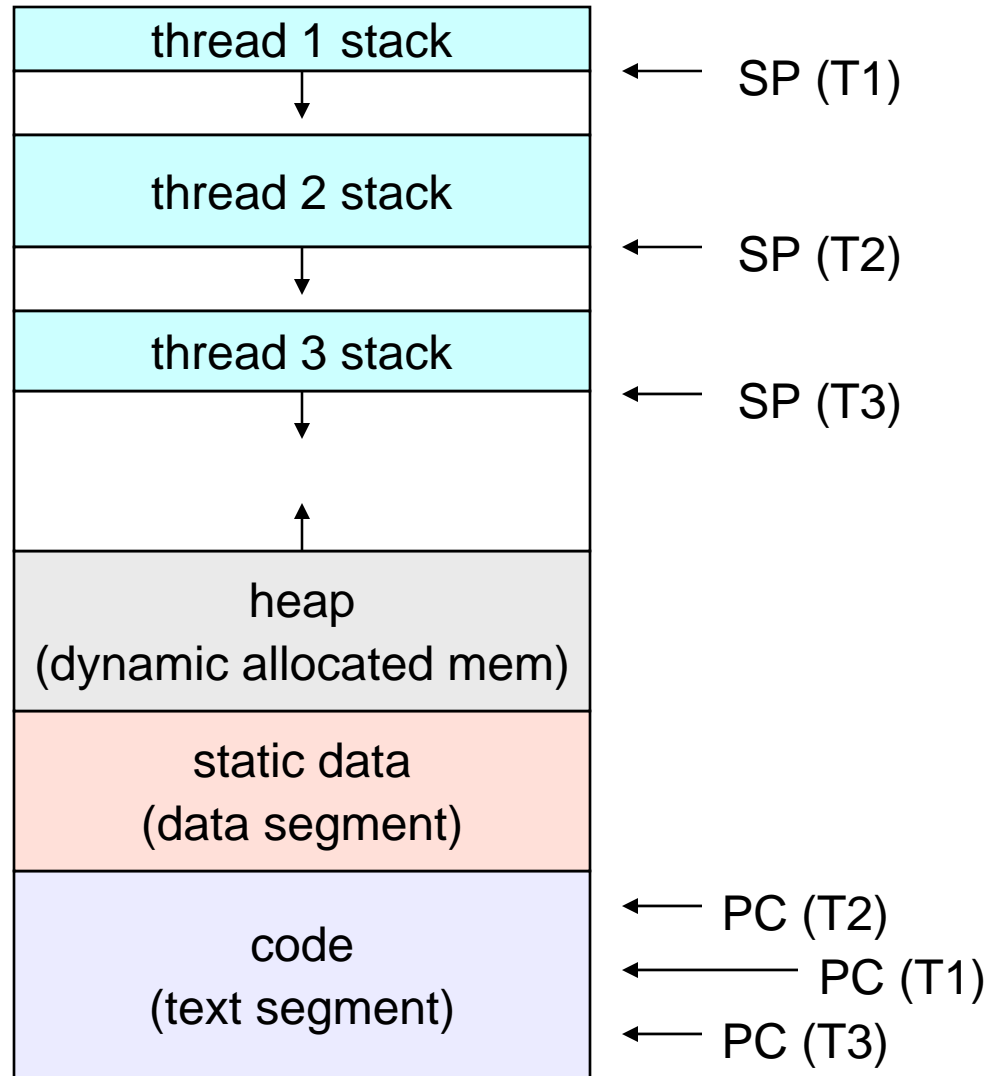
The design space



(old) Process address space



(new) Process address space with threads



Memory management - paging

- Processes view memory as a contiguous address space from bytes 0 through N
 - virtual address space (VAS)
- Logically divided into **pages** of fixed size (e.g., 4KB)
- Pages are scattered across physical memory **page frames** - not contiguous
 - virtual-to-physical mapping
 - this mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
 - the virtual address 0xDEADBEEF maps to different physical addresses for different processes

Address translation

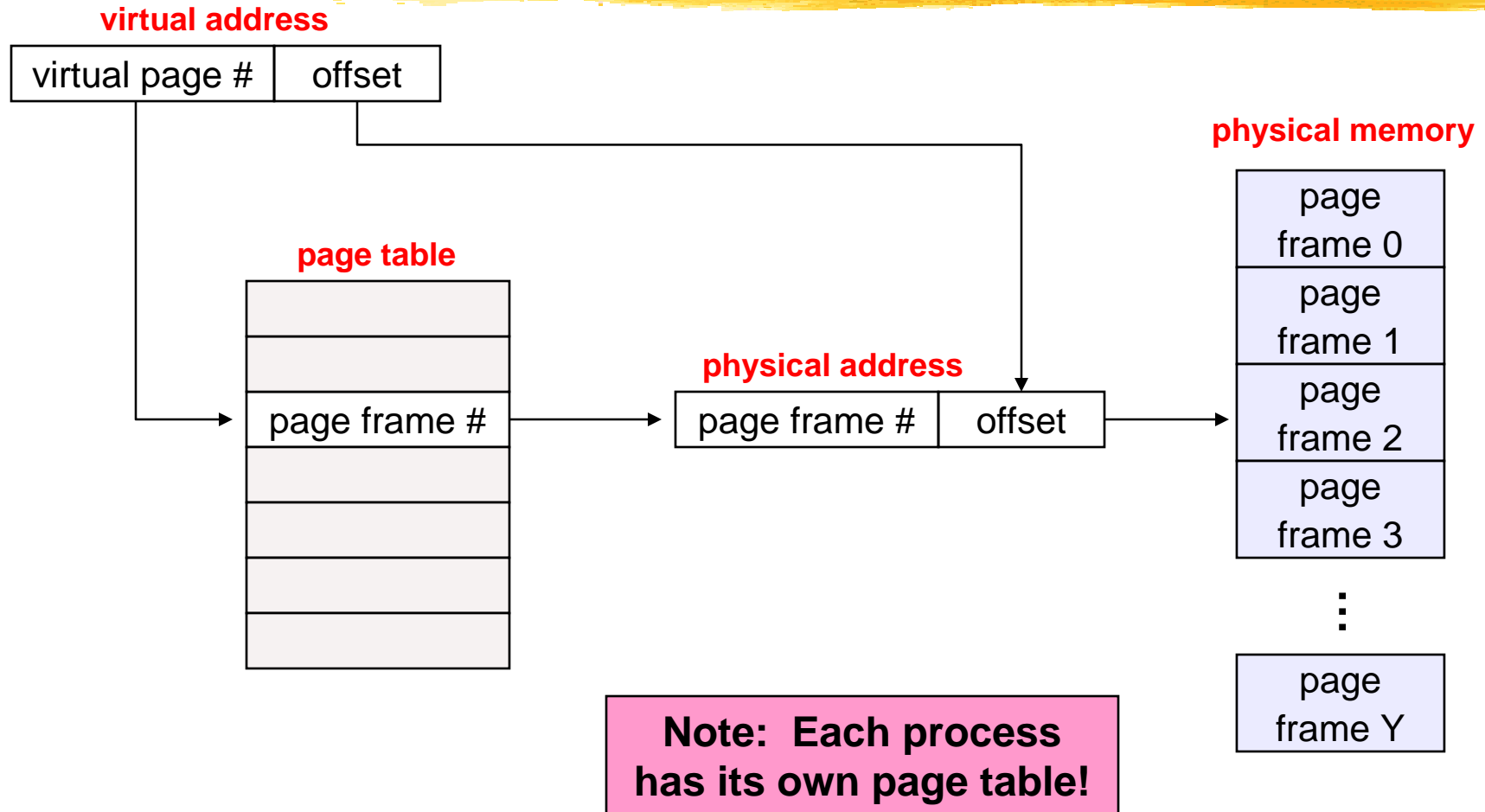
■ Translating virtual addresses

- a **virtual address** has two parts: virtual **page number** & **offset**
- virtual page number (VPN) is index into a **per-process** page table
- page table entry contains page frame number (PFN)
- physical address is PFN::offset

■ Page tables

- managed by the OS
- map virtual page number (VPN) to page frame number (PFN)
 - VPN is simply an index into the page table
- one page table entry (PTE) per page in virtual address space
 - i.e., one PTE per VPN

Mechanics of address translation



Example of address translation

- Assume 32 bit addresses
 - assume page size is 4KB (4096 bytes, or 2^{12} bytes)
 - VPN is 20 bits long (2^{20} VPNs), offset is 12 bits long
- Let's translate virtual address 0x13325328
 - VPN is 0x13325, and offset is 0x328
 - assume page table entry 0x13325 contains value 0x03004
 - page frame number is 0x03004
 - VPN 0x13325 maps to PFN 0x03004
 - physical address = PFN::offset = 0x03004328

Page Table Entries (PTEs)



■ PTE's control mapping

- the **valid bit** says whether or not the PTE can be used
 - | says whether or not a virtual address is valid
 - | it is checked each time a virtual address is used
- the **referenced bit** says whether the page has been accessed
 - | it is set when a page has been read or written to
- the **modified bit** says whether or not the page is dirty
 - | it is set when a write to the page has occurred
- the **protection bits** control which operations are allowed
 - | read, write, execute
- the **page frame number** determines the physical page
 - | physical page start address = PFN

Paged virtual memory



- All the pages of an address space do not need to be resident in memory
 - the full (used) address space exists on secondary storage (disk) in page-sized blocks
 - the OS uses main memory as a (page) cache
 - a page that is needed is transferred to a free page frame
 - if there are no free page frames, a page must be evicted
 - evicted pages go to disk (only need to write if they are dirty)
 - all of this is transparent to the application (except for performance ...)
 - managed by hardware and OS
- Traditionally called paged virtual memory

Page faults



- What happens when a process references a virtual address in a page that has been evicted?
 - when the page was evicted, the OS set the PTE as invalid and noted the disk location of the page in a data structure (that looks like a page table but holds disk addresses)
 - when a process tries to access the page, the invalid PTE will cause an exception (**page fault**) to be thrown
 - | OK, it's actually an interrupt!
 - the OS will run the page fault handler in response
 - | handler uses the "like a page table" data structure to locate the page on disk
 - | handler reads page into a physical frame, updates PTE to point to it and to be valid
 - | OS restarts the faulting process
 - | **there are a million and one details ...**

How do you "load" a program?

- Create process descriptor (process control block)
- Create page table
- Put address space image on disk in page-sized chunks
- Build page table (pointed to by process descriptor)
 - all PTE valid bits 'false'
 - an analogous data structure indicates the disk location of the corresponding page
 - when process starts executing:
 - instructions immediately fault on both code and data pages
 - faults taper off, as the necessary code/data pages enter memory

Oh, man, how can any of this possibly work?



■ Locality!

■ temporal locality

- | locations referenced recently tend to be referenced again soon

■ spatial locality

- | locations near recently references locations are likely to be referenced soon (think about why)

■ Locality means paging can be infrequent

- once you've paged something in, it will be used many times

- on average, you use things that are paged in

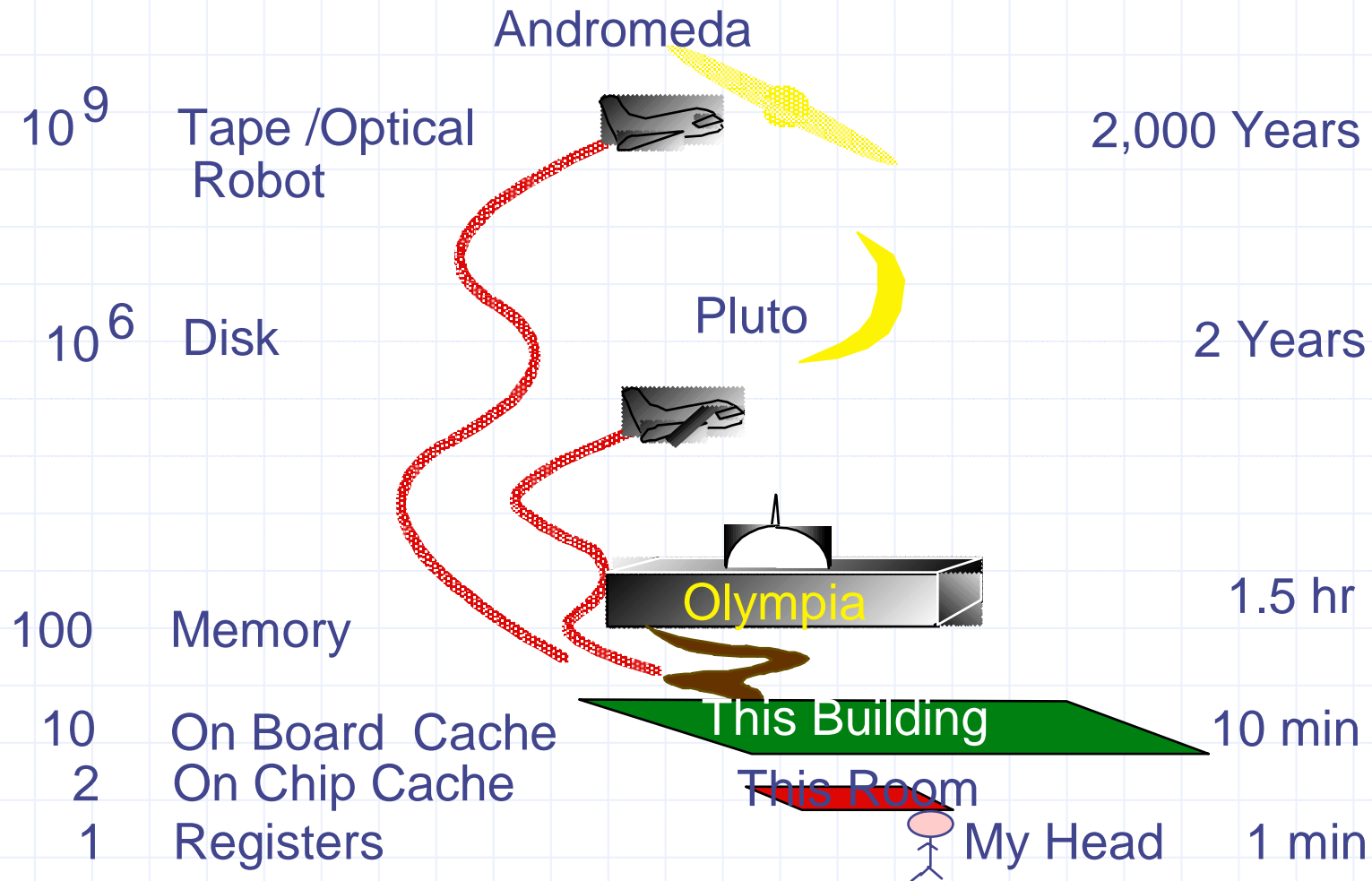
- but, this depends on many things:

- | degree of locality in the application

- | page replacement policy and application reference pattern

- | amount of physical memory vs. application "footprint" or "working set"

Storage Latency: How Far Away is the Data?

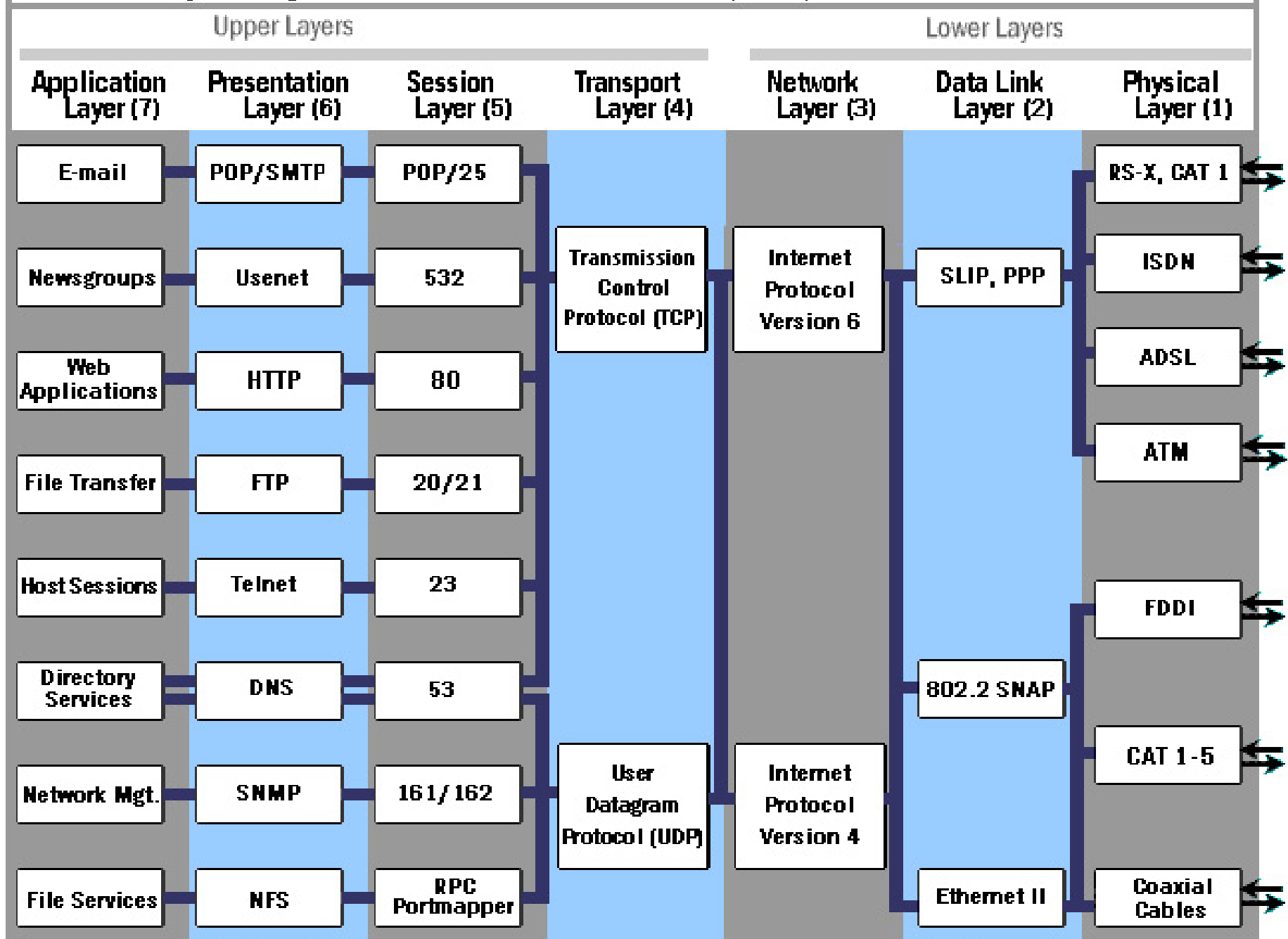


Architecture, Systems, and Networking in ~~80~~ 75 minutes: Networking in 30 minutes

CSE 490h, Autumn 2008

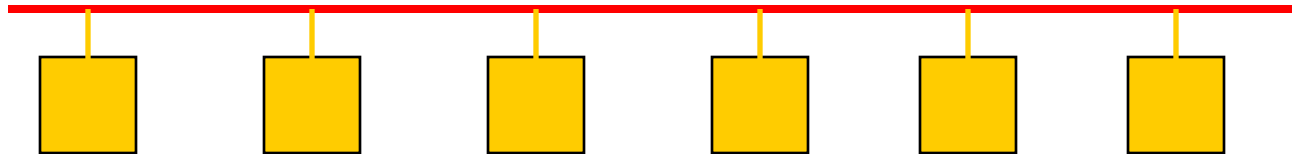


Open Systems Interconnection (OSI) Reference Model



Data link layer: Ethernet

- Broadcast network



- CSMA-CD: Carrier Sense Multiple Access with Collision Detection

- Analogy: Standing in a circle, drinking beer and telling stories

- Packetized - fixed

- Every computer has a unique physical address

- 00-08-74-C9-C8-7E



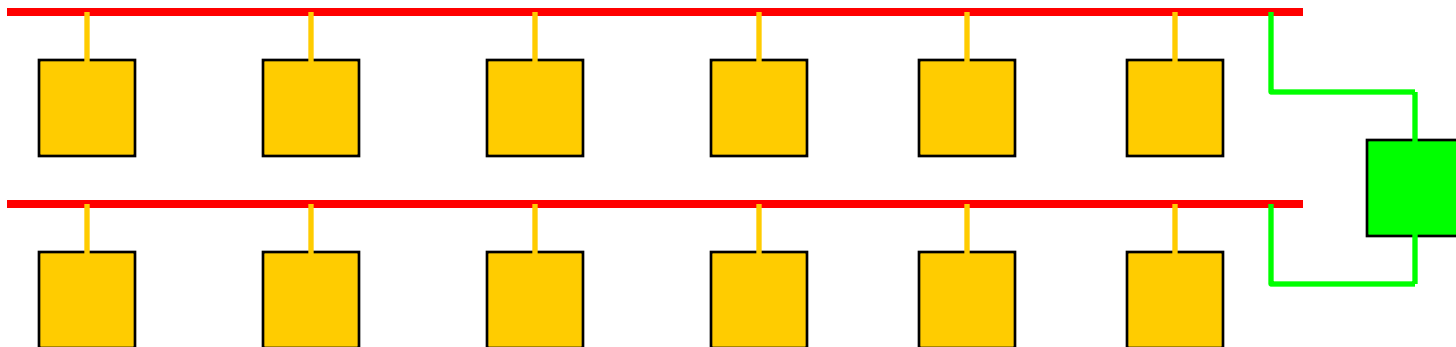
- Packet format



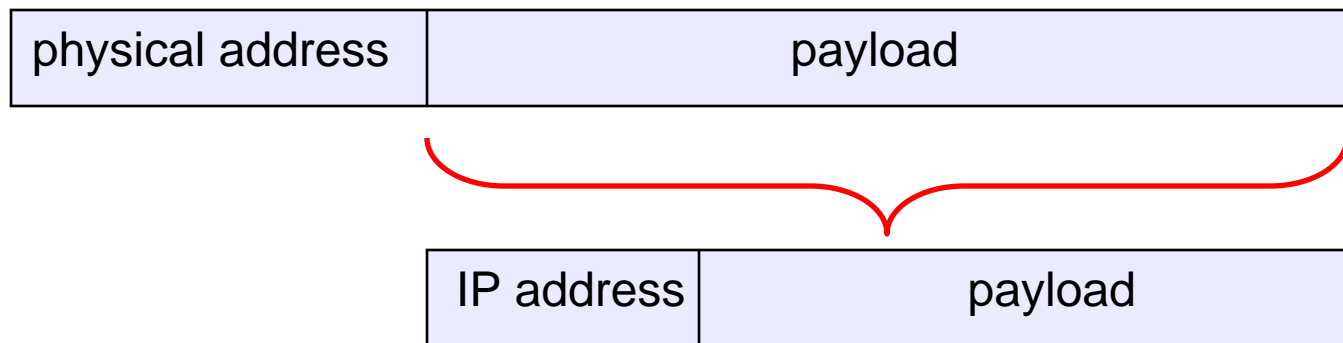
- Interface listens for its address, interrupts OS when a packet is received


Network layer: IP

- Internet Protocol (IP)
 - Routes packets across multiple networks, from source to destination
- Every computer has a unique Internet address
 - 172.30.192.251
- Individual networks are connected by **routers** that have physical addresses (and interfaces) on each network



- A really hairy protocol lets any node on a network find the physical address on that network of a router that can get a packet one step closer to its destination
- Packet format

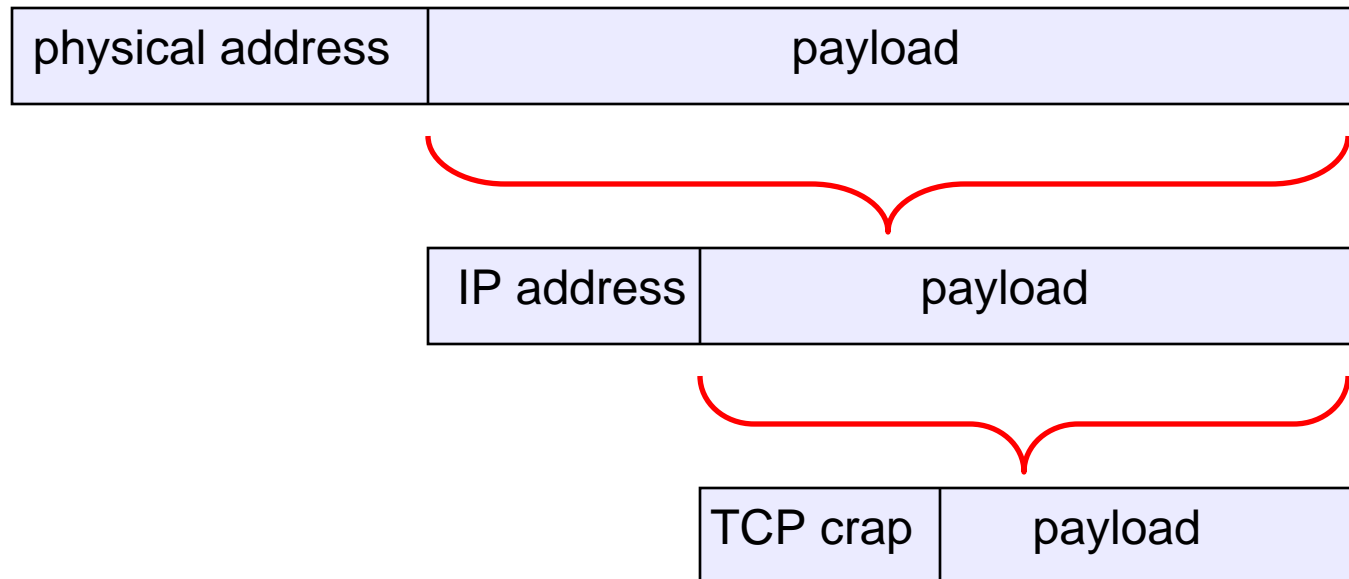


- 
- A separate really hairy protocol, DNS (the Domain Name Service), maps from intelligible names (lazowska.org) to IP addresses (209.180.207.60)
 - So to send a packet to a destination
 - Use DNS to convert domain name to IP address
 - Prepare IP packet, with payload prefixed by IP address
 - Determine physical address of appropriate router
 - Encapsulate IP packet in Ethernet packet with appropriate physical address
 - Bombs away!
 - Detail: port number gets you to a specific address space on a system

Transport layer: TCP

■ TCP: Transmission Control Protocol

- Manages to achieve reliable multi-packet messages out of unreliable single-packet datagrams
- Analogy: Sending a book via postcards - *what's required?*

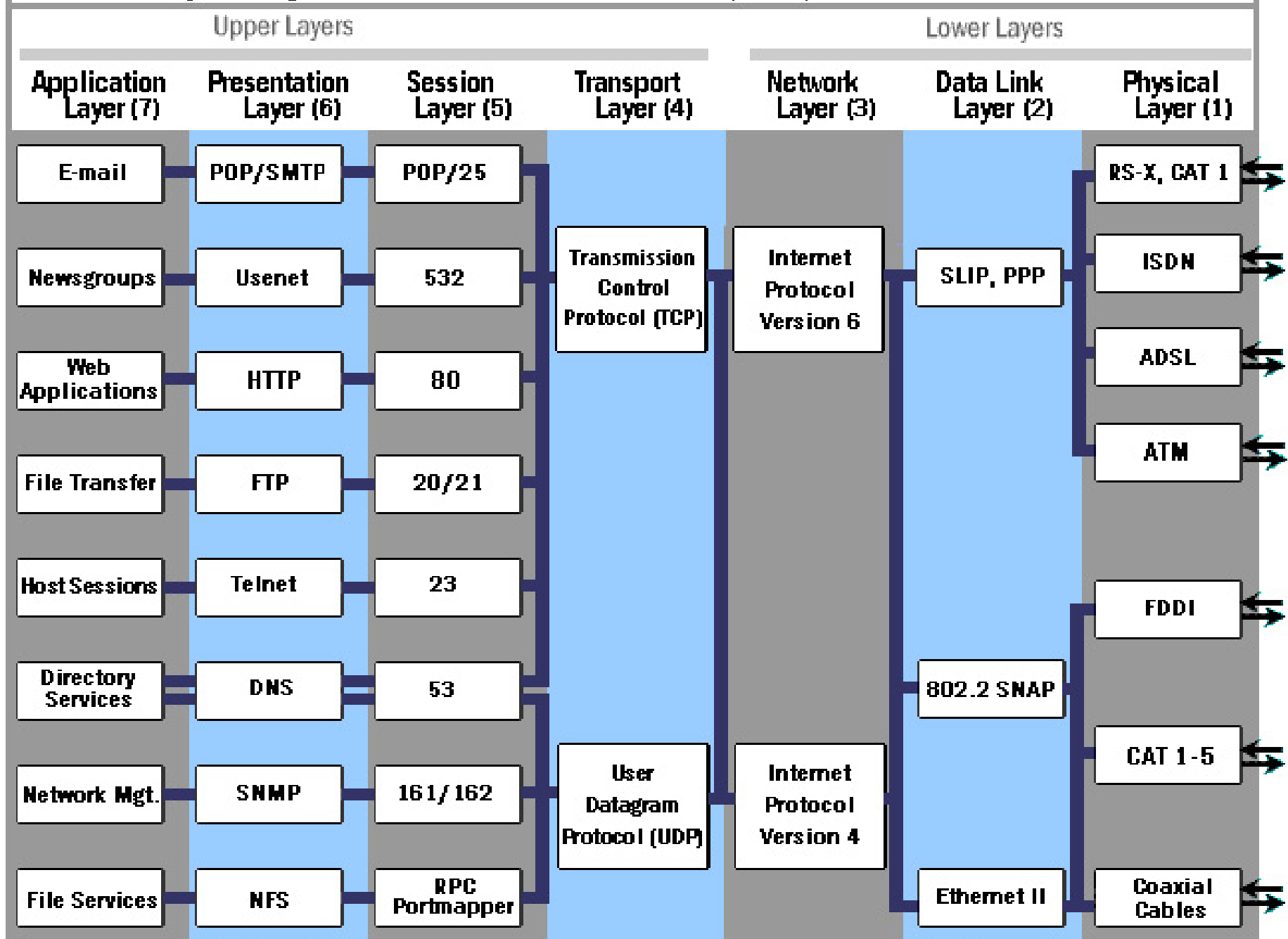


Summary



- Using TCP/IP and lower layers, we can get multi-packet messages delivered reliably from address space **A** on machine **B** to address space **C** on machine **D**, where machines **B** and **D** are many heterogeneous network hops apart, without knowing any of the underlying details
- Higher protocol layers facilitate specific services
 - email: smtp
 - web: http
 - file transfer: ftp
 - remote login: telnet

Open Systems Interconnection (OSI) Reference Model



Client/Server communication

- The prevalent model for structuring distributed computation is the client/server paradigm
 - A **server** is a program (or collection of programs) that provides a service to other programs
 - | e.g., file server, name server, web server, mail server ...
 - | server/service may span multiple machines
 - often, machines are called servers too
 - E.g., the web server runs on a Dell server computer
 - A **client** is a program that uses the service
 - | the client first **binds** to the server
 - locates it, establishes a network connection to it
 - | the client then sends **requests** (with data) to perform **actions**, and the server sends **responses** (with data)
 - e.g., web browser sends a "GET" request, server responds with a web page
- TCP/IP is the transport, but what is the higher-level programming model?

Messages



- Initially, people hand-coded messages to send requests and responses
 - Message is a stream of bytes - "op codes" and operands
- Lots of drawbacks
 - Need to worry about message format
 - Have to pack and unpack data from messages
 - Servers have to decode messages and dispatch to handlers
 - Messages are often asynchronous
 - After sending one, what do you do until response comes back?
 - Messages aren't a natural programming model

Procedure calls



- Procedure calls are a natural way to structure multiple modules inside a single program
 - every language supports procedure calls
 - semantics are well-defined and well-understood
 - programmers are used to them
- “Server” (called procedure) exports an API
- “Client” (calling procedure) calls the server procedure’s API
- Linker binds the two together

Procedure call example

```
Client Program:  
...  
sum = server->Add(3,4);  
...
```

```
Server API:  
int Add(int x, int y;
```

```
Server Program:  
int Add(int x, int y) {  
    return x + y;  
}
```

- If the server were just a library, then "Add" would just be a local procedure call

Remote Procedure Call (RPC)

- Traditional procedure call syntax and semantics across a network
- The most common means used for remote communication in *client/server systems*
- Used both by operating systems and applications
 - NFS is implemented as a set of RPCs
 - HTTP is essentially RPC
 - DCOM, CORBA, Java RMI, etc., are just RPC systems

RPC



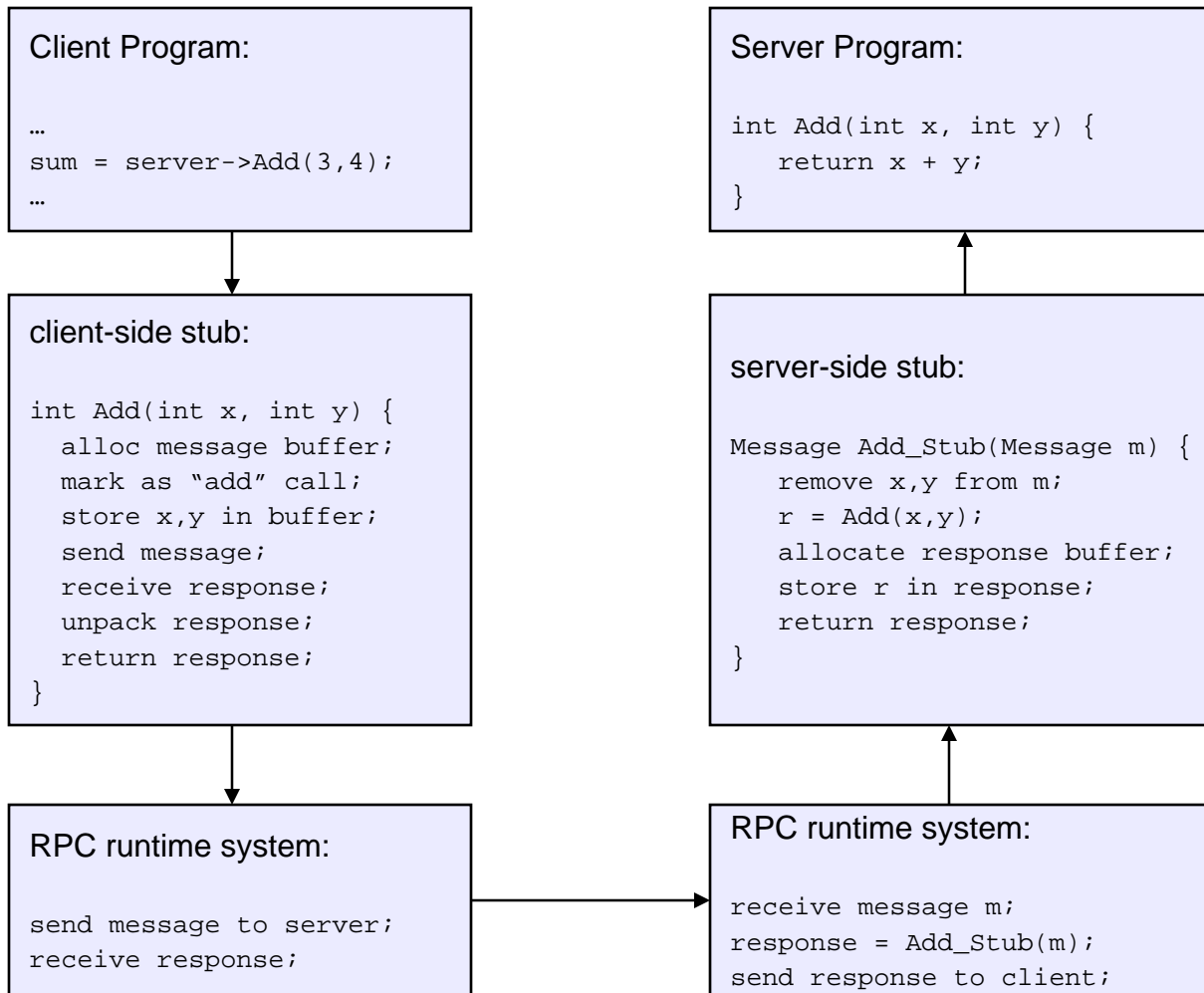
- Use procedure calls as the model for distributed (remote) communication
 - Have servers export a set of procedures that can be called by client programs
 - | similar to library API, class definitions, etc.
 - Clients do a local procedure call, as though they were directly linked with the server
 - | under the covers, the procedure call is converted into a message exchange with the server
 - | *largely invisible to the programmer!*

RPC issues



- There are a bunch of hard issues:
 - How do we make the “remote” part of RPC invisible to the programmer?
 - | and is that a good idea?
 - What are the semantics of parameter passing?
 - | what if we try to pass by reference?
 - How do we bind (locate/connect-to) servers?
 - How do we handle heterogeneity?
 - | OS, language, architecture, ...
 - How do we make it go fast?

RPC example invocation



Topics:

- interface description
- stubs
- stub generation
- parameter marshalling
- binding
- runtime system
- error handling
- performance
- thread pools

RPC model



- A server defines the service interface using an **interface definition language (IDL)**
 - The IDL specifies the names, parameters, and types for all client-callable server procedures
 - example: ASN.1 in the OSI reference model
 - example: Sun's XDR (external data representation)
- A "**stub compiler**" reads the IDL declarations and produces two stub procedures for each server procedure
 - The server programmer implements the service's procedures and links them with the **server-side stubs**
 - The client programmer implements the client program and links it with the **client-side stubs**
 - The stubs manage all of the details of remote communication between client and server using the **RPC runtime system**

RPC stubs



- A client-side stub is a procedure that looks to the client as if it were a callable server procedure
 - It has the same API as the server's implementation of the procedure
 - A client-side stub is just called a "stub" in Java RMI
- A server-side stub looks like a caller to the server
 - It looks like a hunk of code that invokes the server procedure
 - A server-side stub is called a "skeleton" or "skel" in Java RMI
- The client program thinks it's invoking the server
 - But it's calling into the client-side stub
- The server program thinks it's called by the client
 - But it's really called by the server-side stub
- The stubs send messages to each other, via the runtime, to make the RPC happen transparently

RPC marshalling



- Marshalling is the packing of procedure parameters into a message packet
 - The RPC stubs call type-specific procedure to marshal or unmarshal the parameters of an RPC
 - the client stub marshals the parameters into a message
 - the server stub unmarshals the parameters and uses them to invoke the service's procedure
 - On return:
 - the server stub marshals the return value
 - the client stub unmarshals the return value, and returns them to the client program

RPC binding



- Binding is the process of connecting the client to the server
 - The server, when it starts up, exports its interface
 - | identifies itself to a network name server
 - | tells RPC runtime that it is alive and ready to accept calls
 - The client, before issuing any calls, imports the server
 - | RPC runtime uses the name server to find the location of the server and establish a connection
- The import and export operations are explicit in the server and client programs
 - A slight breakdown in transparency
 - | more to come...

RPC transparency

- One goal of RPC is to be as transparent as possible
 - Make remote procedure calls look like local procedure calls
 - We've seen that binding breaks this transparency
- What else breaks transparency?
 - Failures: remote nodes/networks can fail in more ways than with local procedure calls
 - network partition, server crash
 - need extra support to handle failures
 - server can fail independently from client
 - "partial failure": a big issue in distributed systems
 - if an RPC fails, was it invoked on the server?
 - Performance: remote communication is inherently slower than local communication
 - if you're not aware you're doing a remote procedure call, your program might slow down an awful lot...

RPC and thread pools



- What happens if two client threads (or client programs) simultaneously invoke the same server procedure using RPC?
 - Ideally, two separate threads will run on the server
 - So, the RPC run-time system on the server needs to spawn or dispatch threads into server-side stubs when messages arrive
 - is there a limit on the number of threads?
 - if so, does this change semantics?
 - if not, what if 1,000,000 clients simultaneously RPC into the same server?

RPC in the web world



- REST, SOAP, and XML-RPC are different religious denominations of RPC using XML
 - Modern web services (e.g., flickr, AWS's S3) offer APIs based on all of these
- XML is nothing but self-describing data
 - Great for parameter lists, if verbosity is not an issue, which it isn't these days
- Protocol Buffers: Google's version of this
- Thrift: Facebook's version of this
 - Both PBs and Thrift use an IDL that compiles into stubs for a wide variety of languages

Flickr Services

[API Documentation](#) | [Feeds](#) | [Your API Keys](#) | [Apply for a new API Key](#)

The Flickr API is available for non-commercial use by outside developers. Commercial use is possible by prior arrangement.

Read these first:

- [Overview](#)
- [Encoding](#)
- [User Authentication](#)

- [Dates](#)
- [Tags](#)
- [URLs](#)
- [Buddyicons](#)

- [Flickr APIs Terms of Use](#)

- [API Keys](#)
- [Developers' mailing list](#)

Photo Upload API

- [Uploading Photos](#)
- [Replacing Photos](#)

API Methods

activity

- [flickr.activity.userComments](#)
- [flickr.activity.userPhotos](#)

auth

- [flickr.auth.checkToken](#)
- [flickr.auth.getFrob](#)
- [flickr.auth.getFullToken](#)
- [flickr.auth.getToken](#)

blogs

- [flickr.blogs.getList](#)
- [flickr.blogs.postPhoto](#)

contacts

- [flickr.contacts.getList](#)
- [flickr.contacts.getPublicList](#)

Flickr Services

[API Documentation](#) | [Feeds](#) | [Your API Keys](#) | [Apply for a new API Key](#)

Uploading Photos

This is the specification for building photo uploader applications.

It works outside the normal Flickr API framework because it involves sending binary files over the wire.

Uploading apps can call the [flickr.people.getUploadStatus](#) method in the regular API to obtain file and bandwidth limits for the user.

Uploading

Photos should be POSTed to the following URL:

```
http://api.flickr.com/services/upload/
```

Authentication

This method requires authentication with 'write' permission.

For details of how to obtain authentication tokens and how to sign calls, see the [authentication api spec](#). Note that the 'photo' parameter **should not** be included in the signature. All other POST parameters should be included when generating the signature.

Arguments

photo

The file to upload.

title (optional)

The title of the photo.

description (optional)

A description of the photo. May contain some limited HTML.

tags (optional)

A space-separated list of tags to apply to the photo.

is_public, is_friend, is_family (optional)

Set to 0 for no, 1 for yes. Specifies who can view the photo.

safety_level (optional)

Set to 1 for Safe, 2 for Moderate, or 3 for Restricted.

content_type (optional)

Set to 1 for Photo, 2 for Screenshot, or 3 for Other.

hidden (optional)

Set to 1 to keep the photo in global search results, 2 to hide from public searches.

Example Response

When an upload is successful, the following xml is returned:

```
<photoid>1234</photoid>
```

photoid is the id of the new photo. This response is formatted in the [REST API response](#) style.

Error Codes

If the upload fails, a [REST API error response](#) is returned. The following error codes are possible:

2: No photo specified

The photo required argument was missing.

3: General upload failure

The file was not correctly uploaded.

4: Filesize was zero

The file was zero bytes in length.

5: Filetype was not recognised

The file was not of a recognised image format.

6: User exceeded upload limit

The calling user has reached their monthly bandwidth limit.

96: Invalid signature

The passed signature was invalid.

97: Missing signature

The call required signing but no signature was sent.

98: Login failed / Invalid auth token

The login details or auth token passed were invalid.

99: User not logged in / Insufficient permissions

The method requires user authentication but the user was not logged in, or the authenticated method call did not have the required permissions.

100: Invalid API Key

The API key passed was not valid or has expired.

105: Service currently unavailable

The requested service is temporarily unavailable.