Operating System Support

Prof. Dave Bakken

Cpt. S 464/564 Lecture

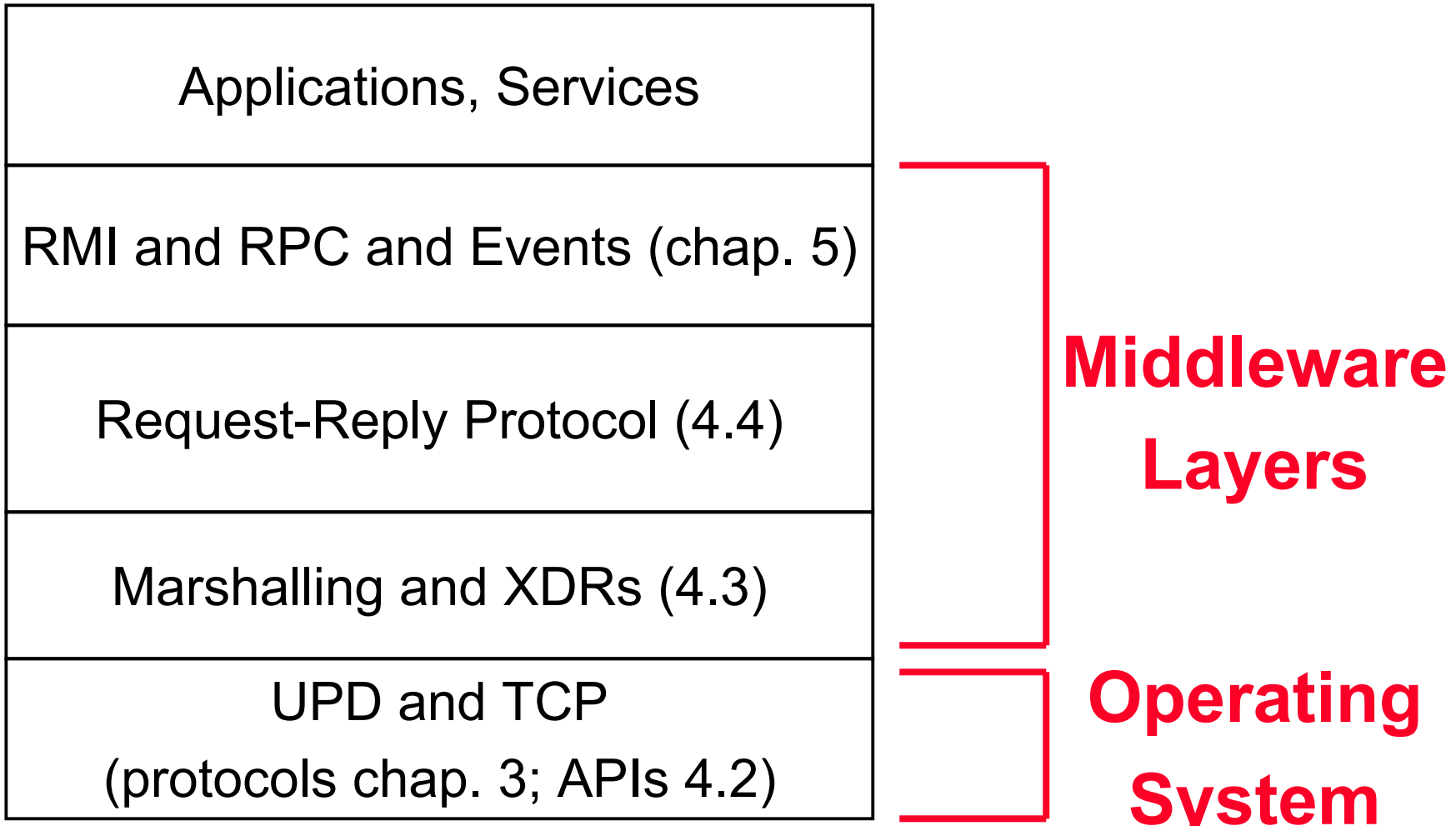
Textbook, Chapter 5

Oct 16+18, 2000

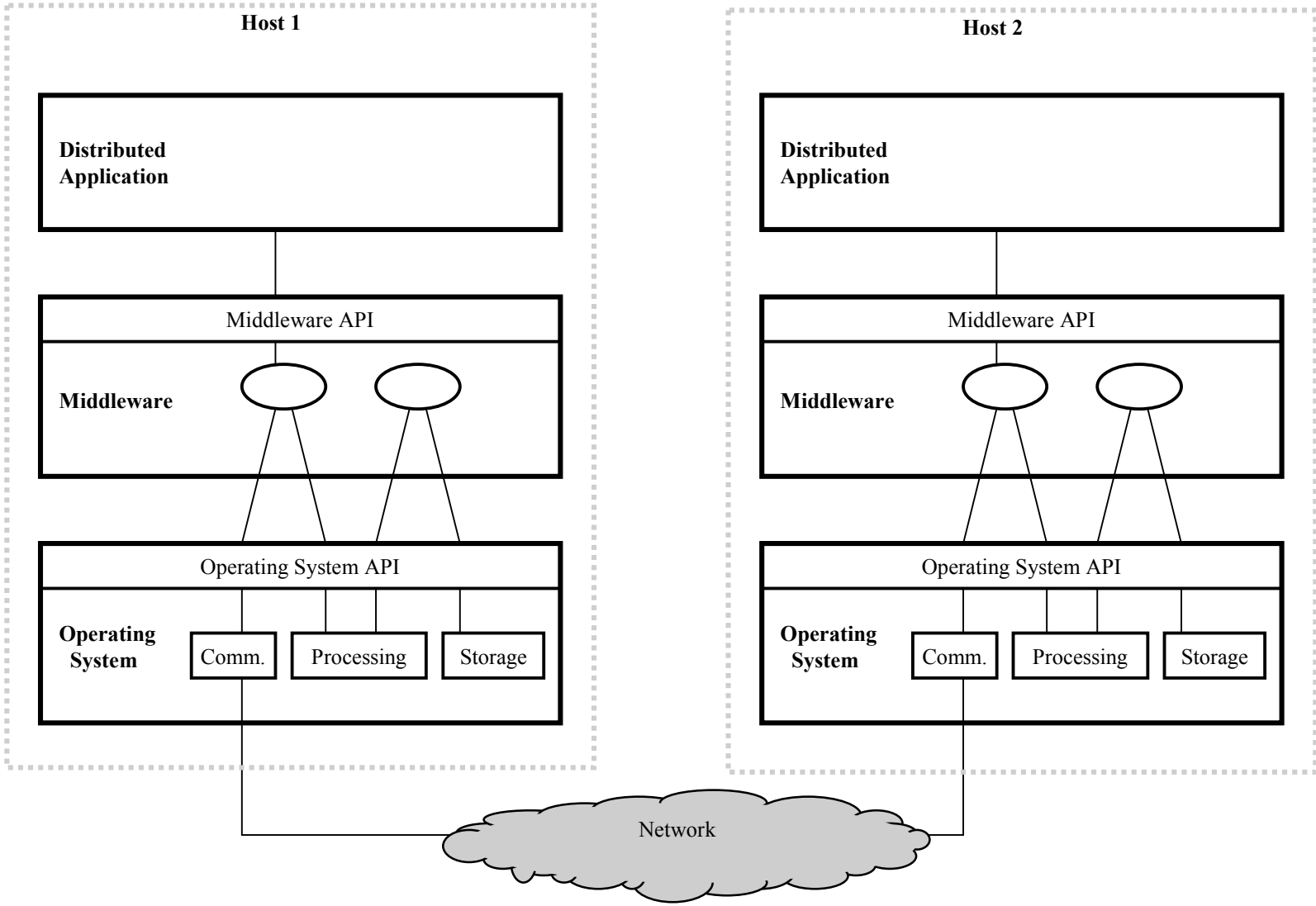
Administrative Items

- Office hours (OH) Monday 10/16 will be 95% cancelled, due to family sickness
  - See me right after class if you need to!!
  - Make an appointment if you need to see me before Wednesday 10-12am OH
- New book in ETRL 301 very soon: *Advanced CORBA Programming with C++*
  - By far book with most C++ CORBA examples
  - A few Orbix-specific features, but generally well documented
- Reminder: Midterm exam is Wednesday October 25th
  - Half or so of the lecture 10/18 will be a review for this….
- Note on Chapter 6 in text: do not worry about the specific features of all the research systems, or other topics from this chapter not in these slides. But do learn the different kinds of OS support in these slides
  - If we don't finish these slides Wednesday 10/18 then only that part of Chapter 6 that I lecture on is testable, unless I state otherwise.
- Project #3 notes and questions….

Middleware Layers and IPC



Middleware and Operating Systems



Outline

- **The operating system layer**
- Protection
- Processes and threads
- Communication and invocation
- Operating systems architecture

Middleware Builds on an Operating System

- Issue: how well can the requirements of middleware be met by the OS?
- Requirements
  - Efficient and robust access to physical resources
  - Flexibility to implement a variety of resource management policies
- Classical OS definition:

- OS provides (somewhat) higher abstractions than the lower-level resources (communications, processor, storage (memory, disk))
  - File abstraction, not disk block abstraction and disk device drivers API
  - ....
  - Provided via system call API
- Similarly, middleware makes the distributed system (much more) programmable.

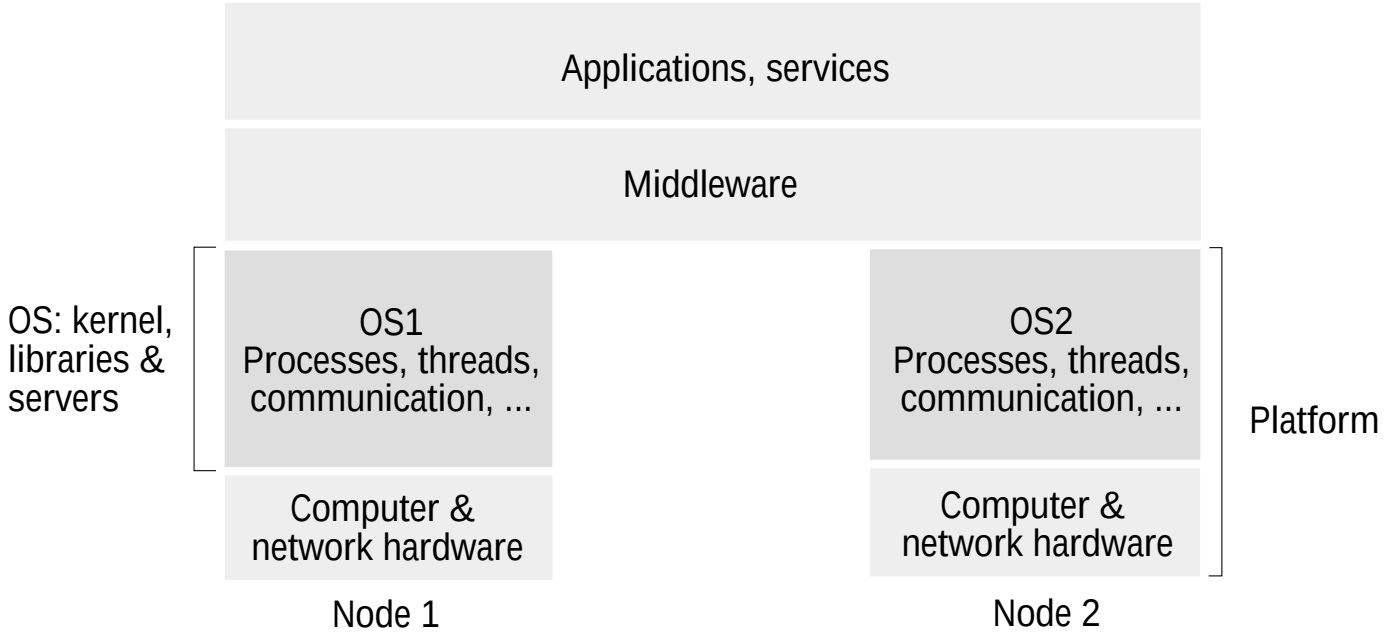
Network Operating Systems

- Network Operating System (NOS): an OS which has networking capability built into it
  - Most do today, but a few specialty ones do not.
  - Examples that do:
- Access here can sometimes be provided in a network transparent way
  - NFS
- NOSs lets hosts retain autonomy in managing their resources
  - Multiple system images, one per node
  - System programing uses remote APIs
  - Users use rlogin or telnet or ssh
- Alternative: single system image
  - Users never know/care where programs run or where resources are located
  - OS has control over all resources on all hosts
  - OS transparently locates a process where it chooses
  - Called a “distributed operating system”: Mach, Amoeba are major ones

Middleware and NOSs

- Distributed Operating Systems
  - Are not in general use
  - Are barely researched any more
- Reasons
  - Application software base (emulations didn't perform well enough)
  - Users want autonomy for their computers (mainly for performance)
- NOSs provide a workable balance between
  - Autonomy
  - Providing network-transparent resource access to clients and users
- Users need system software (middleware and OS) to have good performance

Figure 6.1 System layers



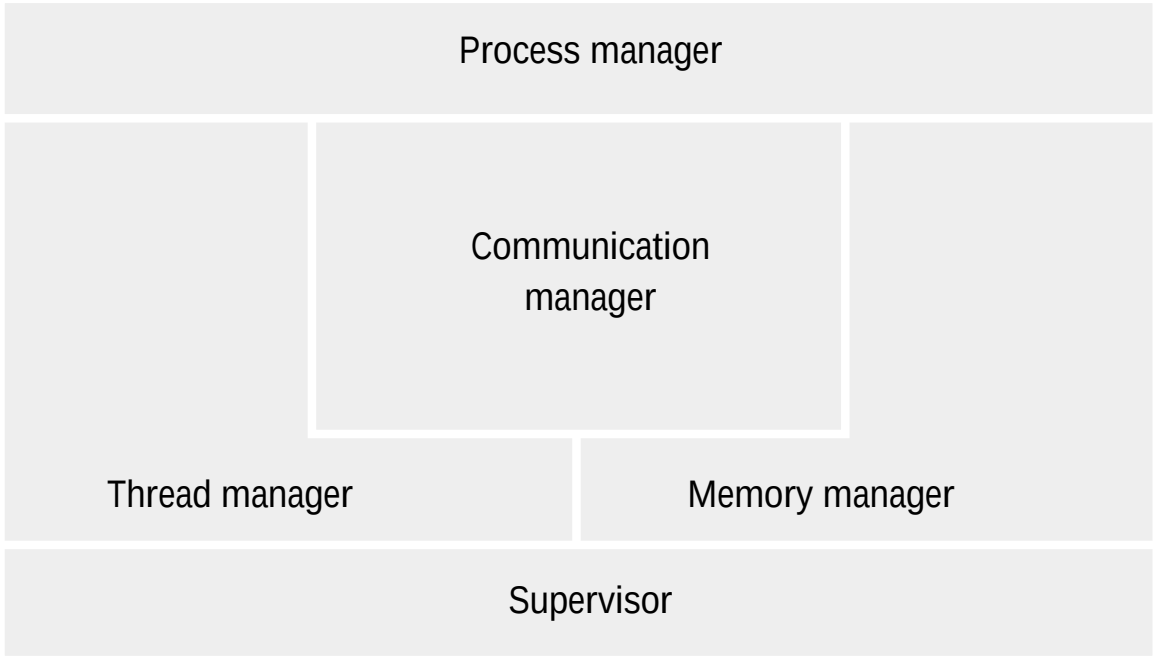
Operating System Layer

- Goal: understand the impact of particular OS mechanisms on middleware's ability to implement their programming abstractions reasonably
  - Including delivering resource sharing to users, deep down
- Main architectural components of interest
  - Kernels
  - Server processes executing on an OS
  - Client processes executing inside the kernel (via a system call)
- System call API must provide
  - Encapsulation: useful interface, hiding details (which ones?)
  - Protection: protect resources from illegitimate access (which ones?)
  - Concurrent processing: let clients share resources concurrently (why?)
    - Resource managers implement this, providing concurrency transparency

Operating System Layer (cont.)

- Clients access resources (indirectly) via
  - Invocations to a server object
  - System call to kernel
  - We call either an invocation mechanism
  - Can be implemented in library, kernel, server
- OS tasks related to an invocation mechanism
  - Communication
  - Scheduling
- Core OS Components
  - Process manager: handle creation of processes and invocations to them
  - Thread manager: create, synchronize, and schedule threads
  - Communications manager: communicate between threads in a different process on same computer (sometimes on different computers)
  - Memory manager: manage physical and virtual memory
  - Supervisor:
    - Dispatch interrupts, system call traps, other exceptions
    - Control MMU and hardware caches
    - Example: known as Hardware Abstraction Layer (HAL) in Windows NT

Figure 6.2 Core OS functionality



Outline

- The operating system layer
- **Protection**
- Processes and threads
- Communication and invocation
- Operating systems architecture

Protection

- Threats to a system
  - Malicious code (bad guys+gals)
  - Buggy code (imperfect guys+gals)
- Counter threat #1: type-safe programming language
  - Invoke a module only through a capability
    - Unforgeable safe reference
    - Given by system
  - Cannot change module variables, only invoke via its interface
  - Examples: Java, Modula-3 (C++?)
- Counter threat #2: hardware support
  - Requires a trusted component: a kernel

Kernels and Protection

- Kernel: program that
  - Always runs
  - Always has complete access privileges (MMU, processor registers, …)
- An address space is a domain of protection for memory
- CPU support: hardware register telling if privileged instructions are permitted to be executed
- User-level process: one executing in user mode and with a (restricted) user address space
- System-level process: one executing in privileged (supervisor) mode and with a system-level address space
  - Can manipulate system data structures
- User-level process calls kernel's address space via protected and well-defined invocation
  - Usually implemented by a machine TRAP instructino

Outline

- The operating system layer
- Protection
- **Processes and threads**
  - **Address spaces**
  - **Creation of new processes**
  - **Threads (LOTS OF MATERIAL)**
    - **Advantages of having multiple threads**
    - **Programming with threads (Java example)**
    - **Alternate design choices for implementing threads**
- Communication and invocation
- Operating systems architecture

Processes and Threads

- Process: execution environment plus one or more threads
- Execution environment: unit of resource management (of local kernel-managed resources):
  - Address space
  - Thread synchronization resources (e.g., semaphores)
  - Thread communication interfaces (e.g., sockets)
  - Higher-level abstractions for resources (open files, windows)
- Thread: OS abstraction of an activity
  - Q: why is it called an abstraction? What is abstracted?
- Goal of threading: increase concurrent execution
  - Overlap of I/O and computation
  - Utilize multiple processors
- Multi-threaded process: one that can support threads
- Heavyweight process: another name for a process
- Lightweight process: another name for a thread

Address Spaces

- A unit of management of a processor's virtual memory (VM)
- Divided into different areas (separated by inaccessable VM):
  - Text (program code): at bottom, does not change
  - Heap (global or static data): above text, grows towards top
  - Stack (local data): in middle, grows towards heap
  - Auxilary regions: at top, above stack, can have infinite # of them
- Auxiliary regions
  - Can be used to put threads' stack in
    - Supports multiple threads
    - Can check bounds and growth rate of a thread's stack
  - Can be used to share memory
- Shared memory region: memory in different address spaces mapping to the same physical memory
  - Libraries: save a lot of memory by dynamically loading only what is needed
  - Kernel: system call data structures
  - Optimized IPC for threads/processes on same computer

Clusters

- Cluster: a set of off-the-shelf computers
  - Connected by a high-speed interconnection
  - Running commodity OS
  - Often dedicated to a single service or activity, not user logins or running multiple applications
- Example uses
  - Support thin-client or X-terminal systems
  - Provide fault tolerance via replication
  - Run parallel programs
    - Note: Network latencies are less than disk latencies!
- Clusters are a very important class of distributed computer!
- We focus on more general distributed systems in this class
  - Except this slide 

Creation of New Processes

- Process creation is provided by OS
  - E.g., Unix fork() and exec()
- Distributed system process infrastructure subdivided into
  - Choice of target host
  - Creation of an execution environment on that host
- Choice of process host: policy, subdivided
  - Transfer policy: decision to create locally or remotely
  - Location policy: decision on what node for a remote decision
  - Q: how and why might a location policy choose a particular host for process creation?

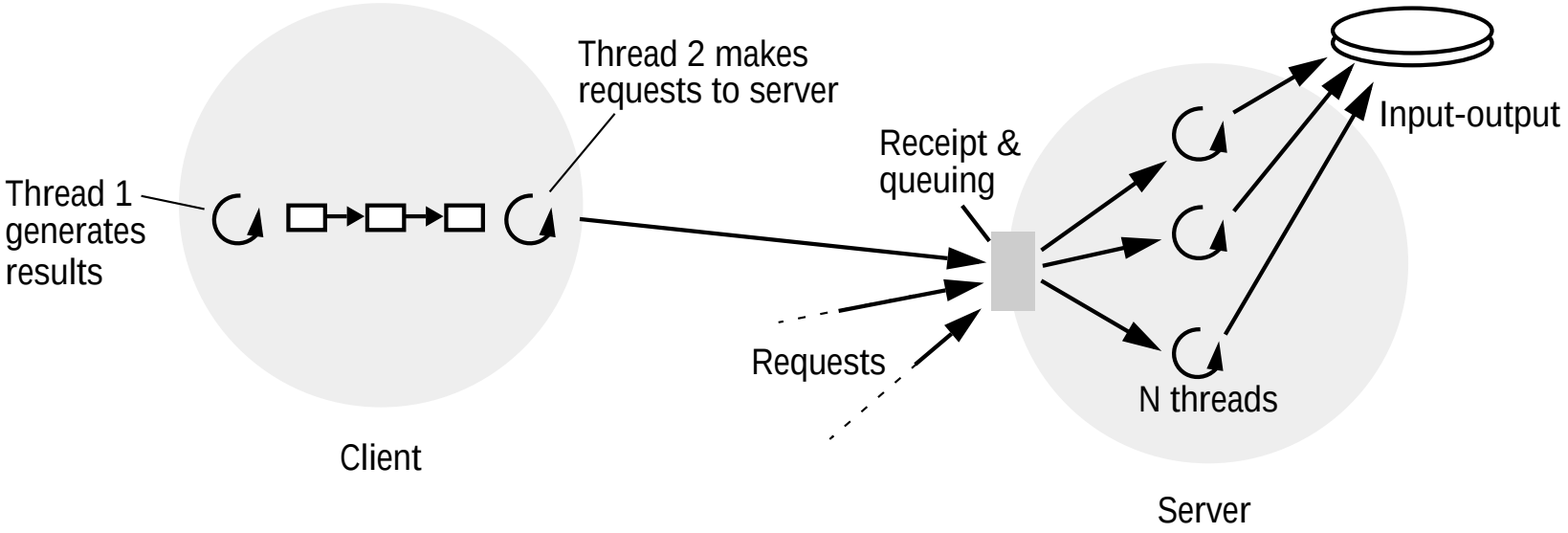
Process Location Policies and Organization

- Static process location policy: one that is determined (carefully) at design time
  - Deterministic: Host X always locates process on Host Y
  - Probabilistic: Host X locations on Hosts A-E at random
- Adaptive process location policy: one that considers runtime conditions in its decision
  - Q: what conditions might influence decision?
- Management structure of load-sharing systems
  - Centralized: only one load manager component in the system
  - Hierarchical: Managers decide some of the tree, and transfer to a common ancestor for other parts
  - Decentralized: hosts implement a distributed algorithm
- Ways to initiate a process creation
  - Sender-initiated load sharing algorithm: one where the host which needs a new process to be created initiates the creation process (e.g., its load threshold)
  - Receiver-initiated load sharing algorithm: A node whose load gets too low advertises available CPU cycles
    - Needs ability to transfer an existing process to it, or subdivide work of an existing one

Creation of a New Execution Environment

- Approach #1: create it using one of a number of preconfigured and initialized address spaces
- Approach #2: created it based on an existing execution environment
  - Unix fork()
- (Midterm Exam study time optimization: don't worry about Mach and Chorus examples here, or Figure 6.4)

Figure 6.5 Client and server with threads



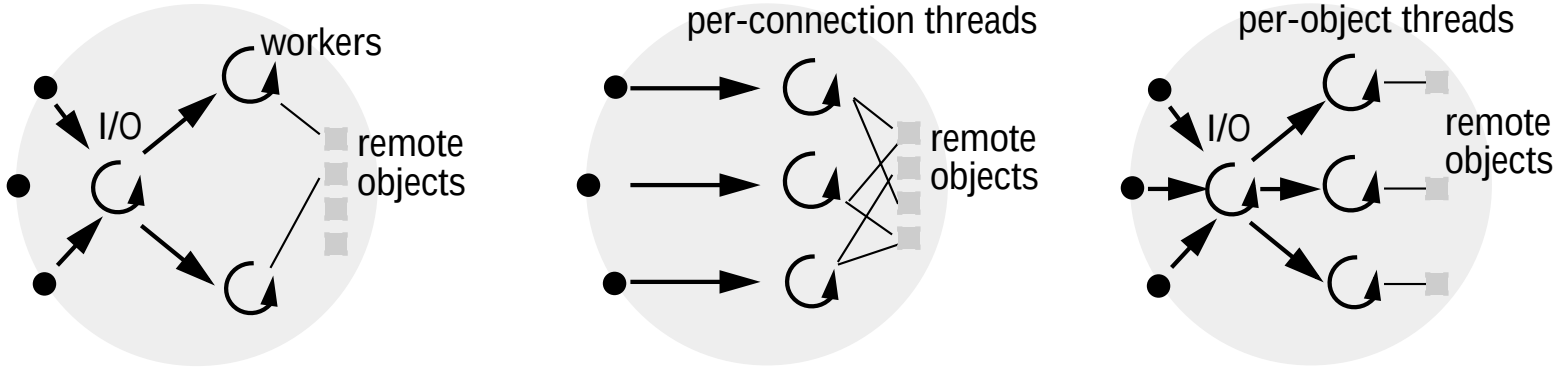
Advantages of Threads

- Example from Figure 6.5
  - Each request takes 2 milliseconds of CPU at server
  - Each request takes 8 milliseconds of I/O delay for disk
- With single processor computer: 10 milliseconds/request, so max throughput 100 requests/second
- New case
  - 2 threads in pool, can overlap computation with I/O
  - But serialized I/O
  - Best case: all processing overlapped with I/O, but still 1000/8 == 125 requests per second
- Another case: assume I/O not serialized, and no performance hit for concurrency
  - With infinite #threads in pool, can get 500 requests per second
- Better case: disk caching
  - Can radically cut I/O time and increase server throughput (if locality of reference is common)

Threading Architectures

- Figure 6.5 was worker pool architecture
  - Extension: add multiple queues and priorities (HP WebQoS; Bhatti and Friedrich 1999)
  - Disadvantage: not too flexible (too few or many threads)
  - Disadvantage: Switching costs between I/O and worker threads to manipulate shared queue
- Thread-per-request architecture
  - One worker thread for each incoming request (destroyed when done)
  - Advantage: threads don't contend for shared queue
  - Advantage: Never have too few threads for the given requests
  - Disadvantage: overhead of thread creation and destruction
- Thread-per-connection architecture: one thread associated with each client-server connection
- Thread-per-object architecture: one thread for each remote object
- Similarities between thread-per-connection and thread-per-object
  - Advantages: lower thread-management overheads
  - Disadvantages: clients may be delayed if not enough threads

Figure 6.6 Alternative server threading architectures (see also Figure 6.5)



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

Threading and Clients

- Example in Figure 6.5
- Can help a given client to overlap computation with communication
- Q: common example(s) in distributed systems?

Treads vs. Multiple Processes

- Why not just use multiple processes?
- Reasons
  - Creating a thread is cheaper than creating a new process
  - Switching between threads is cheaper than switching between processes
  - Sharing data and other resources is also cheaper
- But threads are not protected from one another
  - Fundamental tradeoff of safety versus performance

Figure 6.7 State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Programming Threads

- Some languages support threads natively
  - Synchronizing Resources (SR)
  - Java
  - Ada95
  - Modula-3
- Other languages require use of an external library
  - POSIX pthreads very common

Java Threads Overview

- *Thread* class provides constructor and management methods in Fig 6.8
- Thread lifetimes
  - Start in SUSPENDED state
  - After start() called it is RUNNABLE
  - Some object calls its run() method (can call self or other)
- Threads can be collected into groups
  - Assigned at creation time
  - Uses: security and common priorities
- Java *synchronized* implements a monitor (mutex region) over entire methods or just code blocks
- Java threads can be blocked waiting for a callback from an arbitrary object (part of top Java class, *Object*)
  - Much like condition variables in concurrent programming terms
  - Blocked thread calls wait() Waker calls notify() to wake up just one thread or notifyAll() to wake all
- (Note: skipping rest of Section 6.4 after Java discussion; not testable.)

Figure 6.8 Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Set and return the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
Enter the *READY* state and invoke the scheduler.

destroy()
Destroy the thread.

Figure 6.9 Java thread synchronization calls

thread.join(int millisecs)

Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()

Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), object.notifyAll()

Wakes, respectively, one or all of any threads that have called *wait()* on *object*.

Outline

- The operating system layer
- Protection
- Processes and threads
- **Communication and invocation**
- Operating systems architecture

Invocation Performance

- (Note: skipping first part of Section 6.5, until 6.5.1)
- Clients require reasonable performance
- Achievable on LANS usually
  - Software overheads dominate here, not communication costs
- Internet realities (compared to LANs)
  - Network latencies high and with high variance
  - Bandwidth often low
- Kinds of invocations (all involve communication both ways)
  - Calling local procedure
  - Invoking method of local object
  - Making a system call
  - Calling a remote procedure
  - Invoking a method of a remote object
- Performance-related distinctions between above kinds of invocations
  - Synchronous or asynchronous
  - Domain transition involved (cross an address space)
  - Involve communication over a netowrk
  - Involve thread scheduling and switching

Figure 6.11 Invocations between address spaces (cont'd on next slide)

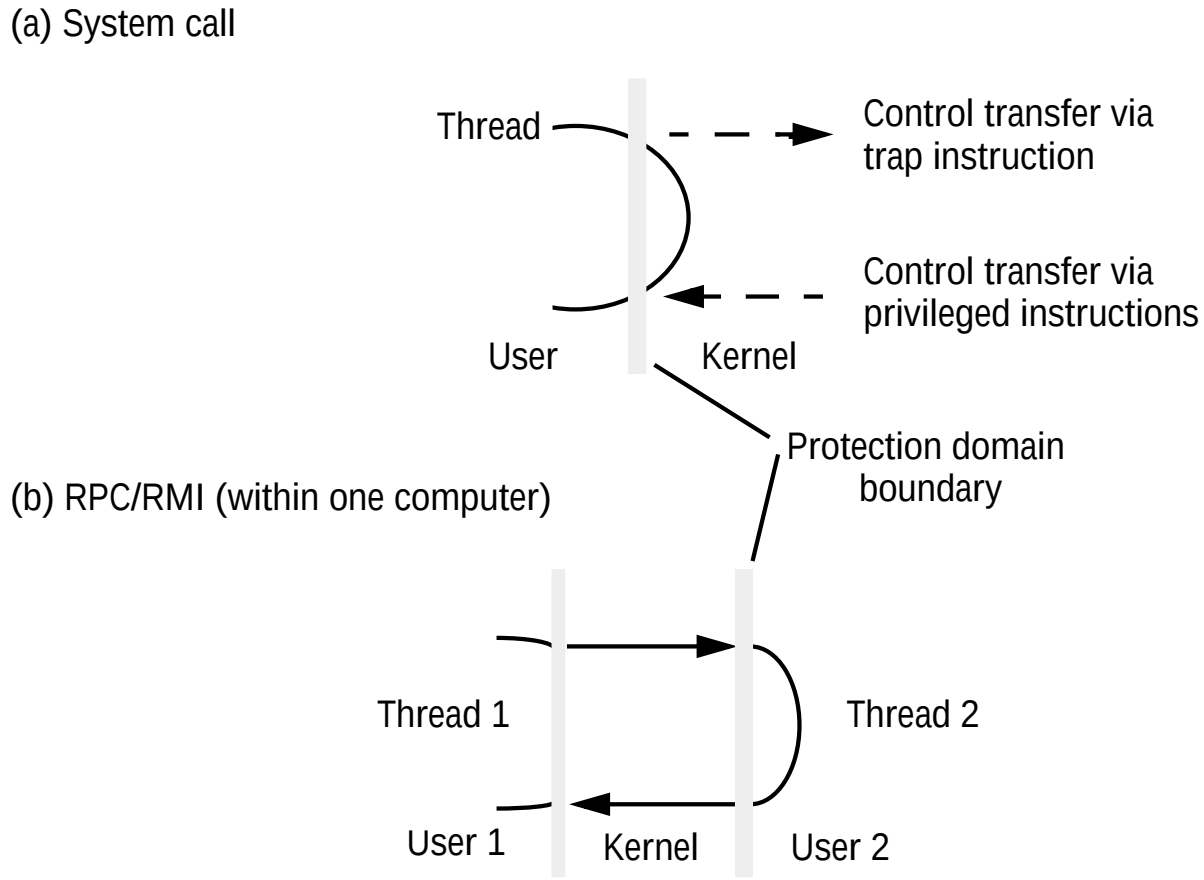
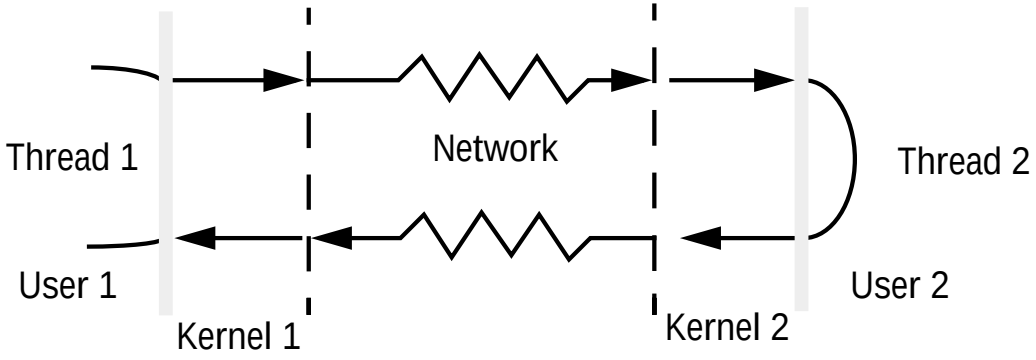


Figure 6.11 (cont'd) Invocations between address spaces

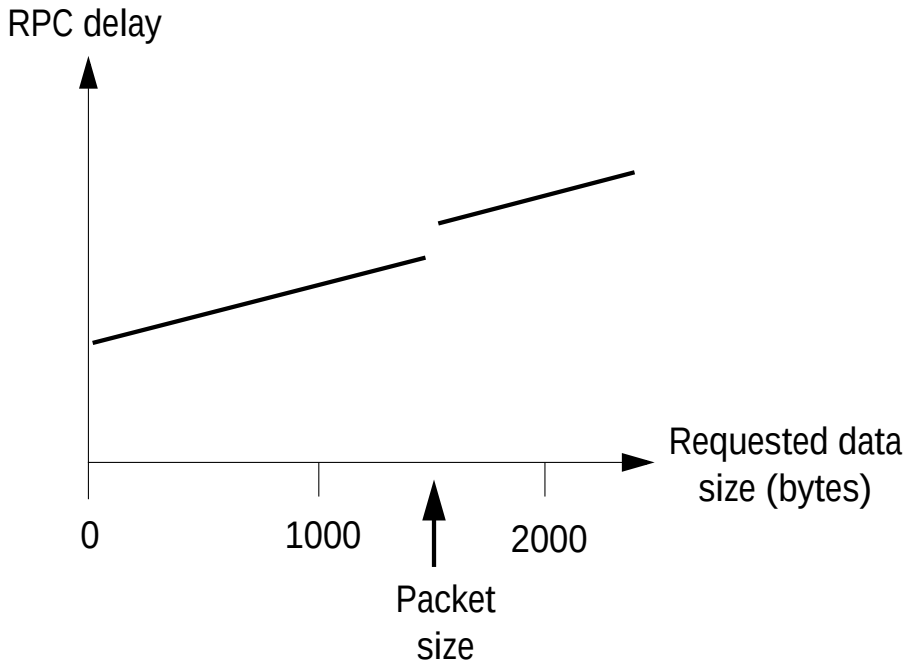
(c) RPC/RMI (between computers)



Network Invocation Costs

- Null RPC: one sent without any parameters or return values
  - Useful for measurements: measures fixed, unavoidable network latencies
- Current "ballpark" times (microseconds)
  - Null conventional procedure call: <1
  - Null RPC across fast LAN: 100s (tenths of milliseconds)
  - Null RPC across WAN: 30,000 – 1,000,000
- Q: what common program lets you execute a Null RPC or close to experiment?
- Invocation costs (observed delay) go up with arguments
  - More a function of the network bandwidth than just latency

Figure 6.12 RPC delay against parameter size



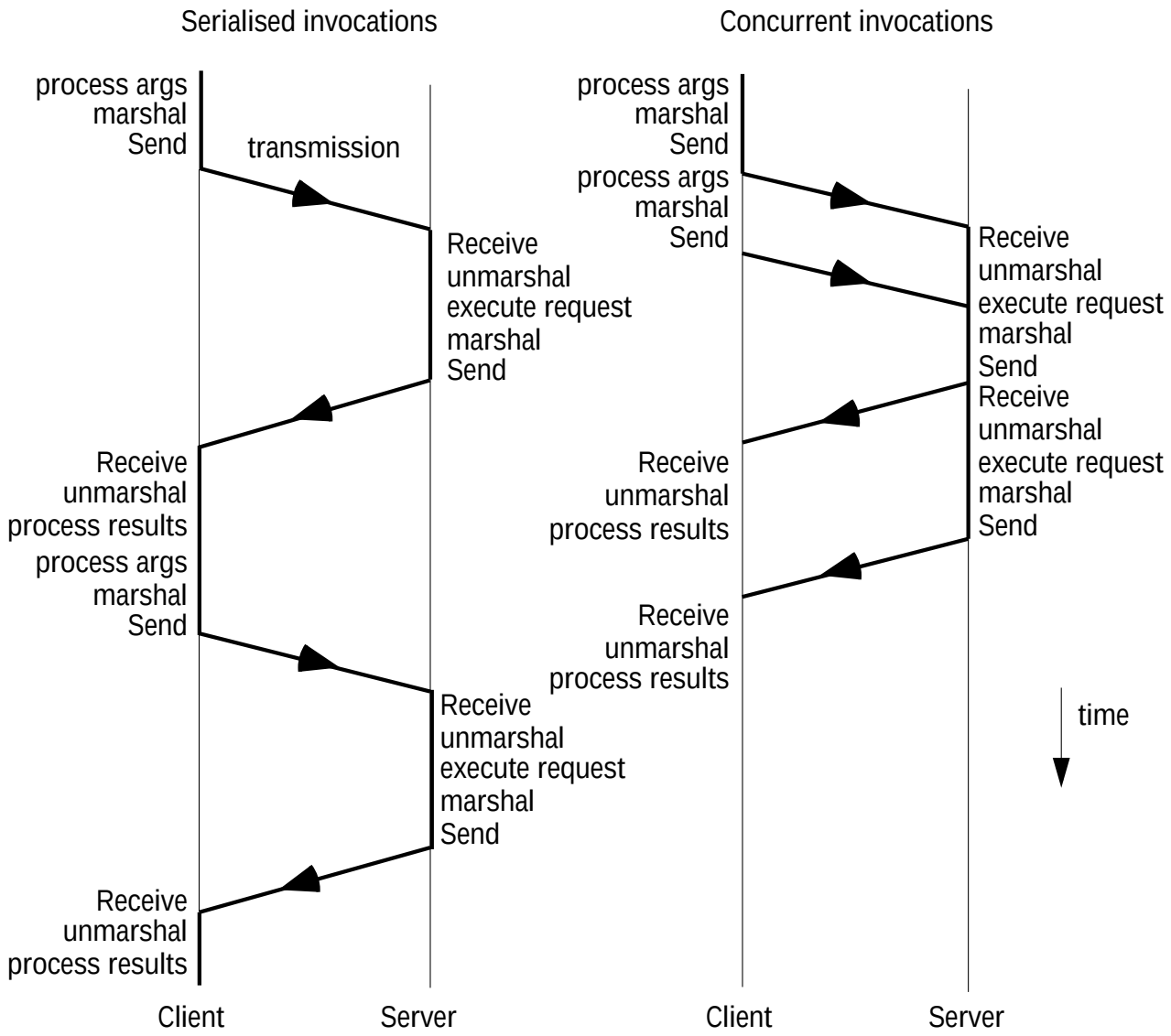
Throughput of RPC

- Throughput of an RPC can be important measure, too
  - Very important when bulk data has to be transferred
  - Some costs are network latencies
  - Other costs are a function of the system software
- Invocation delay components (other than network transmission times for application data)
  - Marshalling
  - Data copying
    - Across user-kernel boundary
    - Across each protocol layer (unless sophisticated support a la x-kernel)
    - Between network interface and kernel buffers
  - Packet initialization
    - Note headers can include a checksum, proportional to size of arguments
  - Thread scheduling and context switching
    - System calls (e.g., kernel communications API)
    - Server thread(s)
    - Possible separate network manger process in the OS
  - Waiting for ACKs

Asynchronous Operation

- (Note: skipped in 6.5.1 from “Memory sharing” to end of 6.5.1, notably lightweight RPC)
- Recall internet realities
  - High latencies
  - Low bandwidths
  - High server loads
  - Network disconnection and reconnection (failures or mobility)
- Asynchronous operation helps overcome high latencies; two cases
  - Concurrent invocations
  - Asynchronous structuring of an application (natural for some)
- Q: everyday example of concurrent invocations?

Figure 6.14 Times for serialised and concurrent invocations



Language Support for Asynch. Operation

- Asynch invocation returns a promise: an "IOU" for a reply
  - Called future in Cronus
- Program later checks if invocation done using promise
  - Blocking
  - Non-blocking (polling)
  - Either returns the reply arguments (return value and **out** and **inout** args)
- Note Mike Dean, Cronus distributed object system and distributed application developer for BBN since ~1984, lecturing Monday Oct 30th…

Outline

- The operating system layer
- Protection
- Processes and threads
- Communication and invocation
- **Operating systems architecture (just microkernels vs. monolithic ones)**

Monolithic vs. Microkernels

- (NOTE: due to the upcoming test, this slide and next are the only things you are responsible for knowing in Section 6.6, Operating System Architecture)
- Kernels can be designed two ways: monolithic and microkernel
- Monolithic
  - Massive: one big piece
  - Undifferentiated: non-modular coding (lots of "cheating" between layers, optimizing, etc.)
  - Intractable: really hard to change
- Microkernel
  - Provide only most basic OS abstractions: address spaces, threads, local communication
  - All other services ("subsystems") are in OS servers above microkernel, dynamically loaded
  - Subsystems can include support for different langauges and OS emulations

Comparing Monolithic and Microkernels

- Microkernel advantages
  - Extensibility
  - Can enforce modularity behind memory protection boundaries
  - Small kernel probably has fewer bugs
- Monolithic advantages
  - Performance
    - Difference decreasing last decade
    - But emulating other OS still too slow…