History Phase 4 (1988—): Internet

- Developed by the research community
- Based on open standard: Internet Protocol
- Internet Engineering Task Force (IETF)

- Technical basis for many other types of networks
- Intranet: enterprise IP network

- Services Provided by the Internet
  - Shared access to computing resources: telnet (1970's)
  - Shared access to data/files: FTP, NFS, AFS (1980's)
  - Communication medium over which people interact
    » email (1980's), on-line chat rooms, instant messaging (1990's)
    » audio, video (1990's, early 00's)
  - Medium for information dissemination
    » USENET (1980's)
    » WWW (1990's)
    » Audio, video (late 90's, early 00's) - replacing radio, TV?
    » File sharing (late 90's, early 00's)

LAN: Local Area Network
ISP: Internet Service Provider
NAP: Network Access Point

The Morris Internet Worm (1988)
- Internet worm (Self-reproducing)
  - Author Robert Morris, a first-year Cornell grad student
  - Launched close of Workday on November 2, 1988
  - Within a few hours of release, it consumed resources to the point of bringing down infected machines

- Techniques
  - Exploited UNIX networking features (remote access)
  - Bugs in `finger` (buffer overflow) and `sendmail` programs (debug mode allowed remote login)
  - Dictionary lookup-based password cracking
  - Grappling hook program uploaded main worm program

LoveLetter Virus (May 2000)
- E-mail message with VBScript (simplified Visual Basic)
- Relies on Windows Scripting Host
  - Enabled by default in Win98/2000
- User clicks on attachment—infected!
  - E-mails itself to everyone in Outlook address book
  - Replaces some files with a copy of itself
  - Searches all drives
  - Downloads password cracking program
- 60-80% of US companies infected and 100K European servers
History Phase 5 (1995—): Mobile Systems

- Ubiquitous Mobile Devices
  - Laptops, PDAs, phones
  - Small, portable, and inexpensive
    - Many computers/person!
  - Limited capabilities (memory, CPU, power, etc...)
- Wireless/Wide Area Networking
  - Leveraging the infrastructure
  - Huge distributed pool of resources extend devices
  - Traditional computers split into pieces. Wireless keyboards/mice, CPU distributed, storage remote
- Peer-to-peer systems
  - Many devices with equal responsibilities work together
  - Components of “Operating System” spread across globe

Datacenter is the Computer

- (From Luiz Barroso’s talk at RAD Lab 12/11)
- Google **program** == Web search, Gmail,...
- Google **computer** ==
  - Thousands of computers, networking, storage
- Warehouse-sized facilities and workloads may be unusual today but are likely to be more common in the next few years

Migration of Operating-System Concepts and Features

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mainframes</td>
<td>no software</td>
<td>time shared</td>
<td>distributed systems</td>
<td>fault tolerant</td>
</tr>
<tr>
<td>mini-computers</td>
<td>no software</td>
<td>multi-user</td>
<td>UNIX</td>
<td>UNIX</td>
</tr>
<tr>
<td>desktop computers</td>
<td>no software</td>
<td>multi-user</td>
<td>multi-user</td>
<td>UNIX</td>
</tr>
<tr>
<td>handheld computers</td>
<td>no software</td>
<td>multi-user</td>
<td>multi-user</td>
<td>UNIX</td>
</tr>
</tbody>
</table>

Implementation Issues

- Policy vs. Mechanism
  - Policy: **What** do you want to do?
  - Mechanism: **How** are you going to do it?
  - Should be separated, since both change
- Algorithms used
  - Linear, Tree-based, Log Structured, etc...
- Event models used
  - threads vs event loops
- Backward compatibility issues
  - Very important for Windows 2000/XP
- System generation/configuration
  - How to make generic OS fit on specific hardware
**Administrivia: Time for Project Signup**

- Section assignments are done
  - Watch for section assignments after the class
  - Attend new sections tomorrow
- Project Signup: Watch “Group/Section Assignment Link”
  - 4-5 members to a group
    » Everyone in group must be able to actually attend same section
  - Only submit once per group!
    » Everyone in group must have logged into their cs162-xx accounts once before you register the group
  - Due Friday 1/29 by 11:59pm

<table>
<thead>
<tr>
<th>Section</th>
<th>Time</th>
<th>Location</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>W 10:00A-11:00A</td>
<td>2 Evans</td>
<td>Matei Zaharia</td>
</tr>
<tr>
<td>102</td>
<td>W 2:00P-3:00P</td>
<td>75 Evans</td>
<td>Andy Konwinski</td>
</tr>
<tr>
<td>103</td>
<td>W 3:00P-4:00P</td>
<td>2 Evans</td>
<td>Ben Hindman</td>
</tr>
</tbody>
</table>

**Administrivia (2)**

- Cs162-xx accounts:
  - Make sure you got an account form
  - If you haven’t logged in yet, you need to do so
- Tuesday: Start Project 1
  - Go to Nachos page and start reading up
  - Note that all the Nachos code will be printed in your reader (TBA)

---

**Concurrency**

- "Thread" of execution
  - Independent Fetch/Decode/Execute loop
  - Operating in some Address space
- Uniprogramming: *one thread at a time*
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Get rid concurrency by definition
  - Does this make sense for personal computers?
- Multiprogramming: *more than one thread at a time*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP/7, Mac OS X
  - Often called “multitasking”, but multitasking has other meanings (talk about this later)
- ManyCore ⇒ Multiprogramming, right?

**Goals for Today**

- How do we provide multiprogramming?
- What are Processes?
- How are they related to Threads and Address Spaces?

---

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated by John Kubiatowicz.
The Basic Problem of Concurrency

• The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive
    access to shared resources
• OS Has to coordinate all activity
  - Multiple users, I/O interrupts, ...
  - How can it keep all these things straight?
• Basic Idea: Use Virtual Machine abstraction
  - Decompose hard problem into simpler ones
  - Abstract the notion of an executing program
  - Then, worry about multiplexing these abstract machines
  - Dijkstra did this for the “THE system”
    - Few thousand lines vs 1 million lines in OS 360 (1K bugs)

Recall (61C): What happens during execution?

Execution sequence:
- Fetch Instruction at PC
- Decode
- Execute (possibly using registers)
- Write results to registers/mem
- PC = Next Instruction(PC)
- Repeat

How can we give the illusion of multiple processors?

• Assume a single processor. How do we provide the
  illusion of multiple processors?
  - Multiplex in time!
• Each virtual “CPU” needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others...?)
• How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
• What triggers switch?
  - Timer, voluntary yield, I/O, other things

Properties of this simple multiprogramming technique

• All virtual CPUs share same non-CPU resources
  - I/O devices the same
  - Memory the same
• Consequence of sharing:
  - Each thread can access the data of every other
    thread (good for sharing, bad for protection)
  - Threads can share instructions
    (good for sharing, bad for protection)
  - Can threads overwrite OS functions?
• This (unprotected) model common in:
  - Embedded applications
  - Windows 3.1/Macintosh (switch only with yield)
  - Windows 95—ME? (switch with both yield and timer)
Modern Technique: SMT/Hyperthreading

- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If multiprocessor, should schedule each processor first
- Original technique called “Simultaneous Multithreading”
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 (“Hyperthreading”), Power 5

---

How to protect threads from one another?

- Need three important things:
  1. Protection of memory
     » Every task does not have access to all memory
  2. Protection of I/O devices
     » Every task does not have access to every device
  3. Protection of Access to Processor:
     Preemptive switching from task to task
     » Use of timer
     » Must not be possible to disable timer from user code

---

Recall: Program’s Address Space

- Address space ⇒ the set of accessible addresses + state associated with them:
  - For a 32-bit processor there are \(2^{32} = 4\) billion addresses
- What happens when you read or write to an address?
  - Perhaps Nothing
  - Perhaps acts like regular memory
  - Perhaps ignores writes
  - Perhaps causes I/O operation
    » (Memory-mapped I/O)
  - Perhaps causes exception (fault)

---

Providing Illusion of Separate Address Space:
Load new Translation Map on Switch
Traditional UNIX Process

- Process: Operating system abstraction to represent what is needed to run a single program
  - Often called a "HeavyWeight Process"
  - Formally: a single, sequential stream of execution in its own address space
- Two parts:
  - Sequential Program Execution Stream
    » Code executed as a single, sequential stream of execution
    » Includes State of CPU registers
  - Protected Resources:
    » Main Memory State (contents of Address Space)
    » I/O state (i.e. file descriptors)
- Important: There is no concurrency in a heavyweight process

How do we multiplex processes?

- The current state of process held in a process control block (PCB):
  - This is a "snapshot" of the execution and protection environment
  - Only one PCB active at a time
- Give out CPU time to different processes (Scheduling):
  - Only one process “running” at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
    - Sample mechanisms:
      » Memory Mapping: Give each process their own address space
      » Kernel/User duality: Arbitrary multiplexing of I/O through system calls

CPU Switch From Process to Process

- This is also called a "context switch"
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
  - Less overhead with SMT/hyperthreading, but... contention for resources instead

Diagram of Process State

- As a process executes, it changes state
  - new: The process is being created
  - ready: The process is waiting to run
  - running: Instructions are being executed
  - waiting: Process waiting for some event to occur
  - terminated: The process has finished execution
Process Scheduling

- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are Scheduling decisions
  - Many algorithms possible (few weeks from now)

What does it take to create a process?

- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix fork())
  - Semantics of Unix fork() are that the child process gets a complete copy of the parent memory and I/O state
  - Originally very expensive
  - Much less expensive with "copy on write"
- Copy I/O state (file handles, etc)
  - Medium expense

Process =? Program

- More to a process than just a program:
  - Program is just part of the process state
  - I run emacs on lectures.txt, you run it on homework.java - Same program, different processes
- Less to a process than a program:
  - A program can invoke more than one process
  - cc starts up cpp, cc1, cc2, as, and ld

Multiple Processes Collaborate on a Task

- High Creation/memory Overhead
- (Relatively) High Context-Switch Overhead
- Need Communication mechanism:
  - Separate Address Spaces Isolates Processes
  - Shared-Memory Mapping
    » Accomplished by mapping addresses to common DRAM
    » Read and Write through memory
  - Message Passing
    » send() and receive() messages
    » Works across network
Shared Memory Communication

- Communication occurs by "simply" reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems

Inter-process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message) - message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus, syscall/trap)
  - logical (e.g., logical properties)

Modern "Lightweight" Process with Threads

- Thread: a sequential execution stream within process (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: a single program made up of a number of different concurrent activities
  - Sometimes called multitasking, as in Ada...
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)
  - Heavyweight Process = Process with one thread
- Single and Multithreaded Processes
  - Threads encapsulate concurrency: "Active" component
  - Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
  - Why have multiple threads per address space?
Examples of multithreaded programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done

Examples of multithreaded programs (con't)

- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing
- Some multiprocessors are actually uniprogrammed:
  - Multiple threads in one address space but one program at a time

Thread State

- State shared by all threads in process/addr space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- State “private” to each thread
  - Kept in TCB = Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack - what is this?
- Execution Stack
  - Parameters, Temporary variables
  - return PC is kept while called procedures are executing

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
    B();
    printf(tmp);
}
B() {
  C();
}
C() {
  A(2);
}
A(1);
```
Classification

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
<td></td>
</tr>
<tr>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc)</td>
<td>Mach, OS/2, Linux Windows 9x??</td>
<td>Win NT to XP Solaris, HP-UX, OS X</td>
</tr>
</tbody>
</table>

Real operating systems have either
- One or many address spaces
- One or many threads per address space
Did Windows 95/98/ME have real memory protection?
- No: Users could overwrite process tables/System DLLs

Example: Implementation Java OS

- Many threads, one Address Space
- Why another OS?
  - Recommended Minimum memory sizes:
    » UNIX + X Windows: 32MB
    » Windows 98: 16-32MB
    » Windows NT: 32-64MB
    » Windows 2000/XP: 64-128MB
  - What if we want a cheap network point-of-sale computer?
    » Say need 1000 terminals
    » Want < 8MB
- What language to write this OS in?
  - Java/Lisp? Not quite sufficient - need direct access to HW/memory management

Summary

- Processes have two parts
  - Threads (Concurrency)
  - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (yield(), I/O operations) or involuntary (timer, other interrupts)
- Protection accomplished restricting access:
  - Memory mapping isolates processes from each other
  - Dual-mode for isolating I/O, other resources
- Book talks about processes
  - When this concerns concurrency, really talking about thread portion of a process
  - When this concerns protection, talking about address space portion of a process