





**Note:** You are looking at a static copy of the former PineWiki site, used for class notes by James Aspnes from 2003 to 2012. Many mathematical formulas are broken, and there are likely to be other bugs as well. These will most likely not be fixed. You may be able to find more up-to-date versions of some of these notes at <http://www.cs.yale.edu/homes/aspnes/#classes>.

Consolidated lecture notes for CS422 as taught in the Spring 2007 semester at  Yale by  Jim Aspnes.

## Contents

1. OperatingSystemsOverview
2. What is an operating system?
3. History
4. What abstractions does an OS provide?
5. Recurring themes
6. OperatingSystemStructure
7. Hardware
8. CPU operation
9. I/O
10. BIOS
11. Memory
12. BootProcess
13. Boot process: general
14. PC architecture
15. IntelAssemblyProgramming
16. What you should read instead of this page
17. Real mode x86 programming
  1. Opcodes and operands
18. Processes
19. Processes
20. Multitasking
  1. Option 1: Cooperative multitasking
    1. Mechanism
  2. Option 2: Pre-emptive multitasking
    1. Mechanism
    2. Gotchas
21. Scheduling
22. Threads
23. Why use threads?
24. Thread creation: user view
25. User threads vs kernel threads
26. Why not use threads?
27. ConcurrencyControl
28. The need for concurrency control
  1. Example: Updating the web hits counter
  2. Race conditions
29. Mutual exclusion
  1. Naive approach
  2. Another naive approach
  3. Peterson's algorithm
  4. Preventing pre-emption
  5. Hardware support for locking
30. Spinlocks
31. Semaphores
  1. Applications
  2. Implementation
32. Monitors
33. Condition variables
34. Deadlock detection and avoidance
35. Deadlock
36. What is deadlock?
  1. Example: too little memory
  2. Example: I/O
  3. Example: bidirectional pipe
37. Processes and resources
38. Necessary conditions
39. Resource-allocation graphs
40. Preventing deadlock
41. Avoiding deadlock
  1. The Banker's algorithm
42. Dealing with deadlock
43. ProcessorScheduling
44. Processor scheduling: basics
  1. CPU burst cycle
  2. Types of jobs
  3. Performance measures
45. When scheduling happens
46. Algorithms
  1. First-come first-served
  2. Round robin
  3. Shortest-job-first
  4. Priority scheduling
  5. Multilevel queue scheduling
  6. Multilevel feedback-queue scheduling
47. Multiple processors
48. Algorithms in practice
49. Evaluation
50. InterProcessCommunication
51. Motivation

- 51. Motivation
- 52. Shared memory
- 53. Message passing
  - 1. Interface
    - 1. Channels and naming
    - 2. Sending a message
    - 3. Receiving a message
  - 2. Implementation
    - 1. Using shared memory
    - 2. Across a network
  - 3. Exceptions
- 54. Remote procedure calls
- 55. Remote method invocation
- 56. Efficiency issues
- 57. MemoryManagement
- 58. MemoryLayout
- 59. Fixed addresses chosen by the programmer
- 60. Fixed address chosen by the linker and/or loader
- 61. Dynamic linking
- 62. Dynamic loading
- 63. MemoryProtection
- 64. Segmentation approach
- 65. Paging
- 66. Software memory protection
- 67. Paging
- 68. Basic idea
- 69. Translation Lookaside Buffer
- 70. Page table tricks
- 71. Page table structures
- 72. VirtualMemory
- 73. Terminology
- 74. Handling a page fault
- 75. What can we page?
- 76. Performance
- 77. Page replacement strategies
- 78. Buffering strategies
- 79. Other virtual memory tricks
  - 1. Shared pages
  - 2. Copy-on-write
  - 3. Memory-mapped files
- 80. Bad outcomes
- 81. InputOutput
- 82. I/O devices (user view)
- 83. I/O devices (hardware view)
  - 1. Controllers
  - 2. Input and output instructions
  - 3. Memory-mapped I/O
  - 4. Hybrid approach
  - 5. Direct Memory Access (DMA)
- 84. A typical I/O operation from start to finish
- 85. I/O system architecture
  - 1. Device driver architecture
    - 1. What a device driver module looks like
    - 2. Loading a device driver
    - 3. Risks
  - 2. Device-independent I/O components
- 86. Example: console/keyboard driver
  - 1. Output
  - 2. Input
- 87. Example: disk drivers
- 88. BlockDevices
- 89. The problem with mass storage
- 90. Hard disk structure
- 91. Bad blocks and error correction
- 92. Disk scheduling
- 93. Block device driver implementation
- 94. Buffering, caching, and prefetching
- 95. Network-attached storage
- 96. Removable storage
- 97. Multiple disks
- 98. Other block devices
- 99. FileSystems
- 100. Interface
  - 1. Files
    - 1. Structure
    - 2. Naming
    - 3. Metadata
    - 4. Operations
    - 5. Access methods
  - 2. Directories
  - 3. Mount points
  - 4. Special files
  - 5. Special file systems
    - 1. /proc
    - 2. Archival file systems
    - 3. Distributed file systems
    - 4. Unions
    - 5. Encrypted filesystems
  - 6. Access control
- 101. Implementation
  - 1. Disk layout
    - 1. Block size
    - 2. Tracking blocks in a file
    - 3. Tracking free blocks
    - 4. Directories
    - 5. Physical layout
  - 2. Pathname translation
  - 3. Caching
- 102. Consistency checking

- 1. How inconsistencies arise
- 2. Recovering from inconsistencies
- 3. Preventing inconsistencies
- 103. More info
- 104. LogStructuredFilesystem
- 105. The first step: journaling
- 106. Ditching the main filesystem
  - 1. The inode map
  - 2. Checkpoints
- 107. Space recovery
  - 1. Segment summary data
  - 2. Data compaction
  - 3. Which segments to clean?
- 108. Performance
- 109. Why aren't all filesystems log-structured?
- 110. Networking
- 111. Local-area vs wide-area networks
  - 1. LANs
  - 2. WANs
- 112. Addressing: IP
- 113. UDP and TCP
  - 1. TCP: more details
    - 1. Connection setup
    - 2. Receive window
    - 3. Retransmission control
    - 4. Shutting down a connection
- 114. Higher-level protocols
- 115. OS implications
- 116. Socket interface
- 117. Effect of failures
- 118. NetworkFileSystems
- 119. Naming issues
- 120. Caching and consistency
- 121. Stateful vs stateless servers
- 122. Data representation
- 123. Case study: NFS version 2
- 124. DistributedSystems
- 125. Examples of distributed systems
- 126. Distributed coordination
- 127. Timestamps
- 128. Distributed mutual exclusion
- 129. Distributed transactions
- 130. Agreement protocols
- 131. Paxos
- 132. The Paxos algorithm
- 133. Informal analysis: how information flows between rounds
- 134. Safety properties
- 135. Learning the results
- 136. Liveness properties
- 137. ComputerSecurity
- 138. Goals of computer security
- 139. Protection
  - 1. Principle of least privilege
  - 2. Users, roles, and groups
  - 3. Protection domains
  - 4. Access matrices
    - 1. Who controls the access matrix?
    - 2. Copy rights
    - 3. Confinement
  - 5. Implementation
    - 1. Access control lists
    - 2. Capabilities
- 140. Implementation
  - 1. Authentication
    - 1. Something you know
    - 2. Something you have
    - 3. Something you are
    - 4. Two-factor authentication
  - 2. Authorization
  - 3. Enforcement
  - 4. Intrusion detection and recovery
- 141. Practical attacks and defenses
  - 1. Attacks on individual machines
    - 1. Buffer overflow and other code injection exploits
    - 2. Viruses
    - 3. Trojan horses
    - 4. Worms
  - 2. Network attacks
    - 1. Replay attacks
    - 2. Man-in-the-middle attacks
- 142. How much of a problem is security?
- 143. Virtualization
- 144. A brief history of operating systems
- 145. Why virtualize?
- 146. Virtualization techniques
  - 1. Emulation
  - 2. Hypervisors
    - 1. Using breakpoints
    - 2. Using code rewriting
    - 3. Using paravirtualization
    - 4. Using additional CPU support
- 147. Applications
- 148. UsingBochs
- 149. Basic use
- 150. Debugging
  - 1. Breakpoints and breakpoint gotchas
- 151. Alternatives

```

152. UsingSubversion
153. Basic concepts
154. Subversion commands: basics
    1. svn co [url]
    2. svn up
    3. svn commit
155. Getting information
    1. svn log
    2. svn status
    3. svn diff
    4. svn cat
156. File operations
    1. svn add [file]
    2. svn mkdir [directory]
    3. svn rm [file]
    4. svn cp [source] [destination]
    5. svn mv [source] [destination]
157. Fixing things
    1. svn revert [file]
    2. svn resolved [file]

```

# 1. OperatingSystemsOverview

## 2. What is an operating system?

- Difficult legal question in Europe and in Clinton-era US
- We can consider several definitions:

### User's view

Software that came with my computer that I can't get rid of.

### Textbook view

Software that hides details of the underlying hardware, mediates access to resources, and enforces security policies.

### System programmer's view

Anything that runs in ring 0.

- We will mostly adopt the programmer's view.
- Essential idea is that OS provides an abstraction layer on top of the bare hardware.
  - Typical modern system: Hardware -> BIOS -> kernel -> libraries -> user programs.
  - Where to draw the boundary? E.g., do libraries fold into the OS?
  - Natural boundary is the system call.
    - Modern CPUs have different protection levels or rings (4 on Intel CPUs, but nobody uses anything but ring 0 and ring 3).
      - Kernel code (ring 0) has full access to hardware.
      - User code (ring > 0) doesn't.
    - If user code tries to execute a privileged instruction, it traps to kernel code.
      - On IA32, uses same mechanism as for hardware interrupts:
        - CPU switches to ring 0.
        - Various magic happens in the background (memory system switches context, interrupts may be disabled).
        - IP jumps to new location based on interrupt vector table in low memory.
          - => kernel is whatever the interrupt table sends you to?
      - Kernel can now decide what to do in response to the "illegal instruction"
        - Userspace process is doing something wrong => kill it!
        - Or maybe it just needs some help from the kernel => help it.
          - This is the system call case.
          - Eventually return using specialized opcode (`iret` on IA32)
          - `int/iret` acts like very expensive context-switching procedure call.

## 3. History

- In the beginning: no OS
  - Program raw hardware with a soldering iron (still used in 23rd century according to Star Trek!)
  - Eventually graduate to punch cards/paper tape/magnetic tape.
  - Model is that a batch-mode program gets complete control over the entire machine.
- IBM virtual machines
  - Build one machine that pretends to be many machines.
  - Each individual machine still acts exactly like the classic IBM 1401 (or whatever) that you programmed with punch cards.
    - Includes virtual punch card readers, magnetic tapes, etc. with bug-for-bug compatibility.
    - We actually haven't gotten very far from this.
  - Most programming is still done in batch mode.
- Timesharing
  - Build abstract processes that run on an idealized virtual machine.
  - Works nicely for interactive systems with many users.
  - Enabling economic change: cheap computers and expensive users.
    - Happened first in well-funded research laboratories e.g. Xerox PARC, Bell Labs, MIT AI Lab.
    - Now we all do it.
- Distributed systems
  - Get thousands or millions of machines to work together.
  - Enabling economic change: cheaper to buy many small computers than one big one; cheap networks.

- Dates back to 1960's parallel machines like ILLIAC, now shows up as peer-to-peer systems and server farms.
- We mostly won't talk about these.

## 4. What abstractions does an OS provide?

- Hide ugly details of hardware interfaces.
  - Most hardware ships with software drivers.
  - This means hardware designers don't have to provide nice interfaces since driver can paper it over.
    - This is usually a good thing!
    - Software is easier to change than hardware
      - Can upgrade capabilities by upgrading drivers.
      - Can fix bugs without burning new ICs.
    - Software is cheaper than hardware: can exploit brainpower of the CPU
      - Example: WinModems
      - But sometimes we want the speedup we get from hardware (e.g. graphics cards)
  - OS goal: provide a standardized interface to device drivers.
    - New picture: hardware -> BIOS + device drivers -> kernel -> libraries -> user programs.
  - Secondary goal: convince hardware designers to write drivers for your OS
    - Works best if you own a large chunk of the OS market.
    - Obstacle to new OS development: even experimental OS's have to pretend to be Windows or Linux to device drivers!
- Pretend we have more resources than we really have.
  - Process abstraction
    - Yes, you only bought 1 CPU (or 2, or next year 8), but we'll pretend you have infinitely many.
    - This works by *time-sharing* CPU(s) between processes.
      - **Cooperative multitasking:** polite processes execute *yield* system call, kernel switches to another process.
        - This lets a runaway process freeze your system!
        - But it's easy to implement.
        - Used in practice in pre-NT Windows, pre-OSX MacOS, even today in many embedded OSs (e.g. PalmOS).
      - **Pre-emptive multitasking:** kernel switches to another process whether you like it or not!
        - No runaway processes
        - But requires more work in kernel *and* in processes since they have to deal with concurrency issues.
    - Why this works: most programs have "bursty" demand on CPU.
      - If you are waiting  $10^{10}$  clock cycles for the user to click the mouse, why not let somebody else use the CPU?
      - Same thing for  $10^8$ -cycle disk operations.
      - With good scheduling, nobody will notice we are doing this.
    - Further design decisions:
      - How to choose which processes to schedule? E.g. real-time, priorities, fair-share: many possible policies.
      - How much context to store for each process?
        - E.g. threads vs processes: does a process get its own address space in addition to its own virtual CPU?
        - Some historical oddities that persist for convenience like current working directories.
      - How to manage processes?
  - Virtual memory abstraction
    - We'll take small fast memory and a big slow disk and combine them to get big fast virtual memory.
      - Only works because most memory isn't used very often.
      - Usually requires hardware support (memory manager)
    - Bonus: each process gets its own address space
      - Security payoff: bad process can't scribble on good process
      - Compiler payoff: don't need to write position-independent code
- I/O abstractions
  - Next step up from hardware drivers: make all your devices look the same
  - Keyboards, printers, console display, the mouse, many USB devices: all streams of bytes.
  - Disks, flash drives, remote network storage: all blocks of bytes.
    - Old days: all stacks of 80-column EBCDIC punch cards
  - Goal is to simplify programmers' lives.
    - Why in kernel and not in libraries? Easier to change kernel for new hardware since we have to bring in device drivers anyway.
- Filesystem abstractions
  - Present big-bag-of-bits as structured collection of files.
    - File is typically a big-bag-of-bits.
    - But there is a tree structure organizer files into directories/folders.
    - Files may also have metadata.
      - What it is called.
      - Who has writes to the file.
      - When it was created.
      - What program to use to read it.
    - Goal is to provide enforced common language for application program developers.
    - Unix approach: represent things that aren't files in the filesystem anyway
      - E.g. `/dev/tty`, `/proc/cpuinfo`, `/proc/self/cmdline`.
      - Allows reuse of same interface everywhere.
    - Things to think about:
      - Why not do this at the library level?
      - Why a tree structure when databases dumped this design for tables?
        - Note that attempts to bring in tables (e.g. in Windows) have generally failed.
      - Why unstructured files?
        - Old days: all stacks of 80-column EBCDIC punch cards
        - Windows: text vs binary files

- These seem to mostly be holdovers.
- Security
  - Any time we provide an abstraction, we can enforce limits on what you can do with it
  - Process limits
  - File permissions
  - I/O permissions
  - In some systems, very fine-grained control, SECRET/TOP SECRET enforcement, etc.
- Other things that get shoved into the kernel for performance.
  - E.g. window systems

## 5. Recurring themes

- Suffering
  - OS development happens without all the nice tools an OS gives you.
    - This makes kernel programming harder.
      - Have to use low-level languages and assembly.
      - Debugging tools are limited.
  - You are also running on the bare hardware.
    - Mistakes are harshly punished.
- Modularity
  - OS is a large software system like any other => we can't build it unless we can divide up the work
  - Layered approach
    - Build up from core abstractions adding new functionality at each layer
      - E.g.: BIOS -> device drivers -> memory management -> process management -> I/O management -> network abstractions -> filesystem
      - Getting the order of the layers right can be tricky: what if I want to swap VM out to files?
  - Microkernel approach
    - We like processes so much we'll push everything we can out into userspace.
    - Advantage: can restart your filesystem daemon without rebooting the whole machine.
    - Disadvantage: where are you loading your filesystem daemon from?
    - Very active area of research in 1980's and 1990's, but then most people gave up because of bad performance.
    - (But still used in some niches and performance has been getting better.)
  - Exokernel approach
    - Provide minimal partitioning of resources and then push everything into the libraries.
    - Generally not seen in the wild.
    - Main selling point: cool name.
  - Monolithic/modular kernel approach
    - Kernel is one giant program.
    - But use OO techniques to modularize it.
      - E.g. standard device driver interface with dynamic loading.
    - Advantage is that internal calls in the kernel are cheap procedure calls instead of expensive context switches.
    - Disadvantage is that one bad module can destroy your entire system.
    -
  - Virtualization
    - Run an entire OS inside a process
    - Rationale: protects system against an OS you don't trust
    - Possible other rationale
      - Systems are now being built from multiple processes
      - => need a second level of process abstraction to encapsulate whole systems
      - E.g. virtual WWW servers.
      - Possibly a sign that process hierarchy is too flat: analogy to 1960's BASIC/FORTRAN subroutines that couldn't be recursive.
- Security
  - Why separate address spaces/file permissions/etc.?.
    - Can't trust users, they make mistakes.
    - Can't trust programmers, they make mistakes that execute at  $10^{10}$  instructions per second.
    - Can't trust poorly-socialized script kiddies, they don't share your agenda.
    - Good fences contain the damage (but note most OSs don't really try to protect against malicious users).
  - Why reboot?
    - Bugs produce errors that accumulate over time.
    - Restarting processes from scratch resets to a known state.
    - (Same reason humans aren't immortal: easier to manufacture new humans than repair the old ones.)
    - (Also same reason people throw away their virus-infested PCs rather than disinfect them.)
  - Trade-off: "A computer is only secure when it's unplugged."
    - Too much security makes users bypass it.
    - Too much security destroys your work.
- Performance
  - Famous old SGI story
    - Suppose we install a kernel on  $10^8$  machines.
    - Every day each machine wastes 1 second of a user's life.
    - $10^8$  machines times  $10^3$  days =  $10^{11}$  wasted seconds = 30 wasted lifetimes.
    - => bad coders are worse than serial killers.
  - Less of an issue over time as machines get faster
  - But defining constraint on early OS development (also why PC OS's recapitulated mainframe OS's 20-25 years later)
  - Question to ask: what features of my OS are solutions to performance issues that are no longer relevant?
- Consistency

- Backwards-compatibility is a huge issue, especially for commercial OSs.
  - Free OS users tolerate disasters as part of the whole hair-shirt mentality :-]
  - Microsoft story
    - When SimCity broke because Windows closed the use-after-free loophole, users blamed Windows and not SimCity
    - So Windows now recognizes SimCity and runs a special compatibility version of malloc/free
- If your API is too complicated or too restrictive, natural selection comes into play.
- Sometimes crummy interfaces persist through lock-in or survival of the least inadequate.
  - E.g. NFS, X11, sockets, and other de facto sub-standards.
- Hard to guess what users want => safe OS research makes existing interfaces 10% faster.
- But you become famous for new approaches that catch on.
- Separation between policy and mechanism
  - We can't predict exactly what scheduling/security/etc. policies users will want.
  - So build the OS so we can change the policy later.
  - Modularity helps here just as in regular programming.

## 6. OperatingSystemStructure

Description of how a typical OS is organized, with emphasis on x86.

## 7. Hardware

- One or more CPUs
- Various devices
  - Keyboard
  - Graphics controller
  - Storage devices
  - Wikipedia: Scratch\_monkey
- Memory controller
  - Sits between CPU/devices and physical memory
  - May implement virtual memory
- Interrupt controller
  - Sits between devices and CPU(s)
  - Chooses how and when to divert CPU from normal computation

## 8. CPU operation

### Normal operation

- Fetch next instruction based on IP register
- Do what it says

### Interrupts

- Device signals some event
- Interrupt controller kicks CPU
- CPU calls interrupt handler
  - Disables further interrupts
  - Pushes current state onto the stack
  - (In protected mode) switches to ring 0
  - Jumps to interrupt handler at standard location (new CS:IP stored at physical address 0x04\*(interrupt number) for x86 real mode)
  - Interrupt handler does its thing
  - Returns (using `iret` on x86)
    - Pops state and re-enables interrupts

### Traps

- Like interrupts, except something bad happened (division by 0, overflow, segmentation faults, etc.)
- Mechanism is exactly the same on x86 except for interrupt number

### System calls

- Simulate interrupt using `int` instruction (old x86 method) or `sysenter` (newer x86's).
- Typically just one trap number is used for all system calls (0x2e in Windows, 0x80 in Linux).
  - Further dispatch in the kernel through handler table indexed by syscall number (typically passed in AX register).
- Details of system call are stored in registers.
- Return value(s) come back in registers.
- Note that system calls are expected to change register state, unlike interrupts and traps which are expected to be invisible to the running program.

### BIOS calls

- Like system calls, only jumping into ROM supplied with your motherboard.
  - Standard locations and interface dating back to 1981 IBM PC.
  - See Wikipedia: BIOS\_Interrupt\_Calls for a sketchy list.

- INT 13 for disk access (`int $13` in AT&T syntax)
- Mostly used in real mode by DOS.
- Modern OSs bypass BIOS for most operations.
  - Except power management and other whizzy system control features that are closely tied to the motherboard hardware.

## 9. I/O

- Interrupts don't provide much information beyond "look at me! look at me!"
- Data is shoved back and forth through memory.
- For early part of the course we'll let the BIOS deal with this.

## 10. BIOS

Stands for *Basic Input/Output System*. It's what comes built-in to your motherboard and what is called first at boot time. See `BootProcess`. It may also provide a primitive core OS that manages simple devices (although most modern OSs bypass this interface).

## 11. Memory

OS memory layout is pretty much the same as in a process. We have executable code (the *text segment*), preinitialized data (the *data segment*), dynamically-allocated data (the *heap*), and a stack.

In x86 real mode these segments are typically pointed to by segment registers, e.g. CS for the code segment, DS for the data segment, SS for the stack segment. See `IntelAssemblyProgramming`.

Certain physical addresses are reserved in IBM PC architecture. This puts constraints on where the OS can operate in real mode.

A typical setup is:

00000-003FF	Interrupt-vector
00400-01000	Buffers and system stuff
01000-90000	Kernel memory
90000-9FFFF	Kernel stack
A0000-	System ROM
B8000-	Video RAM
FFFF0-FFFFF	BIOS entry point

Important point: Physical addresses above FFFFF are inaccessible in real mode. Physical addresses above A0000 (640K) are here-there-be-monsters territory, not usable by the OS.

---

CategoryOperatingSystemsNotes

## 12. BootProcess

## 13. Boot process: general

To **boot** an OS we start with the bare hardware and whatever it provides, and load in increasingly large pieces of the OS until we have our system fully up and running. Typically this may involve several stages:

1. Load up or have pre-stored some initial bootstrap routine.
2. Bootstrap routine finds and initializes storage devices.
3. Bootstrap routine reads *boot loader* from some boot device (e.g. a hard drive, CD-ROM, or USB key; or in the bad old days: punch cards, paper tape, cassette tapes).
4. Boot loader reads kernel from boot device.
5. Kernel initializes rest of hardware.
6. Kernel loads user-space startup code.
7. Startup code brings up user system.

## 14. PC architecture

On an x86 PC, the initial bootstrap is handled by the BIOS, which lives in ROM wired onto the motherboard. The BIOS expects to run in real mode (simulating a vintage 1981 8088-based PC), so that's what the system comes up in. The details of the process are as follows:

1. BIOS setup.
  1. On reset, the CPU switches to real mode and jumps to FFFF0. (Really FFFFFFFF0, but address gets truncated.) This is the BIOS entry point.
  2. BIOS executes Power On Self Test (POST).
  3. BIOS calls initialization routines provided in ROM by video card (at 0xC0000) and disk controller (0xC8000).
  4. Memory test.
  5. Look for I/O devices and assign them interrupt vectors.



6. Look for bootable drive.
  - Boot sector at (0,0,1) tagged with 55 AA marker.
7. Load boot sector to address 0000:7c00.
8. Jump to boot sector.
2. Boot sector.
  1. Initialize stack.
    - SS gets 9000
    - SP gets FFFE (for 16-bit mode)
  2. Load kernel to some standard location, e.g. 0000:1000
  3. Jump to kernel.

The rest of the setup is up to the kernel, and is likely to include switching to protected mode, setting up memory management and interrupt handling, further device initialization, initializing the filesystem, loading user-space programs, etc.

---

Category: Operating Systems Notes

## 15. Intel Assembly Programming




You will need to learn some IA-32 assembly language programming for CS422. This document starts with pointers to IA-32 assembly language documentation and then continues with some specific details that might be more directly relevant to the course.


## 16. What you should read instead of this page


My recommendation is to start with Kai Li's notes on IA-32 programming at <http://www.cs.princeton.edu/courses/archive/fall06/cos318/docs/pc-arch.html>.

Then move on to the


Official IA32 Intel architecture software developer's manuals:


- *Volume 1: Basic Architecture*:  [ia32-vol1.pdf](#).
- *Volume 2: Instruction Set Reference Manual*:  [ia32-vol2.pdf](#).
- *Volume 3: System Programming Guide*:  [ia32-vol3.pdf](#).

One trap in all of this is that `gas`, the GNU assembler, uses a different syntax for assembly language from the Intel style. See the  `gas` manual for an extensive discussion of this.

A less authoritative guide to x86 assembly written in `gas` syntax can be found at  [http://en.wikibooks.org/wiki/X86\\_Assembly](http://en.wikibooks.org/wiki/X86_Assembly).

Two helpful gcc tricks:

1. You can find out what the C compiler turns a given chunk of C code to using `gcc -S file.c`.
2. You can make use of assembly-language code inside C code using the `asm` syntax in gcc. (Note that you may need to provide extra directives to the assembler if you are coding for unusual targets, e.g. 16-bit mode boot loaders should include `asm( ".code16gcc" );` at the top of every C file). With sufficiently clever use of this feature you can keep most of your code in C and use assembly only for very specific low-level tasks (like manipulating segment registers, calling BIOS routines, or executing special-purpose instructions that never show up as a result of normal C code like `int` or `iret`). See the  gcc documentation for more on using the `asm` mechanism.

 [http://devpit.org/wiki/Compiler\\_Options\\_for\\_Creating\\_Odd\\_Binaries](http://devpit.org/wiki/Compiler_Options_for_Creating_Odd_Binaries) has some nice discussion of how to generate unusual binaries using gcc and ld.

## 17. Real mode x86 programming

For the first few assignments you will be working in **real mode**, which is the x86 architectures mode that emulates a vintage 1976 8086 CPU. The main advantage of real mode is that you have a flat 20-bit address space running from 0x00000 to 0xFFFFF and no memory management or protection issues to worry about. The disadvantage is that you only have 16-bit address registers to address this 20-bit space.

The trick that Intel's engineers came up with to handle this problem was to use **segmented addressing**. In addition to the four 16-bit **data registers** AX, BX, CX, and DX and the four 16-bit **address registers** BP, SP, DI, and SI there are four 16-bit **segment registers** (later extended to six) CS, DS, ES, and SS. Addresses in real mode are obtained by combining a 16-bit segment with a 16-bit offset by the rule  $0x10 * \text{segment} + \text{offset}$ . This operation is commonly written with a colon, so for example the physical address of the stack is `SS:SP = 0x10*SS+SP`.

### 17.1. Opcodes and operands

Instructions typically operate one or two registers, immediate values (i.e. constants), or memory locations. An instruction is written as an **opcode** followed by its **operands** separated by commas.

Perhaps the most useful opcode is `mov`, equivalent to an assignment. It comes in several flavors depending on the size of the value you are moving: `movb` = 1 byte, `movw` = 2 bytes, `movl` = 4 bytes. If you don't add the size tag the assembler picks one based on the size of the destination operand. In AT&T syntax as used in `gas` the first operand is the source, the second is the destination (this is backwards from Intel syntax).

Here is `movw` conjugated with various addressing modes:

```
movw $4, %ax    # copy the constant 0x04 into AX
movw 4, %ax     # copy the contents of memory location DS:0x04 into AX
movw %ax, %bx   # copy the contents of AX to BX
movw (%si), %ax # copy the contents of memory location pointed to by SI to AX
movw 4(%bp), %ax # copy the contents of location at SS:BP+4 to AX
movw %ax, 12(%es:%si) # copy AX to location ES:SI+12
```

Note that most of the time we don't bother specifying the segment register, but instead take the default: CS for instructions, SS for the stack (`push`, `pop`, anything using BP or SP), and DS for everything else except string instruction destinations, which use ES. But we can always specify the segment register explicitly as in the last example.

Arithmetic operations follow the pattern for `mov`, e.g.

```
addw %ax, %bx   # add AX to BX (in C: bx += ax)
incw 4(%bp)     # increment *(SS:BP+4)
cmpw %cx, %dx   # compare CX to DX; like subtraction but throws away result
```

Control flow is handled by jump instructions. Targets are **labels** which are followed by a colon (think `goto` in C):

```
loop:
    jmp loop      # very fast loop
```

Conditional jumps are often more useful:

```
    movw $0, %ax
loop16:
    incw %ax
    cmpw %ax, $10
    jle loop16    # jump if A <= 0x10
```

Two specialized jumps are `call` and `ret`, which are used for procedure calls and returns. These push or pop the IP register on the stack as appropriate. The `int` and `iret` instructions are slightly more complicated variants of these used for simulating interrupts; we'll run into these more later.

See the documentation for many more instructions.

---

CategoryOperatingSystemsNotes

## 18. Processes

**Processes** are the abstraction that lets us divide up the CPU and other system resources between different programs. Historically, the idea of a process grew out of the separate virtual machines of classic mainframe OS architectures. Over time, some of the features traditionally associated with a process (such as a separate address space) have been shed, giving rise to a newer notion of a **thread**, which slices up the CPU without dividing up other resources. Curiously, this is almost the reverse of what happened with subroutines, which started out only affecting the instruction pointer in many early programming languages (BASIC, FORTRAN) but later added separate address spaces (local variables, objects). We'll follow the historical pattern by talking about processes first.

## 19. Processes

A process typically encompasses a bunch of things bundled together:

- In user space:
  1. Access to the CPU!
  2. An address space, including:
    1. The running program.
    2. Pre-initialized data.
    3. A heap for dynamic allocation.
    4. A stack.
- In kernel space:
  1. Pointers to various system resources used by the process (e.g. Unix file descriptors).
  2. Odd but useful historical legacies like the *current working directory*.
  3. Security information: What privileges the processes has, what user it is running on behalf of.
  4. Internal bookkeeping stuff: resource tracking, address-space management.

In order to implement processes, we need to solve two problems:

1. How do we divide up the CPU between processes?
2. How do we keep all the other bits of processes separate?

The first problem can be further divided into building a mechanism that lets the CPU switch between processes (**multitasking**) and choosing how to decide which process it should run next (**scheduling**). (This is an example of the split between *mechanism* and *policy* that recurs throughout OS design and development.) The second problem is a question of allocating the right data structures, which we'll talk about below, and setting up the memory system to give each process its own address space, which we'll defer to VirtualMemory.

## 20. Multitasking

Here's a picture of a single process A running along doing its stuff:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Here are two processes A and B running along in parallel (say on a system with multiple CPUs):

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

And here are A and B sharing a single CPU:

```
AAAAAAA      AAAAAA      AAAAAA
BBBBBBBBBBBBB BBBB      BBBB      BBBB
```

The trick here is that as long as A and B don't have the ability to detect that they've been shunted aside, they won't be able to tell the difference between this picture and the previous picture. So when designing a multitasking system, we just have to make sure that we clean up anything that B does when it's running so that A doesn't notice and vice versa.

We also have to figure out how to switch the CPU's instruction pointer from A's code to B's code. Fortunately this is not a new problem: We do it all the time when executing a procedure call. The difference is between executing a procedure call and doing a **context switch** between processes is that (a) we generally don't expect to come back, and (b) the "calling" process doesn't necessarily know that the context switch is happening. But since we already have mechanisms for calling other code, we can exploit this to build context switches. There are two typical ways this is done, depending on whether we expect our processes to cooperate or not.

## 20.1. Option 1: Cooperative multitasking

Here we implement a context-switch as a first-class operation triggered by the running process, typically through a `yield` system call. So we might write code like

```
#include <myfancyos.h>

int
main(int argc, char **argv)
{
    while(1) {
        check_for_input();
        do_something_about_it();
        yield();
    }

    exit();
}
```

(Note that `exit` will also be a system call or a library wrapper to a system call.)

### 20.1.1. Mechanism

Here most of the code just runs like standard C code. But when we execute `yield`, several strange things happen under the hood:

1. The CPU state is saved. Typically this involves pushing the IP and process registers onto the stack. On an x86 machine, this could be done using the `CALL` instruction to save the IP (automatically done as part of the `yield` procedure call if it is implemented as a library procedure; for software interrupts we rely on similar behavior in `INT`) and the `PUSHAD` instruction to push all the registers. For other architectures that don't provide such convenient tools the process may be more complicated, but is usually a simple matter of programming. The last step is to save the stack pointer itself in the **process control block**—the data structure that holds all the extra junk the kernel has to keep track of for the process.
2. Any other process state that shouldn't be clobbered is saved. At minimum this involves switching a pointer in the kernel to a new process control block. It may also involve talking to the memory management hardware to switch to a new address space and deal with any effects on caches.

As the automobile service manuals say, "installation is the reverse of removal":

1. Copy the saved state of the new process to wherever it needs to go; switch the PCB pointer and update memory management if necessary.
2. Restore the stack pointer and starting popping: `POPAD`, `IRET` or somesuch.

You should be wondering here how `yield` knows which processes to switch to; we'll talk about this under *Scheduling* below. You should also notice that even if we don't implement any other separation between processes, we at minimum have to have separate stacks. This is the main thing that separates a multi-threaded process from a standard single-threaded one.

Note that these steps are generally very architecture-dependent. Some CPU architectures (including the x86) include features for **hardware context switching**, where state changes can be handled by switching a pointer built into the CPU. Since these features tends to be even more architecture dependent and also depend on the OS agreeing to use the CPU's ideas of what to put in a PCB, they tend not to be used much in OS's aiming for portability. But they are handy if you are coding for a single architecture (like our friends in Redmond or the builders of some embedded systems) and really care about performance.

The advantage of cooperative multitasking is that the implementation is not very hard and you can yield only when it is safe to do so. The disadvantage is that you have to remember to yield a lot, which can produce filibusters if the processes want to be rude. This not only causes trouble with this code snippet:

Toggle line numbers

```
1      while(1);
```

But it also annoys anybody using the same machine as this fine program:

Toggle line numbers

```
1  while(1) {
2      compute_wind_tunnel_simulation_without_yielding();
3      yield();
4  }
```

Here the on-screen mouse pointer is likely to move very jerkily (if at all), since if we can only switch contexts when yielding we don't get to run our window system very often. The solution is to interrupt these rude processes whether they like it or not.

## 20.2. Option 2: Pre-emptive multitasking

**Pre-emption** is when the kernel yields on behalf of a process rather than making the process do it. For the most part, this is a good thing: like running a garbage collector instead of `malloc` and `free` or using virtual memory instead of paging by hand, we are giving up a little bit of control to the underlying system in return for getting rid of a lot of headaches. Unfortunately in each of these cases we create new headaches. The particular new headache we get with pre-emption is that we have to take concurrency seriously.

### 20.2.1. Mechanism

Typically we still allow voluntary yielding, although this operation is usually hidden under some other **blocking** system call (e.g. `read` on data that isn't available yet). But we now also have incoming interrupts from that may produce a context switch between processes. The mechanism for switching out a process is now:

1. Interrupt triggers switch to ring 0 interrupt handler. The interrupt might be from an I/O device but much of the time will be triggered by a timer every fixed number of milliseconds.
2. Interrupt handler calls kernel-internal yield routine.
3. Kernel-internal routine saves process state as before and jumps to the scheduler.

The mechanism for switching a process back in is essentially unchanged.

### 20.2.2. Gotchas

With pre-emptive multitasking, some other process may sneak in and break things while I'm not looking. With separate address spaces (or just separate data structures) this is not a problem. In a threading model with a common address space, this can lead to all sorts of nasty inconsistencies. This is particularly likely to come up inside the kernel, where a common approach is to have one thread per user process sharing a common set of core data structures.

The simplest approach to preventing bad outcomes of unwanted concurrency is to prevent the concurrency, usually through some sort of **locking** or **critical section** mechanism. A lock marks a data structure as inaccessible to anybody but the lock-holder. This requires politeness, and in a reasonable implementation also requires interaction with the scheduler so that processes don't spin waiting for a lock to become available. A critical section enforces that no other thread takes the CPU while it is running. This only requires turning off interrupts (CLI on x86). We'll talk about how to use these and how not to use them later.

## 21. Scheduling

We've deferred the question of how to choose which process to switch to. This is generally the job of a **scheduler**, which might range from anywhere from a couple of lines of code inside `yield` in a simple kernel to a full-blown user-space process in a microkernel-based design. However we set up the scheduling *mechanism*, we can separately consider the issue of scheduling *policy*. Here the goal is to balance between (a) keeping the kernel simple and understandable, and (b) not annoying the user. We also have to keep track of which processes are **blocked** waiting for locks or slow I/O operations.

Here are some common policies:

#### Round-robin

Processes move to the end of a queue when pre-empted (or unblocked). The next process to run is whatever is at the head of the queue. This has the advantage of speed and simplicity, but it doesn't give much control over resource allocation.

#### Random

Instead of putting the runnable processes in queue, put them in a bag and draw the next one to run at random. Theoretically this is approximately equivalent to round-robin assuming your random number generator is reasonably good. OS implementers hate it since an unlucky process may starve.

#### Priorities

Use a priority queue. High-priority processes take precedence over lower-priority processes. Priorities can be assigned either by the user (e.g. the update-the-mouse-pointer-now-so-the-user-doesn't-get-annoyed process beats the wind tunnel simulator) or by the system based on past performance (CPU hogs that have a history of getting pre-empted get lower priority than processes that yield the CPU quickly). This is the strategy used in Linux, which even has a particularly fast (constant-time!) priority queue implementation.

#### Real-time scheduling

Here some processes have hard constraints on when they run next or how often they run (think airplane control systems). The scheduler runs an algorithm that enforces these constraints if possible.

#### Fair-share scheduling

Each user gets a fixed slice of the CPU that is further divided between the user's processes using some other scheduling mechanism.

#### Let the user do it

This is the common microkernel approach alluded to earlier. The kernel's scheduler policy is to always run the designated user-space *scheduler* process, which can delegate its control of the CPU to specific other processes.

There are trade-offs in choosing any of these policies between simplicity, efficiency, performance, predictability, and user control. Which factors are the most important depends on the application. A typical modern OS will provide some hybrid policy whose default has been observed to work for most applications but which can be overridden by the user when necessary.

An orthogonal issue of scheduling policy is the **quantum** or minimum time that we schedule a process for before pre-empting it. With a small quantum we pre-empt a lot (which costs time). With a big quantum the mouse pointer stops moving. Consumer OSs will typically use a fixed quantum of about 10ms, which is less than most humans will notice and comparable to disk-access delays. Real-time embedded OSs may use quanta measured in microseconds that vary depending on the real-time constraints on other processes. Again these are policy trade-offs that depend on the application.

---

CategoryOperatingSystemsNotes

## 22. Threads

A **thread** abstracts the normal flow of control in a program: the sequence of addresses that the IP passes through. In a multi-threaded program, we have the illusion of multiple flows of control corresponding to multiple IPs. Typically this is implemented using time-sharing, where a single IP is switched between different threads. But (usually with OS support) it may also be implemented using multiple CPU cores.

Usually when we talk about threads we are assuming multiple tasks in the same address space; a single thread isolated in its own address space generally goes by the name of a **process** (see Processes). Some systems (e.g. Linux) use **task** to refer to either a thread or a process, the relevant property being that each task has its own IP and its own stack.

## 23. Why use threads?

Multithreaded programs are notoriously difficult to write and debug, since having multiple threads running concurrently with unpredictable timing makes a program's behavior itself unpredictable unless the programmer is very careful. So why would we bother with threads? There are basically four reasons (SGG also give four, but they're a slightly different four):

1. You bought a machine with more than one CPU core, and you'd like to use all of them. If you can rewrite your program to perform non-interfering tasks in separate threads, your program will run faster.
2. Your program is naturally structured to allow parallel computation, and it's easiest to write it this way. There are two very common design patterns that lend themselves to parallel execution: producer-consumer pipelines (e.g. `tr A-Z a-z | tr -cs "a-z'" '\n' | sort | uniq -c | sort -nr`, a five-stage pipeline that finds the most common words in its input when run on Unix), and server programs (e.g. a web server handling multiple simultaneous requests).
3. You want to handle multiple streams of interaction at once that share common data, but you don't want one of them to slow down the others. For example, in your GUI you don't want to wait for the user to type a key before updating the clock, and conversely the user doesn't want to wait for some excessively-complicated web page to render before typing a key. (Also the web server above, which doesn't want to get stuck reading half an HTTP request.)
4. You wanted to spawn 10,000 processes, but since the difference between them was only a few kilobytes it made sense to share the common bits so your computer doesn't burn up. To a certain extent this can be handled with a good shared library implementation, but even exploiting shared libraries it's usually cheaper (both in time and memory) to spawn a thread than a whole new process.

## 24. Thread creation: user view

Typically the user will have some sort of thread library. On POSIX systems, the standard thread library is **Pthreads**. A typical call to create a thread in Pthreads looks like this:

Toggle line numbers

```
1 pthread_attr_init(&attributes);
2 pthread_create(&thread_id, &attributes, function_to_run, function_argument);
```

Here `thread_id` is a retval that returns a handle to the thread (which can be used to do further nasty things to it), `attributes` is a set of thread options (set to a default value by `pthread_attr_init`), and `function_to_run`, and `function_argument` are the standard C-style closure consisting of a function and a `void *` argument. Pthreads also provides procedures for waiting for threads to finish (`pthread_join`), and various other useful tools like mutexes.

In object-oriented languages, threads are often implemented by inheritance from some standard thread class (e.g. `Thread` in Java or `threading.Thread` in Python). Here a thread is created by creating a new object of the appropriate class and kicking it to life with some standard `run()` method.

For more examples see SGG §4.3.

## 25. User threads vs kernel threads

Processes are typically implemented as **kernel threads**—threads that are implemented in and supported directly by the kernel. We can also have **user threads**, which don't necessarily require kernel support. For example, it's not hard to hack up a primitive threading system within a user process pretty much any modern C environment by allocating some extra stacks using `malloc` and abusing `setjmp` and `longjmp` to switch between them (note that this will require some machine-dependent bits to get right). Similarly, Java threads in early JVM implementations ran entirely inside the Java interpreter with no interference from or knowledge of the underlying OS. But in many cases it is helpful to have OS support for user threads. There are three basic models:

1. **Many-to-one:** The simplest model as described above. A user-space thread library manages many user threads on top of a single kernel thread. The advantage is that it doesn't require kernel support. The disadvantage is that since the kernel doesn't know about

user threads it can't schedule them, at most one user thread can run at a time (since the thread library can't get at more than one CPU), and any action that blocks the single kernel thread (e.g. a blocking read system call) blocks all the user threads.

2. **One-to-one:** The kernel allocates one kernel thread for each user thread, e.g. using the `clone` system call in Linux. This gives essentially all the advantages might want from threads: better scheduling, concurrency across multiple CPU cores, no blocking between user threads. One slight disadvantage is that requiring kernel involvement increases costs, and may put a limit on how many user threads a process can have. Most widely-used OSs use this model.
3. **Many-to-many:** A process may have many user threads and many kernel threads, with user threads mapped to the pool of kernel threads according to various possible rules. This combines the good features of the previous two models at the cost of additional complexity.

Here's something you don't see: nested threads, where one user thread in a many-to-one model may itself be divided into further threads. The closest a standard OS will get to this is running a virtual machine, although some microkernels have the machinery to support this. Exercise: What would nested threads actually look like, and why would or wouldn't you want to support them?

## 26. Why not use threads?

A program that uses threads will (in general) be nondeterministic: its behavior will change from one execution to the next. This makes debugging using the standard approach of making small changes and seeing if they improve things much more difficult—how do you know if the program is really fixed or just decided to behave differently this time? It also causes trouble when two (or more) threads share data structures without using some sort of `ConcurrencyControl`.

---

CategoryOperatingSystemsNotes

## 27. ConcurrencyControl

**Concurrency control** is the process of preventing threads running in the same address space from interfering with each other.

## 28. The need for concurrency control

### 28.1. Example: Updating the web hits counter

Let's suppose we have a multi-threaded web server that keeps track of how many times it serves a particular page (since we get paid \$0.0002 per impression, say). Naively, we might try using the following code:

Toggle line numbers

```
1 void
2 update_hit_counter(void)
3 {
4     hit_counter = hit_counter + 1;
5 }
```

On a IA-32 machine, gcc (with no optimization flags) will turn this into

```
movl    hit_counter, %eax
addl    $1, %eax
movl    %eax, hit_counter
```

I.e., read `hit_counter`, add 1 to it, and write the result back.

Suppose we have two threads P and Q both trying to increment `hit_counter`, which starts at 0. If P is interrupted after reading `hit_counter` but before writing it back, P and Q will both read 0 from `hit_counter` and both write 1 back. We just lost \$0.0002!

### 28.2. Race conditions

Any situation where the order of operations between two threads affects the outcome of the computation is called a **race condition**. Race conditions create nondeterminism and are thus bad. We'd like to get rid of race conditions when we can.

## 29. Mutual exclusion

Suppose we can ensure that nobody else touches `hit_counter` during **critical section**: the 3 machine-language instructions it takes to update it. Then the `update_hit_counter` procedure will appear to execute instantaneously: it will be **atomic**. So how do we enforce this?

### 29.1. Naive approach

Toggle line numbers

```
1 void
2 update_hit_counter(void)
3 {
4     while(hit_counter_busy); /* wait for other process to finish */
5
6     hit_counter_busy = 1;
```

```

7
8     /* start of critical section */
9     hit_counter = hit_counter + 1;
10    /* end of critical section */
11
12    hit_counter_busy = 0;
13 }

```

Here we have an attempt at a **spin-lock**: a busy bit that warns the other threads that we are updating the hit counter. This particular implementation doesn't work:

1. P and Q both read 0 from `hit_counter_busy` and exit their loops.
2. P and Q both run the rest of the procedure together.

This does work if we can test if `hit_counter_busy` is 0 and set it to 1 as a single atomic operation (called a **test-and-set**). Some hardware provides this operation. But it is in fact possible to do mutual exclusion even without test-and-set.

## 29.2. Another naive approach

Toggle line numbers

```

1  /* For two threads 0 and 1 */
2  int thread_waiting[2]; /* initially zero */
3
4  void
5  update_hit_counter(int me)
6  {
7      /* try to grab the lock */
8      thread_waiting[me] = 1;
9
10     /* did I win? */
11     for(;;) {
12         if(thread_waiting[!me] == 0) break;
13     }
14
15     /* start of critical section */
16     hit_counter = hit_counter + 1;
17     /* end of critical section */
18
19     thread_waiting[me] = 0;
20 }

```

This does guarantee mutual exclusion. One of the threads sets `thread_waiting` first, and the second thread will read it and spin until the first thread finishes (since it does its read after setting `thread_waiting` for itself). Unfortunately it also allows for **deadlock**: if each thread sets its `thread_waiting` flag before looking at the other's, both with spin. So we are still losing.

## 29.3. Peterson's algorithm

Toggle line numbers

```

1  /* For two threads 0 and 1 */
2  int thread_waiting[2]; /* initially zero */
3  int turn; /* initially arbitrary */
4
5  void
6  update_hit_counter(int me)
7  {
8      /* try to grab the lock */
9      thread_waiting[me] = 1;
10     turn = me;
11
12     /* did I win? */
13     for(;;) {
14         if(thread_waiting[!me] == 0) break;
15         if(turn != me) break;
16     }
17
18     /* start of critical section */
19     hit_counter = hit_counter + 1;
20     /* end of critical section */
21
22     thread_waiting[me] = 0;
23 }

```

Intuition: We use the same argument as the previous solution to show that at most one thread wins the `thread_waiting` race and executes the first `break` statement. If neither thread wins the `thread_waiting` race, then only one thread is the first to write to `turn`. The first thread to write to `turn` escapes as soon as the second thread writes to `turn`. (Think of `turn` as saying whose turn it is to wait, not whose turn it is to go.) So we avoid the previous deadlock, since the winner eventually completes the critical section and sets its `thread_waiting` flag back to zero, letting the other thread out. (See MutualExclusion for a more detailed proof and some other algorithms.)

Problem: only works for two threads. To handle more threads, we need to build a tree of these, which gets very complex very fast, or use a different algorithm. Even with a smarter algorithm we provably need to allocate—and have each thread read—at least one memory location per thread that might access the mutex, which gets expensive.

## 29.4. Preventing pre-emption



On a uniprocessor (or inside a single process), we may be able to implement a critical section by temporarily preventing pre-emption.

Toggle line numbers

```
1 void
2 update_hit_counter(int me)
3 {
4     turn_off_interrupts();
5
6     /* start of critical section */
7     hit_counter = hit_counter + 1;
8     /* end of critical section */
9
10    turn_on_interrupts();
11 }
```

This is a pretty good strategy for threads inside the kernel, where we have direct access to e.g. CLI and STI instructions. It also works well with co-operative multitasking (just remember not to call `yield` inside the critical section). But it doesn't help with multiple processors, since our two threads might genuinely be running in parallel!

## 29.5. Hardware support for locking

For a multi-processor machine, we pretty much want some sort of hardware support, e.g. the `xchg` operation in IA-32, which performs a hardware-level lock on the shared-memory bus and then exchanges a value with memory, effectively acting as a test-and-set.

Usually the details of the particular operation are hidden under some higher-level abstraction, such as a **lock** (also called a **mutex**). A lock is a data structure that supports **acquire** and **release** operations, with the rule that at most one process can successfully complete an acquire operation before somebody executes release. Further abstractions can then be built on top of the lock.

## 30. Spinlocks

A **spinlock** is a lock where blocked processes "busy wait," checking the lock over and over again until it becomes free. The selling point for a spinlock is that the code can be made very simple and the cost of acquiring and releasing the lock is very very small, at most a few memory-bus clock cycles. The disadvantage is that a processor spinning on a lock doesn't do anything else while it's waiting.

An example of an implementation of a spinlock for x86 using the `xchg` operation can be found at [Wikipedia: Spinlock](#) (warning: Intel assembler syntax).

One way to get around the problem of busy-waiting is to have the waiting process sleep if it can't acquire the lock quickly, as in the following code, which assumes a (hardware-dependent) `test_and_set` procedure defined elsewhere.

Toggle line numbers

```
1 /* atomically write a new value to location and return the old value */
2 extern int test_and_set(int *location, int value);
3
4 #define LOCK_TRY_TIMES (1024)
5
6 void
7 acquire_lock(int *lock)
8 {
9     int count;
10
11     for(;;) {
12         for(count = LOCK_TRY_TIMES; count > 0; count--) {
13             if(test_and_set(lock, 1) == 0) {
14                 /* got it */
15                 return;
16             }
17         }
18
19         /* didn't get it, go to sleep */
20         yield();
21     }
22 }
23
24 void
25 release_lock(int *lock)
26 {
27     test_and_set(lock, 0);
28 }
```


The value of `LOCK_TRY_TIMES` should be set to minimize the total cost of (1) waiting too long before giving up, and (2) the context switch involved in calling `yield`. If nobody else is waiting for the lock, then it doesn't matter what we set it to, so the question is how long we expect to wait when there is high **contention** (many processors trying to acquire the lock at once). A reasonable strategy if we are just trying to minimize overhead might be to set `LOCK_TRY_TIMES` so that the waiting time is roughly equal to the time to perform two thread context switches (since we will have to swap the thread back in later); this way we pay at most twice the overhead that we would pay if we knew to swap out immediately. This assumes that we mostly care about minimizing overhead and not other factors (like proceeding as quickly as possible once the lock becomes available). If we want to acquire the lock as soon as possible we might be willing to set `LOCK_TRY_TIMES` higher and accept the busy-waiting overhead. At the other extreme, trying the lock just once minimizes busy-waiting, but if nobody holds the lock for very long we may be better off waiting for it.

Another limitation of this simple spinlock is that it doesn't involve the thread scheduler beyond calling `yield`. So it may be that when the scheduler swaps us back in, we still can't get the lock. This is bad not only because we may waste time checking, but also because we may



miss a chance to get in because the scheduler didn't know to schedule us while the lock was free and somebody else snaffled it up while we were waiting. Such a situation is called **starvation**.

## 31. Semaphores

**Semaphores** were invented by Wikipedia: Edsger\_Dijkstra precisely to deal with the problem of waiting for bounded supplies of resources. The basic idea of a semaphore is to provide a counter with two operations *wait* and *signal* (called *P* and *V* in  Dijkstra's original paper, for *proberen* = test and *verhogen* = increment in Dutch), where *wait* waits for the counter value to be positive and then decrements it and *signal* increments the counter.

The simple version of the code looks like this:

Toggle line numbers

```
1 void
2 wait(int *sem, int *lock)
3 {
4     for(;;) {
5         acquire_lock(lock);
6         if(*sem > 0) {
7             sem--;
8             release_lock(lock);
9             return;
10        }
11        release_lock(lock);
12    }
13 }
14
15 void
16 signal(int *sem, int *lock)
17 {
18     acquire_lock(lock);
19     sem++;
20     release_lock(lock);
21 }
```

Note the use of a mutex to protect the semaphore from lost updates. With additional hardware support (e.g. built-in atomic semaphore operations) it may be possible to avoid this.

### 31.1. Applications

We can use semaphores to protect a resource or collection of resources. A semaphore that is initially set to 1 acts like a lock: only one thread can acquire it at a time. A semaphore initially set to a higher value will allow multiple threads in simultaneously. We can also use `signal` to indicate that some resource is available; for example, if we have a group of worker threads waiting for things to do (e.g., kernel threads waiting to run user threads in a many-to-many thread model), we can have them all execute `wait` on a semaphore that is initially zero and use `signal` to indicate that some new job has shown up. Similarly in a producer-consumer model we can use `signal` whenever the producer adds a new item to the buffer and `wait` when the consumer is ready to remove one; this blocks the consumer when there is nothing to do. (With a second semaphore that tracks empty space in the buffer we can also block the producer when the buffer is full.)

### 31.2. Implementation

Semaphores are integrated with scheduling by including a **wait queue** along with the counter value. When a process sees a zero or negative value in the semaphore, instead of spinning it adds itself to the end of the wait queue and blocks (informs the scheduler that it is no longer ready to run). When a process increments the counter, it will wake a blocked process at the front of the queue. A standard optimization is to use negative values in the semaphore counter to indicate the number of processes waiting; this avoids having to test separately if the queue is empty or not.

The code might look something like this:

```
typedef struct semaphore {
    int value;
    ProcessQueue *p; /* queue of processes, implemented elsewhere */
}

extern int GlobalSchedulerLock;

void
wait(Process *p, Semaphore *s)
{
    acquire_lock(GlobalSchedulerLock);

    /* see if we can get in */
    s->value--;

    if(s->value < 0) {
        enqueue(s->q, current_process);
        block(p); /* mark process p as unschedulable */
    }

    release_lock(GlobalSchedulerLock);
}

void
signal(Semaphore *s)
{
    acquire_lock(GlobalSchedulerLock);
```

```

s->value++;

if(s->value <= 0) {
    wake(dequeue(s->q));    /* mark process as runnable */
}

release_lock(GlobalSchedulerLock);
}

```

Both operations must be implemented atomically—including changes to the scheduler state. The wait operation must also be implemented so that the blocked process isn't the one responsible for releasing the scheduler lock. Typically this involves implementing semaphores as system calls.

One decision that will affect how processes interact with semaphores is how we implement the wait list. The fastest and simplest approach is to implement it as a stack (since linked lists like pushing and popping at the front). This, however, may lead to starvation if some thread gets pushed to the end and stays there because other threads grab the semaphore before it rises up. Implementing it as a queue guarantees no starvation (assuming no deadlock), so may be a better choice if fairness is an issue. But the definition of a semaphore is agnostic about which thread wakes up in response to a `signal`, so unless you know that the semaphore you are using guarantees fairness, you can't count on getting it.

On POSIX machines semaphores are available using `sem_init`, `sem_wait`, `sem_post` (signal), etc.

## 32. Monitors

Both mutexes and semaphores behave badly if mistreated. To prevent foolish programmers from failing to release locks, from releasing them more than once (if implemented as semaphores), or from attempting to reacquire a lock that they already have, many program languages provide explicit **monitors**, a language construct that indicates that some block of code should be executed only while in possession of a particular lock. An example are **synchronized methods** and **synchronized statements** in Java. For example, in Java our hit counter could use a synchronized method like this:

```

public synchronized void update_hit_counter() {
    hits++;
}

```

and the run-time would take care of locking the object to avoid trouble.

We could also synchronize on an explicit lock object:

```

public void update_hit_counter() {
    do_something_expensive_where_we_do_not_need_to_lock();

    synchronized(this) {
        hits++;
    }
}

```

Note the use of `this` as a lock; in Java all objects have a lock attached to them to support synchronized methods, which is a reasonable default to choose unless there is some reason to use something else.

One advantage of monitors is that they allow the system to choose whatever locking mechanism is most efficient (or even compile out locks if it can prove that they are not needed).

## 33. Condition variables

A weaker relative of a semaphore is a **condition variable**, which acts like the wait list in a semaphore without the counter. When executing `signal` on a condition variable, exactly one waiting process is unblocked. If no processes are waiting, nothing happens.

One thing to watch out for with condition variables is a missed connection, where process P starts waiting for a condition just after process Q signals it, leading to deadlock. The usual way to avoid this is to tie a condition variable to a mutex, so that P unlocks the mutex and waits on the condition variable as a single atomic operation, and Q locks the mutex before executing the signal (and unlocks it afterwards). Most implementations of condition variables (e.g. `pthread_cond_wait`) provide such an atomic unlock-and-wait operation, which is impossible to implement directly from a separate mutex and simple condition variable since there is otherwise a gap between the unlock and the wait that allows the signal to be missed. (The other order doesn't work because the thread blocks on the wait and never gets to the unlock.)

## 34. Deadlock detection and avoidance

See Deadlock.

---

CategoryOperatingSystemsNotes

## 35. Deadlock

Notes on deadlock for CS422. For more details see SilberschatzGalvinGagne Chapter 7.

## 36. What is deadlock?

**Deadlock** is when two or more tasks never make progress because each is waiting for some resource held by another process.

### 36.1. Example: too little memory

Two processes each demand 1.5 Gb of memory on a machine with only 2 Gb (and no virtual memory). The operating system kindly gives each 1 Gb exactly. Neither process can make progress until the other gives up some of its memory.

### 36.2. Example: I/O

Two interactive processes on a primitive handheld device with no window system each want to control the keyboard (so the user can type at them) and the display (so they can respond). If process P grabs the keyboard while process Q grabs the display, both may be stuck waiting for the other resource.

### 36.3. Example: bidirectional pipe

A common design pattern in Unix-derived operating systems is the pipeline, e.g. `sort | uniq | wc` to count unique lines in a file. In a pipeline, each program feeds its output to the next program. Curiously, there is no built-in mechanism in most shells to create a circular pipeline, where two programs A and B each produce as output the input to the other, even though it may be reasonable in some circumstances for one program to use another as a subroutine, providing its input and consuming its output. Even scripting languages that provide such tools (e.g. Python's `os.popen2` function) bury them in obscure libraries. Why is this?


Let's imagine a simple circular pipeline where process P (say some user program) sends data to process Q (say `tr`) and then reads the result. We'll imagine each process puts its outgoing data in a buffer guarded by a semaphore, and blocks when the buffer fills up. What happens?

If P's input to Q is large enough and Q produces output comparable in size to its input, then Q fills up its outgoing buffer and blocks before P is done writing. This causes P to fill up its output buffer and block. Nobody makes any progress from this point on—we have a deadlock.

## 37. Processes and resources

Deadlocks are described in terms of **processes** (things that can block) and **resources** (things processes can wait for). Processes may or may not correspond to full-blown processes as used elsewhere. It is typically assumed that each resource may have multiple **instances**, where a process is indifferent to which instance it gets and nobody blocks unless the processes collectively request more instances than the resource can provide.

## 38. Necessary conditions

There are four conditions which must hold for deadlock to occur as classically defined. These are known as *Coffman's conditions* from a 1971 survey paper of which Coffman was the first author in alphabetical order (Coffman, E.G., M.J. Elphick, and A. Shoshani, System Deadlocks, ACM Computing Surveys, 3(2):67–78, 1971;  [coffman\\_deadlocks.pdf](#)).

1. **Mutual exclusion.** There must be some resource that can't be shared between processes.
2. **Hold and wait.** Some process must be holding one resource while waiting for another.
3. **No preemption.** Only the process holding a resource can release it.
4. **Circular wait.** There must be some cycle of waiting processes  $P_1, P_2, \dots, P_n$  such that each process  $P_i$  is waiting for a resource held by the next process in the cycle. (Note that this implies hold-and-wait.)

## 39. Resource-allocation graphs

A **resource-allocation graph** depicts which processes are waiting for or holding each resource. Each node in the graph represents either a process or a resource. A directed edge is drawn from process P to resource R if P is waiting for R, and from R to P if P holds R. (See SGG §7.2 for many pictures of resource-allocation graphs.)

The situation where  $P_1$  waits for a resource R held by  $P_2$  corresponds to a path of length 2 in the resource-allocation graph. Circular waiting strings these length-2 paths together into a cycle. It follows that deadlock can occur only if there is a cycle in the resource-allocation graph. The converse is generally not true (although it holds if there is only one instance of each resource). For example, if  $P_1$  holds  $R_1$  and waits for  $R_2$  while  $P_2$  holds  $R_2$  and waits for  $R_1$ , then we have a cycle in the resource-allocation graph, but there is no deadlock if  $R_2$  has multiple instances some of which are held by other processes not involved in the cycle. When these extra instances become available,  $P_1$  stops waiting and the cycle clears up.

## 40. Preventing deadlock

To prevent deadlock, we just have to violate one of the Coffman conditions.

1. **No mutual exclusion.** If there's no need for mutual exclusion, there's no deadlock. This is the best solution when it can be arranged, particularly when resources (read-only files, lock-free data structures) can be shared. This doesn't work for resources that can't reasonably be shared by two processes at the same time (most writable data structures, the CPU, CD-ROM burners, the keyboard). Sometimes resources can be partitioned to avoid mutual exclusion (memory partitioning, window systems).
2. **No hold-and-wait.** Adopt a policy of not letting a process wait for one resource while holding another, either by requiring each process to hold only one resource at a time, or to request all of the resources it needs simultaneously. The first approach may be

severely limiting (for example, an interactive program can't get exclusive access to the keyboard and the display at the same time). The second has two problems: it requires a process to predict what resources it needs in advance, and it may allow a process to starve waiting for a group of resources that never become free all at the same time. Sometimes resources can be consolidated to allow the single-resource approach: it's quite natural, for example, to have a single mutex that covers all three of the keyboard, mouse, and display. In the limit we can lock the entire system (caveat: difficult for distributed systems) in order to lock any resource, but then things are likely to get slow.

3. **Allow preemption.** This is almost as good as eliminating mutual exclusion: if I can bump some process off a resource it is hogging, then I can break a deadlock cycle. The CPU is the primary preemptible resource (although we may get deadlock if we allow processes to block preemption while waiting for other resources—as can happen with disabled interrupts or in cases of **priority inversion** where a high-priority process pushes the low-priority process it's waiting for off the CPU). Another preemptible resource is memory (assuming we can swap out to disk). The difference between preemption and sharing is that in the preemption case we just need to be able to restore the state of the resource for the preempted process rather than letting it in at the same time as the preemptor.
4. **Eliminate cycles.** This is similar to no-hold-and-wait. We'll require processes to grab resources in increasing order according to some total order (one common heuristic is increasing order by the memory address of the mutexes guarding the resources). If process P is holding  $R_1$  while waiting for  $R_2$ , then  $R_2 > R_1$  in the ordering (otherwise P would have grabbed it first). So we can't have a circular wait, because this would give a cycle in the total order on resources. Note that unlike hold and wait we don't get starvation if the resources are allocated fairly: I will eventually come to the front of the line for the next resource on my list and will thus eventually get all the resources. The disadvantage though is that I still need to be able to predict what resources I want in advance or accept that I may not be able to ask for some resource if I was unlucky enough to request some other resource first.

## 41. Avoiding deadlock

Here we aren't going to violate the Coffman conditions, but we may make processes wait for a resource *even if it is available* because granting the resource might lead to deadlock later. Typically this requires that processes declare in advance what resources they might request.

For example, in the keyboard-and-display situation, each process could declare that it may want at some point 1 keyboard and 1 display. Then once process P grabs the keyboard, a deadlock-avoidance algorithm could make Q wait for the display even if P hasn't asked for it.

Formally, this condition is expressed in terms of **safe states**. Intuitively, a state is *safe* if there is some way to let all the processes run without creating deadlock. Formally, a state is safe if there is a **safe sequence** or ordering of the processes  $\{P_i\}$  such that each process  $P_i$  can satisfy its maximum demands only using (a) resources that are currently available plus (b) resource held by processes earlier in the sequence. (The idea is that the safe sequence describes the order in which processes will finish and release their resources, and once all previous processes have finished,  $P_i$  can get whatever it needs and finish too.)

In the keyboard-and-display example, the state where P holds the keyboard is safe using the safe sequence  $P < Q$ , since Q can get everything it needs once P finishes. But the state where P holds the keyboard and Q holds the display is not, since if  $P < Q$  then P can't get its maximum resources. A working deadlock-avoidance algorithm will notice this and keep Q from acquiring the display.

### 41.1. The Banker's algorithm

Due to Wikipedia: Edsger Dijkstra (yes, Dijkstra again), the Banker's Algorithm detects safe states. The method is the following: we write down a matrix whose rows correspond to processes and columns to resources, and put each process's maximum remaining demand in its row. We keep track of a separate vector A of available resources. To determine if a state is safe:

1. Look for a row that is componentwise less than or equal to A. If there is none, the state is not safe.
2. Otherwise, pretend that the process for that row has finished, remove its row from the matrix, and give all of its resources back to A.
3. Repeat until we get stuck or all processes are removed.

(This summary is taken from TanenbaumBook §3.5.4. SilberschatzGalvinGagne §7.5.3 gives more details.)

It's not hard to see that the Banker's Algorithm computes a safe sequence. More work is needed to show that it always finds a safe sequence if one exists. Tanenbaum observes that in practice nobody uses this algorithm given the unreasonable requirement that each process pre-specify its maximum demand.

## 42. Dealing with deadlock

There are several ways to detect and deal with deadlock. In increasing order of complexity:

1. **Do nothing.** If the system locks up, that will teach the user not to try doing what they just did again. This is approach taken in practice for many resource constraints.
2. **Kill processes.** We can detect deadlock by looking for waiting cycles (which can be done retrospectively once we notice nobody is making as much progress as we like). Having done so, we can either kill every process on the cycle if we are feeling particularly bloodthirsty or kill one at a time until the cycle evaporates. In either case we need to be able to reset whatever resources the processes are holding to some sane state (which is a weaker than full preemption since we don't care if we can restore the previous state for the now-defunct process that used to be holding it). This is another reason why we design operating systems to be rebooted.
3. **Preempt and rollback.** Like the previous approach, but instead of killing a process restore it to some earlier safe state where it doesn't hold the resource. This requires some sort of checkpointing or transaction mechanism, and again requires the ability to preempt resources.

A lot of research has been done on deadlock recovery in the database literature, where transactions (blocks of code that lock everything they touch but that are set up to be rolled back if preempted) provide a basic tool allowing the preempt-and-rollback strategy. Some of this work has started appearing in OS research, e.g. in Wikipedia: Software\_transactional\_memory.

## 43. ProcessorScheduling

Some notes on processor scheduling. For more details see SilberschatzGalvinGagne Chapter 5.

## 44. Processor scheduling: basics

Recall the basic picture for processes: We have various tasks (processes or threads) waiting in queues (which may not enforce strict FIFO ordering) for various resources. One of these resources that pretty much every task needs is the CPU. We'd like to arrange access to the CPU to take advantage of the particular needs of particular processes and minimize the noticeable waiting time and general sluggishness of our system. The main mechanism for doing this is a **short-term scheduler** or **CPU scheduler** that choose which process to pull off the ready queue when the CPU becomes idle. More sophisticated schedulers may also include a **long-term scheduler** that tracks the behavior of processes and adjusts their parameters to influence the behavior of the short-term scheduler, or (in batch systems) may even plot out CPU assignments well into the future. We'll mostly be talking about short-term scheduling.

Note that some of the issues in CPU scheduling might apply to other scarce resources, e.g. traffic shaping on a network. Conversely, if you have a lot of CPUs or few processes, CPU scheduling becomes much less of an issue since the resource isn't scarce.

### 44.1. CPU burst cycle

Processes typically alternative **CPU bursts** with I/O bursts. During the CPU burst the process needs the CPU; during the I/O burst the process doesn't, because it's waiting for some slow device. Through observation and experiment, OS researches have determined that the length of CPU bursts measured across all processes is distributed approximately exponentially, with short bursts much more likely than long bursts. This pattern is somewhat true even of single processes (think of a program that has to do several I/O operations in a row after a long computation), but the distribution may be shifted depending on whether the process is **CPU bound** (mostly wanting to use the CPU for long computations) or **I/O bound** (mostly waiting for I/O).

### 44.2. Types of jobs

#### System processes

Core system services. These typically have to be scheduled quickly, or the system keels over dead. Behavior tends to be predictable since they ship with the OS.

#### Real-time processes

Have hard real-time constraints (e.g. turn off the X-ray gun after at most 10 ms).

#### Interactive processes

User notices if they are slow: window system, word processors, web servers.

#### Batch processes

User doesn't notice if they are slow as long as they finish eventually and don't get in the way: background animation rendering, anti-virus scanning.

Goal is to take advantage of the characteristics of these jobs to maximize the right criterion for each.

### 44.3. Performance measures

(See SGG §5.2.)

- CPU utilization: what percentage of time the CPU is not idle. A good scheduler keeps the CPU busy. In particular this means that we want to space processes out so they aren't all trying to do I/O at once.
- Throughput: how many processes are completed per time unit. (Depends on what the processes are doing).
- Turnaround time: how long it takes a particular process to finish.
- Waiting time: total time spent by all processes in the ready queue.
- Response time: time between user input and corresponding system action.

Response time likely will be the sole criterion for real-time processes and the main criterion for interactive processes (and possibly system processes that they depend on). Batch processes are likely to care more about turnaround time. The other measures (which tend to be closely related) are mostly useful to let you know if your scheduler is screwing up somehow, but waiting time can vary dramatically depending on scheduling policy and is the factor that most closely affects response time.

There is a relationship between average queue length  $n$ , average arrival time  $\lambda$  (in processes/second), and average waiting time for a *single* process  $W$  (in seconds):  $n = \lambda W$ . This allows computing any of these quantities from the other two. Total waiting time will be  $nW = \lambda W^2$ ; so total waiting time grows faster than individual process waiting time. This is another reason why waiting time is so important.

Again, if the CPU is free most of the time, it doesn't really matter what scheduling method we use. The goal is to get **graceful degradation** in performance when the system is loaded.

## 45. When scheduling happens

To a first approximation, we schedule at every context switch. Following SGG §5.1.3, we can observe four places where this might happen, as a function of a change in state of some process:

1. Running  $\rightarrow$  waiting, because of an I/O or mutex request, waiting on some condition, or explicitly calling `yield`.
2. Running  $\rightarrow$  ready. (Preemption.)
3. Waiting  $\rightarrow$  ready. Resource becomes available.
4. Termination.

The first and last cases involve a process giving up the CPU, so we have to schedule some other process. A **cooperative** or **non-preemptive** scheduler will run only in these cases. A **preemptive** scheduler may be called in the other cases as well.

If we are doing scheduling often, we want our scheduler to be cheap (possibly even running inside the kernel thread that is being preempted, so we avoid an extra context switch). At the same time, we want to use a good scheduling policy to minimize waiting time (and thus response time for interactive processes). So there is a trade-off between scheduling algorithm complexity and performance. Since performance depends on more unpredictable factors than complexity, this tends to favor simpler algorithms.

## 46. Algorithms

### 46.1. First-come first-served

The ready queue is a queue; new processes go to the end of the line. No preemption. Main advantage: simplicity.

Disadvantage: High waiting times if processes vary in CPU burst length. Consider processes with CPU bursts of 100, 1, 1, 1, 1, and 1 milliseconds. If the first process is the 100-ms one, we get 510 ms of total waiting time as compared to 15 ms if we sort in the other order.

### 46.2. Round robin

Basically FCFS with preemption. After using up its **quantum**, a process is bumped from the CPU whether it is done or not. Waiting time for an individual process now looks roughly like  $\text{ceiling}(\text{length of CPU burst} / ((\text{quantum}) - (\text{context switch time})) * (\text{quantum}) * (\text{average number of waiting processes}))$ ; a quick way to read this is that a process waits for roughly  $(n-1)/n$  of its run time where  $n$  is the size of the ready queue.

Adjusting the quantum is important: With a big quantum, we get poor response time. With a small quantum, we spend a lot of time on unnecessary context switches. A common heuristic is to aim for most CPU bursts to be shorter than the quantum, so that most processes are not preempted.

### 46.3. Shortest-job-first

If we know the burst lengths, we can sort jobs to minimize waiting time as in the FCFS example above. If we don't know the burst lengths, we can guess based on averaging previous burst lengths. Typical approach is to use an exponentially-declining weighted average, which is computed as  $(\text{guess}) = (\text{previous guess}) * \alpha + (\text{last observation}) * (1-\alpha)$  for some constant  $\alpha$ .

This still has the problem that once a big job gets the CPU any jobs that arrive after it will have to wait.

### 46.4. Priority scheduling

Generalizes SJF by allowing for arbitrary priorities other than burst length. The highest-priority job gets the CPU whenever it becomes free, or in a preemption model whenever it is being used by a lower priority job.

Disadvantages: Low-priority jobs may starve (solution: slowly increase the priority of starving jobs). A high-priority job waiting for a low-priority job to do something may have to wait a long time, a situation called **priority inversion** (solution: temporarily bump up the priority of the job being waited for). General problem: Priority queues are an expensive data structure.

For real-time scheduling, **earliest deadline first** is a common priority-based scheme that guarantees completion of all jobs by their deadlines if possible.

### 46.5. Multilevel queue scheduling

Partition the jobs into classes based on their scheduling characteristics and schedule each class separately off of a separate ready queue. So for example we can schedule real-time jobs using earliest-deadline-first, system jobs using priorities, interactive jobs using round-robin, and batch jobs using first-come first-served. We still have to decide how to allocate the CPU between the queues; here typically a mix of priority-based scheduling (real-time jobs always take priority) and timeslicing (10% of the CPU goes to batch jobs) works.

### 46.6. Multilevel feedback-queue scheduling

Like MQS, but assign processes to classes based on their past behavior. So processes with short CPU bursts get put on a queue with a smaller quantum, while processes that never give up the CPU are eventually classified as batch jobs (since they don't do any I/O, nobody will notice!) Hideously complex in its full glory.

## 47. Multiple processors

Ideal case: one processor per process. No scheduling!

More typical case: small number of processors, many processes. Two basic approaches: Centralized scheduling (**asymmetric multiprocessing**) or distributed scheduling (**symmetric multiprocessing**). Latter case does scheduling on a per-processor basis with long-term load-balancing to move jobs from heavily-loaded machines to lightly-loaded machines, either by having the heavily-loaded processors shove jobs away or having the lightly-loaded ones engage in **work stealing** (more efficient, since the lightly-loaded processors have time on their hands to look for new jobs).

Load balancing must balance two conflicting goals:

#### Processor affinity

It's best to keep a job on the same processor as much as possible, since that's where all its cached data sits.

## Load balancing

If we never move anybody, we can't fix imbalances.

A typical solution is to delay load balancing until it's clear that the long-term payoff from adjusting the load exceeds the cost of moving a process. We can also allow processes to lock themselves to particular CPUs if they know better than we do what they are doing.

# 48. Algorithms in practice

See SGG §5.6. Short version: Solaris uses multilevel queue scheduling with distinct scheduling methods for each class; Windows XP uses priority scheduling with a base priority class and "relative priority" adjustments that penalize processes that don't give up the CPU before the end of their quantum; Linux uses static priorities for real-time tasks and dynamic priorities for most tasks.

# 49. Evaluation

Theoretical: use queuing theory. Practical: simulate first, then implement what seems to work. See SGG for more details.

---

CategoryOperatingSystemsNotes

# 50. InterProcessCommunication

# 51. Motivation

In the beginning, processes generally didn't talk to each other except occasionally by leaving notes in the filesystem. But faster methods were needed to get real-time cooperation between processes.

Most interactions between processes fall into a small number of patterns:

## Shared state

A database, the Windows registry, the filesystem, the kernel scheduler data.

## Producer-consumer

One process generates a sequence of data that is consumed by another process.

## Worker threads

Several processes carry out tasks in parallel; requires a task queue and possibly some shared state the threads are operating on.

We want to provide tools for letting processes communicate with each other quickly to carry out tasks with these sort of structures.

# 52. Shared memory

The simplest form of IPC is shared memory. Here the main issue is ConcurrencyControl (which we've already talked about) and possibly MemoryManagement to map the same physical memory to two different address spaces (which we haven't talked about yet).

Advantage: Very fast.

Disadvantages: Requires careful locking to avoid trouble. Doesn't work across multiple machines. Doesn't (by itself) solve the producer-consumer problem.

# 53. Message passing

Here the model is that one process sends a message (some block of bits) and another process receives it, with actual delivery the responsibility of the operating system. The main advantage of message-passing is that it scales to multiple machines, is agnostic about the actual mechanism of delivery (see, for example, <http://www.ietf.org/rfc/rfc1149.txt>, the IETF standard for IP via Carrier Pigeon).

## 53.1. Interface

### 53.1.1. Channels and naming

One issue is how to specify where you are sending a message to and where you want to receive a message from. This is particularly tricky if you start out without knowing any friends.

A distinction can be made between **direct delivery**—sending messages to a specific process—and **indirect delivery**—sending messages to some channel without necessarily knowing what process might be on the other end. The latter is generally more useful, since it decouples a service (processing particular messages) from its implementation.

Some of the strategies used in various kinds of systems:

## Unix pipes

A pipe is an anonymous channel for shipping bytes. You create a pipe with the `pipe` system call, which gives you two file descriptors that you can then use standard POSIX I/O operations on. The recipient of a message is whoever is reading from the output end of the pipe. Since there is no way to hand file descriptors to an arbitrary process in most versions of Unix, pipe handles are usually inherited during a `fork`.

## TCP/IP

An address is given by an IP address (32 bits, or 128 bits for IPv6) and a port number (16 bits). IP addresses may be obtained from more human-readable names using DNS. Many port numbers are standardized, e.g. 21 for telnet, 22 for ssh, 25 for SMTP, 80 for HTTP, 6000 for X11. (Try running `nmap localhost` on the Zoo sometime.) The recipient of a message is whatever process asked the kernel to open that port for it. Sun RPC: Layered on top of TCP/IP. Port numbers are assigned dynamically by a portmapper service (which itself is at a standard port). Peer-to-peer systems: Implement a distributed lookup service to translate queries to IP addresses.

Even once you have a name, there is still the issue of setting up a channel or connection and what the behavior of that channel is. Some variants:

- Can anybody send to a channel or just a particular sender? In the former case, receiving a message should report who sent it.
- Can more than one process receive from a channel? (A channel that can be received from by multiple recipients is often called a *mailbox*.)
- Are messages clearly divided or do you just get a stream of bytes? The latter strategy is what you mostly get from Unix pipes and the TCP parts of the BSD network stack (which is used almost everywhere). The advantage from the point of view of the transport medium is both reduced complexity and the ability to package messages together or split them apart (since it doesn't know where the boundaries are anyway). The disadvantage for the user is that they have to separate messages by hand (but this can be hidden by libraries, or by using protocols like UDP that are message-oriented instead of connection-oriented).
- Is communication synchronous (sender blocks until receiver receives) or asynchronous (message gets there eventually).
- What kind of buffering (if any) does the channel provide?
- What happens if messages are lost in transit?
- What happens if one of the participants in a channel dies?

### 53.1.2. Sending a message

The interface for sending a message can be pretty straightforward: something like `send(msg, length, recipient)`, where `recipient` is the name of a channel. From the sender's perspective, the message is gone and it can go do whatever it likes.

### 53.1.3. Receiving a message

This is trickier. Since the recipient doesn't control when a message is sent (or how long it takes to arrive), we need to somehow get its attention. There are two basic approaches:

#### Event handler

Recipient registers a callback function with the kernel. When a message comes in, the kernel interrupts the recipient and runs the callback function.

#### Event loop

Recipient checks its messages from time to time using either a blocking `receive` function or a non-blocking polling function.

The advantages and disadvantages are similar to those for preemptive vs non-preemptive scheduling. Messages coming in through an event loop are tidier but less timely than messages delivered through interruptions.

Note we can always simulate the event-loop model with an event handler by having the event handler store incoming messages in a buffer. Conversely, a multi-threaded processes can simulate an event handler by having a *listener* thread whose only purpose is to wait for incoming messages.

## 53.2. Implementation

When we look at implementation, we suddenly realize that message-passing doesn't solve the producer-consumer problem, it instantiates it.

### 53.2.1. Using shared memory

For asynchronous message-passing, we need to allocate buffer space somewhere in the kernel for each channel. A send operation is now handled by a system call that (a) checks if the buffer is full and blocks otherwise (e.g., using a semaphore), (b) then writes the message to the end of the buffer. Receive is the reverse, blocking first if the buffer is empty (a second semaphore), and then reading from the start of the buffer. The start and end of the buffer can be maintained by keeping track of two pointers that wrap around at the end, giving a **ring buffer** architecture. Another option that allows for unbounded buffering subject to the memory limits and patience of the kernel is a dynamically-allocated queue.

For synchronous message-passing, the buffer can be much smaller since it only has to hold one message in transit. Here the main implementation issue is how a `send` operation interacts with the scheduler. Since we are going to block the sender anyway, do we just queue it up or should we do something like give up the rest of its time-slice to the recipient to get a fast response? Further optimizations may be possible if we can immediately switch to the recipient, like passing very small messages in registers or very big messages by mapping physical memory between address spaces. These optimizations might be especially important when message-passing is a critical system bottleneck, as in microkernels.

### 53.2.2. Across a network

Typical approach: Buffer on sender's machine, and then separate kernel thread and/or network driver pulls data out of buffer and sends it out through the network hardware. Receiver has matching structure, with incoming data being stuffed in buffers for later delivery.

Many details to work out: acknowledgments, retransmission, checksums, etc. Not our problem.

## 53.3. Exceptions

Bad things can happen. Suppose recipient dies, how do we notify sender?



### Unix approach

SIGPIPE delivered to signal handler (or if not caught, sender dies with Broken Pipe error), errors on write operations.

### TCP approach

Special acknowledgment message says connection is closed, kernel translates to appropriate errors.

How about lost messages?

### Retransmit

Kernel takes responsibility for resending message until it gets through (and acknowledged).

### Ignore it

Sometimes messages just get lost, too bad.

## 54. Remote procedure calls

Allows process P to call some specific procedure in process Q (generally process Q has to register the procedure as available) and gets result. Process Q may or may not be on the same machine.

Implementation is basically two messages: call and return. Arguments to call are translated into standard form (*marshaled*) for transmission. Process P generally blocks while waiting for the return.

Simplifies interface since we already understand procedure calls. But blocking P may be expensive (can be dealt with by using multiple threads in P).

Example: HTTP GET.

## 55. Remote method invocation

Fancy-pants Java RPC. Problem with RPC for OO languages is that arguments might be objects. RPC would send a bitwise copy of the object, but complex objects (particularly objects with state) can't be marshaled. So instead we replace each object in P that is included as an argument with a *delegate*, a new object that lives on Q but forwards any method invocations back to its progenitor on P (via RMI again, now running in reverse). Note that allowing these reverse calls means that P can't block completely during an RMI or we'll get deadlock. This is handled by having each process have a *listener thread* that processes incoming RMI requests, spawning *worker threads* to carry out the actual operations.

## 56. Efficiency issues

Any form of IPC can be expensive, since dealing with multiple processes probably involves both context switches and copying data. One way to deal with the cost is bundling: Many messages or procedure calls can be packed together so that the overhead is amortized. This is done, for example, by the X11 protocol and by the HTTP KeepAlive mechanism. But this usually requires user cooperation or at the minimum asynchronous communication (so the kernel can bundle the messages for the user).

---

CategoryOperatingSystemsNotes

## 57. MemoryManagement

Basic problem: Want to place programs and data in physical memory so that we can find them again.

Specific issues:

- MemoryLayout: Where do we put things?
- Address translation and Paging: How do we map **logical addresses** seen by processes to **physical addresses** seen by the memory hardware?
- MemoryProtection: How do we keep processes from stepping on each other?
- Caching and VirtualMemory: How do we collapse a multilevel memory hierarchy (L1 cache, L2 cache, DRAM, secondary storage, tertiary storage) to a single virtual address space?

---

CategoryOperatingSystemsNotes

## 58. MemoryLayout

Consider a typical C program. We have a large collection of global variables (mostly functions), and the code and/or data for each needs to go somewhere in memory. How do we decide where it goes?

## 59. Fixed addresses chosen by the programmer

The simplest approach (for the computer) is to let the programmer specify fixed addresses for everything. So `hello()` goes at 0x43770, `exit()` goes at 0x3817, etc. This has a lot of problems:

- Programmer has to keep track of a lot of boring clerical details.
- What if we put two things in the same place by accident?

- What if we need to move something or make it bigger?

Consequently, nobody does this, and our compilation tools actually make it pretty hard to do it.

## 60. Fixed address chosen by the linker and/or loader

Instead, we design our object files to be **relocatable**: in addition to the compiled machine code, the file contains a **symbol table** that says what addresses are used in the code and where. A **linker** resolves these symbols to specific fixed locations, and writes its choices everywhere a symbol is used. The linker is also typically responsible for finding necessary routines from libraries.

On Unix, the linker is usually called `ld` and runs after the rest of the compilation is done. You can see what it is doing by looking at symbol tables using the `nm` program. For example, given the following short program `short.c`:

Toggle line numbers

```
1 int
2 main(int argc, char **argv)
3 {
4     return 0;
5 }
```

The output of `nm short.o` after running `gcc -c short.c` (on an IA-64 Linux machine) looks like this:

```
0000000000000000 T main
```

This says that `short.o` contains one text (code) symbol `main`, which is assigned a rather boring address. The output of `nm short` after running `gcc -o short short.o` looks like this:

```
00000000005008c0 A __bss_start
000000000040042c t call_gmon_start
00000000005008c0 b completed.4829
00000000005006d0 d __CTOR_END__
00000000005006c8 d __CTOR_LIST__
00000000005008a8 D __data_start
00000000005008a8 W data_start
00000000004005c0 t __do_global_ctors_aux
0000000000400450 t __do_global_dtors_aux
00000000005008b0 D __dso_handle
00000000005006e0 d __DTOR_END__
00000000005006d8 d __DTOR_LIST__
00000000005006f0 D __DYNAMIC
00000000005008c0 A __edata
00000000005008c8 A __end
00000000004005f8 T __fini
00000000005008c0 a __fini_array_end
00000000005008c0 a __fini_array_start
0000000000400490 t frame_dummy
00000000004006c0 r __FRAME_END__
0000000000500888 D __GLOBAL_OFFSET_TABLE__
0000000000500888 w __gmon_start__
00000000004003c0 T __init
00000000005008c0 a __init_array_end
00000000005008c0 a __init_array_start
0000000000400608 R __IO_stdin_used
00000000005006e8 d __JCR_END__
00000000005006e8 d __JCR_LIST__
00000000005006e8 w __Jv_RegisterClasses
0000000000400550 T __libc_csu_fini
00000000004004d0 T __libc_csu_init
00000000004004d0 U __libc_start_main@@GLIBC_2.2.5
00000000004004b8 T main
00000000005008b8 d p.4828
0000000000400400 T _start
```

Here we have a lot of extra library stuff. Note that one of the symbols `__libc_start_main@@GLIBC_2.2.5` is unresolved; this will be filled in at load time using *dynamic linking* (see below).

## 61. Dynamic linking

The process of linking at load time (when a program starts) is called **dynamic linking**. This is often used for shared libraries, where **static linking** at link time would require making a copy—possibly a very large copy—of each library in each program. Instead, the OS uses address translation trickery to make a single copy of the library available in the address spaces of all processes that use it. But since we don't want to fix the location of this copy (since we don't know what libraries will be loaded and where), we delay resolution of library symbols until load time. This can either be done in the kernel itself or by a userspace program (e.g. `ld.so` in Linux).

In some systems (e.g. Multics), dynamic linking could even allow library routines to be replaced in a running process. Most modern systems don't provide this feature.

## 62. Dynamic loading

We can defer linking even further by allowing a program to run without including all of the procedures it might eventually need. Instead, we allow the program to load new object files after it is already running, a process called **dynamic loading**. In its most general form, this can

be used to add new functionality to an already-compiled program, by allowing it to load in new modules. It can also be used to allow a program to defer loading into memory routines or modules that are infrequently used.

This latter technique of **on-demand dynamic loading** or **autoloading** typically involves replacing routines in dynamically-loaded modules with **stubs**, which call the dynamic loader the first time they are called and then replace themselves with the newly-loaded routine.

---

CategoryOperatingSystemsNotes

## 63. MemoryProtection

**Memory protection:** How to make sure you can't read (or worse write) my data (unless I want you to).

## 64. Segmentation approach

Break physical memory up into **segments**. Limit who can access which segment by supplying each with a **base** and **limit** value and some protection bits. Only let the kernel modify the segment addresses, and trap if a process tries to access an address outside its segment.

## 65. Paging

Same thing but use page tables so that processes can't even see physical addresses they aren't supposed to use. Additional constraints in page tables may mark pages as e.g. execute only or read only, so that programs can't rewrite their code (or shared libraries) by accident.

## 66. Software memory protection

Make processes responsible for not generating out-of-bounds addresses, perhaps by using a strongly-typed language like Java or C# that doesn't give you the ability to hit arbitrary memory addresses.

In general, this violates the *never trust a process* rule, but it can perhaps be enforced if you have control over the compilation toolchain and can verify that the code that you have came out of a trusted toolchain (e.g. using a digital signature). Alternatively, require the compiler to insert explicit bounds checks and only run code that has such checks.

---

CategoryOperatingSystemsNotes

## 67. Paging

Previous forms of address translation: base and bound registers, segmentation. These are not very flexible.

Today: **paging**.

## 68. Basic idea

- Map logical addresses to physical addresses through a **page table**.
  - Typical multilevel page table:  $x.y.z \rightarrow \text{address at table}[x][y]+z$ 
    - $x$  is offset into **page directory** (IA-32 terminology)
    - $y$  is offset into **page table** pointed to by page directory (IA-32 terminology)
  - Page table pointed to by **page table base register** in MMU (CR3 on x86).
  - Actual contents stored in physical memory.
- On Intel machines, done after segmentation.
- Page table entries
  - Physical address
  - Flag bits
    - valid/invalid
      - Attempts to access an invalid page cause a **page fault** which is handled by the kernel
        - (more on what we can do with this soon)
    - Protection mode: readable, writable, executable
    - Protection mode: kernel vs user mode
    - Address of physical memory **frame** holding the page
    - Status bits: dirty, accessed
      - Used to manage VirtualMemory caching

## 69. Translation Lookaside Buffer

- Problem:  $\text{table}[x][y][z]$  requires three 50ns memory accesses!
- Solution: cache frequently-used page table entries in a **translation lookaside buffer**
- This is a special case of **caching**
- Associative memory of (key, value) pairs
- Page numbers are checked against all keys simultaneously

- **TLB hit:** use associated value! We just saved ~100ns.
- **TLB miss:** go to the page table.
  - Fetched entry goes in TLB
  - Somebody gets replaced! Random or LRU.
- Invalidating the TLB
  - If page table changes, TLB entries become out of date.
    - Any change to PTBR invalidates TLB
    - Kernel may invalidate specific entries (e.g. INVLPG instruction on x86)
    - Kernel may invalidate all entries
      - Note TLB is generally not very big, so cost is not necessarily that high
    - Invalidating TLB adds to effective cost of a context switch
- Address-space identifiers
  - Store ASID with each TLB entry
  - TLB hits must match page number and ASID
  - Now context switch doesn't require flushing entire TLB at once

## 70. Page table tricks

- Pages can be shared between processes
- Kernel space can be mapped to the same addresses in all page tables (with protection bit set)
  - Switching to kernel mode or between kernel threads doesn't flush TLB
- Invalid bits can be used to get full software control over memory accesses.

## 71. Page table structures

### Hierarchical paging

- Basic multi-level approach with fixed-size directories/tables/etc.
- Page table can be *very big*: e.g. 64-bit addresses with 4K pages  $\rightarrow 2^{52}$  page table entries per process.
- Multi-level directory reduces cost but not enough.

### Hashed page tables

- The answer to all data structure problems: use a hash table.
- May require following long hash chains if we get a lot of collisions.
- Variant: **clustered page tables** which are essentially multi-level page tables where top levels are hashed.

### Inverted page tables

- Idea: store table as (physical frame, logical page) instead of (logical page, physical frame)
- Selling point: page table bounded by (size of physical memory) / (page size)
- Cost: can't do address translation without searching the entire table
  - So use a hash table
- Subtler cost: can't implement shared pages, since each physical frame maps to exactly one logical page

---

CategoryOperatingSystemsNotes

## 72. VirtualMemory

Basic idea: Allow address spaces to contain logical addresses that don't refer to *any* physical address, but instead point to blocks on a disk. Use the page fault mechanism in the paging system to copy the disk blocks into main memory as needed (this requires flushing other blocks back to disk to make room).

Mostly we are cribbing from SilberschatzGalvinGagne Chapter 9 here, so you should read that instead of this page.

## 73. Terminology

### Backing store

Where we put pages that don't fit in main memory (typically some sort of disk). The specific region on a disk used for backing store is often called **swap space**.

### Pager

Program or other mechanism that manages moving pages back and forth.

### Demand paging

Only fetching pages when requested (as opposed to e.g. loading up all pages from a program at startup).

### Resident set

Set of pages from a process that are present (*resident*) in main memory.

### Page fault

Failed attempt to read a page from main memory.

## 74. Handling a page fault

What does kernel do when a process requests a page marked as invalid in its page table?

- Check against VM page table (possibly implemented in the same structure).
  - Maybe it really is an invalid address → SIGSEGV the process.
  - Otherwise it's out on backing store (or somewhere else).
- If on backing store, allocate a physical memory frame from the free list and copy it in.
  - If free list is empty, this requires flushing some other page first. If the other page hasn't been written to (it is **clean**), we can just throw it away. If it has been written to (it's **dirty**), then we have to write it out before reusing its frame.
- Restart the process when the page is available.

## 75. What can we page?

Userspace processes? Sure, why not? Except don't page out your disk driver daemon.

Kernel code? As long as you don't page out your pager.

Page tables? Yes, if you are *very very careful*.

We will mostly ignore this issue.

## 76. Performance

Expected time for memory access = (probability of page fault) × (cost of servicing page fault) + (normal memory access time).

E.g.

- Normal access time = 10ns
- Service time = 10ms =  $10^6$  normal memory accesses.
- $p = 0.001$
- Expected access time =  $10.01\mu\text{s} = 10^3$  normal memory accesses!

So we need a *very low* page fault rate:  $\sim 10^{-6}$  if we want to only double expected access time, lower if we want to be able to ignore the VM.

Fortunately, that  $10^{-6}$  isn't as ludicrous a page fault rate as it looks: for a typical program with a lot of **locality**, a loop spinning through the same page or two generates millions of accesses that are probably not going to generate page faults as long as we are smart about not swapping out the pages it uses. Getting a low page fault rate by being smart in this way is the job of the **page replacement algorithm**.

## 77. Page replacement strategies

Observation: most processes have a **working set** of actively-used pages, whose size can be much smaller (say 10%) than the size of the address space. If the working sets of all the active processes fit in memory, we can avoid having any page faults.

This leaves us with two conflicting goals (cf. process scheduling):

- Get a good approximation to the working sets resident in memory so that we don't get too many page faults.
- Don't spend too much time thinking about paging.

The page replacement algorithm chooses which pages to get rid of; the idea is that the pages that are left should be ones we are likely to need again. To make this work, we will assume that the working set of each process changes slowly over time, so that a page we are using now we are likely to need again soon.

### FIFO

Flush the page that has been resident longest. Ignores usage. We don't like it: among other things, some request sequences can cause *more* paging as memory gets larger. (See SGG §9.4.2 for analysis of the bad sequence 123412512345, which causes slightly more page faults with 4 frame than 3).

### Optimal paging (OPT)

If we know the request sequence, we can compute an optimal page replacement schedule with dynamic programming. But we usually don't know the request sequence.

### Least-recently-used (LRU)

Flush page that has been untouched the longest. Requires hardware support to track page-use time (e.g. in TLB); overhead to track *every* reference in software is prohibitive. So we are at the mercy of hardware vendors, who don't actually provide last-access-time fields.

### Approximate LRU

What we do get out of hardware is **reference bits** that indicate whether a page has been written to or read from since we last cleared the bits. This lets us make guesses about which pages are LRU or at least which have not been used in a while. Some variants::

#### Second chance (clock algorithm)

Pages are in a circular queue. Clock pointer acts like the pointer in FIFO, except that if a page's reference bit is set, we clear it and move on to the next page in the cycle. In the worst case we clear every bit and get back to the original page (we hope this doesn't happen too often—it basically means we get a very expensive implementation of FIFO). Why this works: frequently-accessed pages will get their reference bits set before we get back to them.

#### Enhanced second chance

Basic second chance prefers flushing unread pages (0 bit) to read pages (1 bit). In the enhanced version we have separate access bits and dirty (modified) bits, and prefer flushing 00 to 01 to 10 to 11; the idea is that given a choice between a clean page to

flush and a dirty page to flush, we flush the clean page because we don't need to to an extra disk access to write it out.

#### Counting schemes

e.g. **least-frequently used**, **most-frequently used**. Mostly useful to make other page-replacement strategies look good.

## 78. Buffering strategies

By keeping a few spare frames around we can speed things up. Instead of flushing a victim page immediately on a page fault, we allocate an empty frame and read the new page into it, so we can restart the processes after 1 disk access instead of 2 disk accesses. The victim page is then flushed in the background when the disk is otherwise free.

A natural way to implement this is to have a swapper daemon process that just flushes likely victims when it can. We can do this because the page replacement algorithm generally only selects victims and doesn't care about what we are replacing them with.

## 79. Other virtual memory tricks

### 79.1. Shared pages

We've already mentioned (in Paging) the use of a paging system to allow processes to share blocks within their address spaces, e.g. for doing InterProcessCommunication or keeping around only a single copy of a read-only executable or shared library. The same tricks work just as well when paging is supplemented by virtual memory.

### 79.2. Copy-on-write

Suppose we have two processes that share most of their initial state but may diverge over time (e.g. the parent and child in a `fork` in Unix). Instead of allocating pages for both copies, we keep a single copy of each duplicate page and clone it only when one of the processes tries to write to it (which we can detect as a page fault by turning off write access to all the shared pages). This is also useful for processes that start off with large blocks of zeros in their address space (e.g. big global C arrays, or the result of calling `sbrk` to ask for more heap space from the OS)—instead of handing out thousands of identical blank pages, we hand out thousands of pointers to the same page, and allocate the real pages only when we need to.

### 79.3. Memory-mapped files

The POSIX `mmap` call lets you take a file in the filesystem and **map** it into memory. Here the VM mechanism allocates a region in the address space that is backed by the file rather than the swap space. Otherwise everything looks exactly like standard VM: pages from the file are read in on demand, and written back (if dirty) when space is needed. There is a slight complication in that eventually the file is unmapped (either explicitly or when the process terminates), and we need to go and flush all the pages when this happens (we may also provide a mechanism to force a flush, e.g. if we want to be sure that our changes will survive a power failure).

## 80. Bad outcomes

The bad outcome for a virtual memory system is **thrashing**, the VM equivalent of a traffic jam where the total working-set size of all active processes exceeds the size of physical memory. Now a large proportion of memory accesses will cause page faults, and though the system will continue to struggle along sluggishly, we can expect to see a slowdown of many orders of magnitude.

This problem is similar in many ways to Deadlock, and has essentially the same solutions: ignore the problem, or detect the disaster and kill or suspend processes to reduce the conflict. As with deadlock, "ignore the problem" is easier to implement and arguably gives the user more control.

---

CategoryOperatingSystemsNotes

## 81. InputOutput

The I/O subsystem is the part of an operating system that interfaces with input/output devices. Since these devices come in a wide variety of flavors, it is usually the most complex part of the kernel. To handle this complexity, the I/O subsystem is typically split up among device drivers that all present their underlying devices to the kernel using one of a small number of standard interfaces—essentially an object-oriented approach.

Usual disclaimer about sketchy lecture notes applies. For more details see SGG Chapter 13 or TanenbaumBook Chapter 5.

## 82. I/O devices (user view)

Many flavors. These can be broadly classified into:

#### Block devices

Disk drives, tape drives, USB memory keys, etc. At an abstract level, these look like a big pile of bytes organized into blocks of some fixed size. (Technically, the use of blocks is an optimization, to amortize control costs over many bytes of data.) Typical operations: read (blocks), write (blocks), seek (next block to read/write).

#### Character devices

Also called **stream devices**. Keyboards, modems, printers, audio cards, robots, USB Nerf cannons. These look like a stream of bytes with no particular structure imposed by the device. Typical operations: read, write. Some may provide very sophisticated functionality

on top of what is provided by the raw device; Unix terminal drivers are a good (?) example.

### Network devices

Somewhat similar to stream devices, but distinguished by having built-in various high-level network protocols that may add more structure (e.g. UDP datagrams, TCP connections).

There are also some specialized devices that less visible to the user but provide essential services to the kernel, like the interrupt controller (which typically includes a clock for scheduling regular interrupts) or a DMA controller (see below).

There are also several issues that come up quickly in designing the user-level I/O interface:

### Asynchronous vs synchronous I/O

**Asynchronous I/O** is **non-blocking**: the program (or CPU) keeps going after starting an I/O operation, and is notified by an interrupt when the operation completes. Most I/O operations at the hardware level are asynchronous—this is good, because I/O is slow. But programming for asynchronous I/O at the user level is difficult. So the kernel will typically present a **synchronous** or **blocking** interface where low-level asynchronous operations appear to be synchronous by the simple expedient of blocking processes that are waiting for them.

### Buffering

Data from I/O devices often arrives at inconvenient times or in inconvenient units. The use of buffers in devices and in the kernel can hide some of this from the user. However, storing data in buffers can slow down I/O by requiring excessive copying, so there are trade-offs involved.

### Sharing

Can an I/O device be shared or multiplexed between multiple processes? If so, the kernel needs to take responsibility for handling multiple requests. Examples of sharable devices include disk drives (usually via a high-level filesystem interface), displays (via window systems), or audio devices (unless you are running an old version of Linux). Other devices like tape drives or modems may be non-sharable, and the kernel has to enforce that only a single process can use each at a time.

### Naming

How does a process specify which device it wants? The Unix approach is to map devices into the filesystem (`/dev/tty`, `/dev/hda1`), which translates device names to a lower-level namespace consisting of major and minor **device numbers**. One can also structure a system to use a low-level namespace directly, but this loses the proverbial extra level of indirection that solves all programming problems.

## 83. I/O devices (hardware view)

At the hardware level, devices look very different from the user's view—so the operating system has to do a lot of work to hide this!

### 83.1. Controllers

We can think of each device as consisting of some physical object (the actual device) and a bit of hardware that presents an abstract interface to the object to the bus. This latter bit of hardware is called the **device controller**, and this is what the CPU interacts with. The question that remains is how to structure the interface between the CPU and the device controller.

Usually, a controller interacts with the CPU in up to three ways:

- Through **control registers**. These act like memory locations, though they may be accessed through specialized I/O instructions (see below). For example, a keyboard device controller might have a write-only control register that sets the state of the LEDs and a read-only control register that holds the last character typed. A more sophisticated controller might have many control registers: a graphics controller might have many options for controlling scan timing, placement of data in memory, etc., that are all accessed through separate control registers.
- Through **shared memory**. The device controller might present some of its data as memory on the memory bus (e.g. the VGA display memory at 0xB8000). Or it might use **direct memory access** (see below).
- Through **interrupts**. Control registers and shared memory don't allow a device to signal when an operation has completed or data is ready. For this purpose, device controllers are given access to CPU interrupts (though usually via an intermediary **interrupt controller**).

### 83.2. Input and output instructions

Many CPUs provide IN and OUT instructions for doing I/O. These look like the MOV instructions used to access memory, and take addresses (**ports**) as arguments that may even go out over the same address bus as memory operations, but a pin on the CPU signals that they are to be interpreted as I/O operations instead of memory operations, and so the supporting hardware routes them to the appropriate device.

Advantages:

- Can use a separate lower-speed bus for I/O and higher-speed bus for memory.

Disadvantages:

- Can't program I/O in C—need some assembly language to use IN and OUT instructions.
- I/O instructions require a separate privilege mechanism from memory protection.
- Can't use most assembly-language instructions to access I/O devices. For example, to test the state of an I/O register we have to read it into a CPU register first and then test it.
- Number of I/O ports may be limited. For example, on x86 you only get 16 bits worth of I/O ports but up to 36 bits worth of memory addresses.

### 83.3. Memory-mapped I/O

Here instead of using a separate address space for I/O, we use the same address space as for memory. The address translation hardware takes responsibility for diverting I/O addresses to I/O devices.

Advantages:

- I/O looks just like memory accesses, which we already know how to do.

Disadvantages:

- Treating I/O as memory can interact badly with cache system.
- Doesn't allow for multiple buses.
- Moves complexity from the CPU to the bus hardware.

## 83.4. Hybrid approach

One common approach is to use both I/O instructions and memory-mapped I/O. The idea is that I/O instructions can be used for small values that change quickly (e.g. control registers) and memory mapping can be used for large chunks of data that are read or written in bulk (e.g., display memory). So, for example, the PC architecture allocates the memory range 0xA0000–0xFFFF0 (640K–1M) to memory-mapped I/O, and it is here that one finds VGA display memory etc.

This approach still requires a bus controller to divert memory-mapped I/O addresses to the I/O bus.

## 83.5. Direct Memory Access (DMA)

For block devices, it often makes sense to bypass the CPU and give them direct access to memory. This is usually done by providing a bridge between the I/O bus and the memory bus in the form of a **DMA controller**. The DMA controller looks to I/O devices exactly like the CPU: it sends the same control instructions and reads data across the same bus. But it is much more limited in what it does to the data; all it can do is stuff it in memory at a physical memory address specified by the CPU (through control registers on the DMA controller accessed by the CPU through the I/O bus), and issue an interrupt when the operation is done.

Note that because the CPU and DMA controller share the same buses, some concurrency control mechanism is needed to prevent them from stepping on each other. (A similar issue arises with multiple CPUs). There are some options here for how aggressive the DMA controller is about grabbing the bus. A polite DMA controller might limit itself to **cycle stealing**—grabbing the bus for a few cycles from time to time, so that the CPU never loses control of the bus for more than a few cycles. A less polite but more efficient DMA controller might use **burst mode** where it seizes the memory and I/O buses long enough to complete a full block read or write operation, or possibly even several such operations. The downside of this approach is that the CPU might stall because it doesn't have enough data in its cache to proceed until the I/O operation is finished.

Using a DMA controller adds complexity to both the hardware and the OS. The payoff is that it takes load off the CPU, which might allow it to proceed with its own CPU-intensive tasks.

## 84. A typical I/O operation from start to finish

Suppose a user process asks to read a block off of the disk (we will ignore the filesystem interface for the moment, and imagine it is reading directly from a non-shared disk device). What happens?

1. Process executes `read` system call with appropriate arguments.
2. Kernel translates arguments to low-level device controller instructions and writes them to the disk driver controller registers (or possibly hands this task off to the DMA controller). It also blocks the process and proceeds with some other process.
3. Disk drive controller does whatever magic it needs to do to get the relevant bits off the physical disk hardware.
4. Bits are read into a buffer on the disk drive controller. These are transmitted serially across a cable between the disk drive controller and the disk.
5. Disk drive controller verifies bytes read by checking checksums, looking for drive error codes, etc. It may retry the read operation if it failed.
6. If everything worked, disk controller signals CPU and/or DMA controller that data is ready.
7. CPU or DMA controller reads data from disk controller buffer and copies it to main memory.
8. Kernel maps or copies data into process's address space and unblocks it.

There are several options for how the various stages of this process work. One big choice is which device is responsible for waiting for the I/O operation to complete. Choices are:

### Programmed I/O

CPU runs everything itself. Device controller is a pathetic stub whose only purpose is to translate CPU bus commands directly to the I/O hardware. Examples are WinModems, some printer controllers, some very early primitive graphics controllers. Advantage: Allows for cheaper hardware; may allow for very sophisticated I/O programming (handy with modems). Disadvantage: Like hiring a Formula 1 mechanic to change your oil; CPU has to **poll** I/O device to see if it needs to do anything; in the worst case, I/O device eats up the entire CPU, locking up everything else while the I/O operation proceeds.

### Interrupt-driven I/O

Device controller does most of the work, notifies CPU when done. CPU still has to extract data itself.

### DMA-based I/O

DMA controller takes over CPU's role, interrupts CPU when data is available in memory. Advantage: minimum load on CPU, especially in the form of fewer interrupts since DMA controller can do more buffering. Disadvantage: maximum hardware complexity.

## 85. I/O system architecture

From the ground up, we have



## Interrupt handlers

These respond immediately to I/O interrupts and handle incoming data, usually by calling an **interrupt service procedure** supplied by the device driver (see below). Core kernel component.

## Device drivers

Higher-level interface that translates user-visible device abstractions (e.g. block devices) to low-level incantations sent to device controllers. Usually in kernel because of the need for privileged I/O instructions. On portable operating systems, are often separated into an **upper half** that is machine-independent and a **lower half** that varies depending on the CPU architecture. The lower half executes machine-specific I/O instructions and usually provides the guts of the interrupt service procedure.

## Device-independent I/O software

Anything that doesn't depend on the specific details of a device beyond its abstract description as e.g. a character or block device.

This may include higher-level mechanisms like network stacks or filesystems as well as lower-level mechanisms like virtual-memory management or I/O scheduling systems.

## User-level I/O libraries

Think `stdio`. Adds another layer of buffering and abstraction on system call interface.

## User programs

What the game is ultimately all about.

# 85.1. Device driver architecture

Having gazillions of bizarre devices, each with its own specialized control mechanism, leads to pressure to move device drivers out of the core kernel. This gives a progression of increasingly decoupled device driver designs:

- Traditional approach: handful of device drivers baked into the kernel at compile time.
- Modern approach: most device drivers compiled separately and loaded into the kernel at run time.
- Microkernel approach: kernel exports enough low-level I/O functionality to allow device drivers to be written as specialized userspace processes.
- Exokernel approach: kernel exports even more low-level I/O functionality to allow device drivers to be built into arbitrary userspace processes via system libraries.

### 85.1.1. What a device driver module looks like

First we need to settle on a standard interface. For a Unix-style block device, we might have something like:

Toggle line numbers

```
1 struct dev_hdr {
2     /* initialize the device; called once at boot time or module load time */
3     int (*init)(void);
4
5     /* shut down the device; called once at shutdown */
6     int (*halt)(void);
7
8     /* read or write the given number of bytes to or from buffer */
9     int (*read)(unsigned long bytes, void *buffer);
10    int (*write)(unsigned long bytes, void *buffer);
11
12    /* next read or write works on this position */
13    int (*seek)(unsigned long position);
14
15    /* loophole operation for everything else */
16    /* uses variable arguments */
17    int (*ioctl)();
18 }
```

A character device would be similar, except it wouldn't provide `seek`.

This is essentially an implementation in C of a C++ or Java method table. The `ioctl` loophole is there to provide extra methods for devices that need them (e.g. for setting parameters on a disk drive, turning autofire off on your Nerf cannon, or toggling the LEDs on your keyboard). In a real OO programming language this would be handled by subclassing.

As kernel development continues and certain operations start appearing across multiple devices, they are likely to move out of the `ioctl` bin and into the main structure. The final result might be something like the `file_operations` data structure from the Linux 2.6 kernel (see `/usr/src/linux/include/linux/fs.h`):

Toggle line numbers

```
1 struct file_operations {
2     struct module *owner;
3     loff_t (*llseek) (struct file *, loff_t, int);
4     ssize_t (*read) (struct file *, char __user *, size_t, loff_t);
5     ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
6     ssize_t (*write) (struct file *, const char __user *, size_t, loff_t);
7     ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
8     int (*readdir) (struct file *, void *, filldir_t);
9     unsigned int (*poll) (struct file *, struct poll_table_struct *);
10    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
11    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
12    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
13    int (*mmap) (struct file *, struct vm_area_struct *);
14    int (*open) (struct inode *, struct file *);
15    int (*flush) (struct file *);
16    int (*release) (struct inode *, struct file *);
17    int (*fsync) (struct file *, struct dentry *, int datasync);
18    int (*aio_fsync) (struct kiocb *, int datasync);
```

```

19     int (*fasync) (int, struct file *, int);
20     int (*lock) (struct file *, int, struct file_lock *);
21     ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
22     ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
23     ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
24     ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
25     unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
unsigned long);
26     int (*check_flags)(int);
27     int (*dir_notify)(struct file *filp, unsigned long arg);
28     int (*flock) (struct file *, int, struct file_lock *);
29     int (*open_exec) (struct inode *);
30 };

```

Here the `struct file *` arguments are pointers to the kernel's internal data structure associated with the device (which is pretending to be a file from the point of view of the file system), the equivalent of `self` or `this` in an OO language. This allows using the same driver procedures for multiple similar devices.

### 85.1.2. Loading a device driver

Device drivers for core resources like the display or keyboard are likely to be built directly into the kernel. Device drivers for hardware that may or may not be present are more likely to show up as loadable modules.

Loading a device driver is much like loading a program. The kernel (or possibly a userspace process with especially good access to kernel memory) reads the driver from disk (typically including a linking and relocation step), allocates space for it, then calls its setup routine to initialize it. This initialization may require further calls to internal kernel routines to allocate more space or new kernel threads, may involve setting up the device, and so forth.

Subsequent calls to the device driver happen as internal procedure calls via the driver's jump table. The driver generally has full access to the kernel's address space and is usually compiled against a copy of the kernel so that it can be linked to kernel procedures.

### 85.1.3. Risks

Loadable kernel modules are the biggest risk to kernel integrity, since they are often written by monkeys but run with full access to kernel internals. A rule of thumb is that device drivers have 5-10 more bugs per thousand lines of code than core kernel code, and unlike a user process a broken device driver can bring down the entire system. In the worst case kernel modules can be used maliciously to build rootkits and similar bits of nastiness. At minimum this means that we can't let just anybody load new modules into the kernel. In the long run, it creates pressure to provide better containment for device drivers, either by using strongly-typed languages and/or signed code to constrain or at least vet their behavior, or by pushing them out into a less destructive part of the system using microkernel trickery.

## 85.2. Device-independent I/O components

These include things like buffering and scheduling mechanisms. The idea is that once we get above a standard device abstraction, we can implement common techniques like buffering and I/O scheduling in one place instead of scattered throughout the device drivers. It is also here that we expect to see things like terminal drivers, filesystems, and network protocol implementations.

## 86. Example: console/keyboard driver

Let's look at how we might write a driver for the console (built-in VGA display) and keyboard on a PC. We might or might not want to separate these out into separate devices, but for the moment let's imagine that we've globbed them together as a single character device, where writing to the device prints to the screen and reading from the device reads characters from the keyboard.

### 86.1. Output

Writing to the screen is the easier task, since the screen is memory-mapped and we don't have to deal with issues like interrupts or the screen not being ready for output yet. We will probably still maintain some auxiliary information like cursor position in some kernel-internal data structure, so that we can present users with a clean stream-of-characters interface rather than force them to keep track of this for themselves.

So for output we will provide a `write` routine that:

1. Takes a buffer of characters and copies them to screen memory at the location of the current cursor.
2. Updates the cursor position.
3. If necessary, wraps or scrolls the screen (which may involve quite a bit of copying!)

This routine will typically be called inside some process's kernel thread after being dispatched from some higher-level `write` system call.

Some questions we might want to ask: Can more than one process write to the screen? If so, what happens? At minimum we probably want to protect the screen data structures with a mutex so that write operations appear to be atomic.

We may also want to allow the user to control other features of the screen like the starting cursor position, character color, etc. There are two obvious ways to go about this:

1. Use **terminal escape sequences**. These are codes embedded in the stream of characters being written that change the terminal's behavior. There is an ANSI standard for terminal escape sequences (see [Wikipedia: ANSI\\_escape\\_code](#)), so this has the advantage of compatibility with vast piles of software that expect to be able to use them. The downside is that it adds complexity to the terminal driver and may require quoting output bytes that aren't intended to control the terminal functions.
2. Use the `ioctl` mechanism or some similar expansion mechanism to provide system calls that allow access to cursor position, character attributes, etc. This is in many ways a less desirable solution, even though it provides more consistency with the standard device driver approach. One reason why this is less desirable is that a program (say a text editor) may wish to update the cursor

position and change character attributes many times within a short sequence of output. With an `ioctl` approach this would require many system calls—and their associated overhead both in time and programmer effort—instead of just one.

In practice the cost of doing multiple system calls means all but the laziest OS's provide some mechanism for processing escape sequences. This choice is analogous to the "little language" that appears in `printf` format specifiers (or FORTRAN FORMAT statements before them), where again a pure procedural interface is replaced by string processing.

## 86.2. Input

Keyboard input is more interesting, since we can't predict when the user will hit a key. On the IBM PC architecture the main keyboard is assigned IRQ 1, so we can use an interrupt handler to snarf keyboard input as it arrives. This is done by executing an `inb` operation on port 0x60, which will return a **scan code** which describes the position of the key that had something happened to it (either a key press or a key release). The scan code is not likely to be very useful to a user process, so we will probably want to do some translation. This gives us three steps in processing a keyboard event:

1. Handling the interrupt. At minimum we need to save the incoming scan code somewhere.
2. Translating the scan code to a more useful form. Many keyboard events we may be able to safely ignore (like releasing the E key). Some will require updating internal state that affects further translation (shift down/shift up). Some might even require doing some output operations to update the keyboard status (caps lock down—requires turning on or off the caps lock light). We may also want to give the user control over how this translation is done (a video game might want to know when the user releases the keep-jumping key).
3. Delivering the result to any interested process or processes.

The interesting part here is that we have a lot of options over where to do these operations. If our goal is to get out of the interrupt handler as fast as possible, we might want to just do step 1 there, sticking the incoming scan code into a ring buffer somewhere and returning immediately. The translation and delivery steps could then be handled by any of (a) a kernel thread that wakes up from time to time when the ring buffer is nonempty (semaphore!) and does translation (monolithic kernel approach); (b) a userspace daemon that does the same (microkernel approach); or (c) a standard library routine that uses an address space mapping that provides direct access to the raw scan code buffer (exokernel approach). The more distant we make the translation and delivery process from the core of the kernel, the more control we give the user (e.g., being able to substitute different translation schemes, being able to stuff fake keyboard events into the input stream, being able to build a fancy keyboard-dispatching daemon that sends all the vowels to process 1 and all the consonants to process 2—who knows?), but the more costs we impose. For keyboard input, the costs are likely to be pretty low no matter what we do, so a simple approach is probably best. This probably either involves providing raw untranslated input or doing translation at the kernel level, perhaps even in the interrupt handler (since it's not very expensive).

We still have to decide how to get keystrokes to the user process. The choices here are eerily similar to the choices between programmed I/O (polling), interrupt-driven I/O, and DMA-based I/O at the kernel level:

1. Provide a `getchar` system call that lets a user process wait for a key event. To implement this, we build a ring buffer in the kernel with an attached semaphore and have the user process (really its corresponding kernel thread) down (*proberen*) the semaphore when it wants a character. The process will wake up again (and extract the character from the ring buffer) when a character is available, because the interrupt handler or device driver upper half will up (*verhogen*) the semaphore when it adds a character to the buffer. We can also make a non-blocking version by allowing the process to peek at the buffer (presumably protected by a mutex) to see if there is anything there. The downside of this approach is that with the blocking version, the process can't do anything else while it is waiting (unless it has multiple threads) and with the non-blocking approach it wastes a lot of time checking for keypresses that haven't arrived yet.
2. Signal the process asynchronously when a character arrives, e.g. using the POSIX SIGPOLL mechanism. Annoying for processes that really do want to wait for keypresses.
3. Deliver the message using some high-level InterProcessCommunication mechanism. The nice thing about this is that now we can make anything pretend to be a keyboard. The disadvantage is that there may be a lot of overhead.

We also have an independent choice about *when* to deliver characters. Many processes might only care about receiving characters when the user hits enter, and might like the OS to handle details like buffering incoming characters and processing simple editing commands. There is a good chance that a process also would like the OS to each typed characters to the screen (at least some of the time). The logical response is to provide a sophisticated **terminal driver** that handles all of these tasks, delivering pre-masticated lines of text to the process via `read` system calls (if the terminal driver lives in the kernel) or an IPC mechanism (if it doesn't—in this case we may be able to give the IPC mechanism a file-like interface so the receiving process doesn't notice). The more sophisticated this terminal driver is, the more incentive there is to move it out of the kernel so that bugs won't trash anything else, but the more incentive there is to keep it in the kernel so that it has fast access to in-kernel data. It's not clear in the limit when the terminal driver grows to include a window system which approach is better; both internal window systems (Windows) and external window systems (X11, as used in Linux and OS/X) are both widely used.

## 87. Example: disk drivers

For block devices like disks the situation is more complicated, because the data rate is much higher (~500 Mb/s vs ~10 bytes/s for the keyboard) and latency is much more of a problem since so much might be waiting on the next disk access. So we want a disk driver that can buffer and multiplex disk accesses so that processes (and internal kernel threads like the virtual memory pager) think they happen quickly even if they don't. We also have to deal with the fact that disk operations are long, multi-stage processes that involve both input and output (even if we are only moving data in one direction), so we can't hope to just bury everything down in an interrupt handler.

We can simplify the design by assigning a kernel thread (or userspace daemon) to manage each disk drive. The input to the thread is a queue of disk operation requests that are inserted by higher-level parts of the I/O subsystem, and the thread runs in a loop where it selects an operation to perform (possibly out of order to optimize disk scheduling—we'll talk more about this when we talk about FileSystems), sends the appropriate commands out to the disk controller or DMA controller, and then sleeps until the disk operation is finished (which will be signaled by an interrupt handler whose job it is to wake up the driver thread). The thread can then deliver the results to the requesting process or kernel thread via internal kernel buffers or a high-level IPC mechanism (the former is likely to be faster since it may involve less copying).

The downside to this approach is that it may involve a lot of overhead if we have a lot of disk drives, or if we need to do more coordinating between disk operations (e.g. by scheduling access to the bus). So a more sophisticated system might have a single kernel thread that polls

all the drives (with the state of the former multiple threads moved into one big table somewhere) or might do disk operations as part of some regular polling procedure that runs as part of the periodic timer interrupt. There are many options here.

---

CategoryOperatingSystemsNotes

## 88. BlockDevices

**Block devices** are I/O devices—principally disk drives and similar mass-storage devices—that are typically accessed by slow operations whose cost is amortized over large blocks of data.

## 89. The problem with mass storage

Mass storage devices are slow. Typical consumer-grade disk drives have **seek times** on the order of 10ms, during which a reasonably fast CPU of 2007 can execute about  $10^8$  instructions. This alone means that disks must be given special attention to ameliorate their huge costs. There are several additional properties of disks that the OS must work hard to ameliorate:

1. There is a long delay between initiating a read or write operation and its completion. This means that the disk I/O subsystem can't block anything else while waiting, and there is an incentive to buffer and cache data as much as possible to hide the delay.
2. Disk delays can vary by a wide margin depending on the physical placement of geometry on disk. This encourages careful scheduling of disk operations.
3. Disks have the role of providing **stable storage**—storage that survives reboots or power failures—but are nonetheless themselves mechanical devices prone to breakdown. This requires redundancy mechanisms like RAID systems and backup to tertiary storage devices like optical disks or magnetic tape.

## 90. Hard disk structure

A hard disk consists of one or more **platters** spinning at 5400–7200 RPM. Each platter has one or more **heads** that ride above it on a thin cushion of air dragged along by the disk platter (disk drive heads are "ground-effect vehicles" like hovercraft). The heads contain electromagnets that can be used to sense or rewrite patterns of magnetic orientation in the disk platter. The entire mechanism is sealed to prevent dust from entering.

A typical arrangement is to mount all heads on a single arm, so that the move toward or away from the center of the disk in unison. This makes it natural to divide the disk up into **cylinders** consisting of stacks of circular regions on each platters which are individually called **tracks**. The **tracks** are further divided into **sectors**, each of which stores one block of data. The interface that the disk presents to the outside typically allows the OS to access data by specifying a cylinder (distance from the center of the disk), track (which platter), and sector (distance around the track), giving a three-dimensional coordinate system. These coordinates are usually lies, for two reasons: the disk usually has some extra space to make up for bad spots on the medium, and the disk's internal controller tracks the bad spots and allocates extra sectors to replace them, and most modern disks allocate more sectors to tracks toward the rim of the disk (which have a larger circumference) than to tracks toward the middle; so there will be some translation between constant-sized logical tracks and variable-sized physical tracks. However, as OS writers we can probably count on some correlation between track numbers and physical locations, and it is useful to exploit this correlation to speed up disk access.

The time to read data off the disk is dominated by the time to move the disk head to the right track and the time to wait for the right sector to come around underneath the head.

## 91. Bad blocks and error correction

It's hard (or at least more expensive) to build a perfect disk. So we can expect our disk platters to have some junky regions that don't hold data very well.

To detect this, we need to use some sort of checksum or error-correcting code. If we notice that some sector is consistently bad, we can put it on a **bad blocks list** that is stored elsewhere on the disk. Historically, this was often the job of the filesystem, under the assumption that bad blocks were (a) rare and (b) stable. Current high-density disks have enough bad blocks that this task is often handed off to the disk controller. With an error-correcting code, we can tolerate a degraded block (since the ECC fixes the missing bits), and if it consistently misbehaves, we'll cross it off. This is particularly important for devices like Flash drives, where individual sectors can only be written a limited number of times before they become unreliable.

## 92. Disk scheduling

Most disk drivers reorder disk operations to reduce head movement. This involves the use of a **disk scheduling algorithm**. Some options:

### First-come first-served

Exactly what it says. For randomly-arranged requests, requires seeking across half the disk on average. Guarantees no starvation.

### Shortest-seek-time first

Greedy approach; perform the operation that requires minimum head movement. Easily allows starvation.

### Elevator algorithm

Head moves in one direction only until it reaches the edge of the disk, then reverses direction. Operations are performed as the head reaches the appropriate track. This theoretically still allows starvation (imagine a continuous stream of operations on a single track that pins the head there), but avoids some of the problems with SSTF. There are several variants depending on whether the head processes requests in both directions, whether it is dumb enough to roll all the way to the edge of the disk even if there is nothing to do there, etc.

### Priority methods

Like priority scheduling in the CPU: some disk operations are given higher priority than others. Examples would be giving priority to the VM system (especially page writes) or to user processes over low-priority background processes (e.g. antivirus scanners—this sort of disk prioritization is one of the features added to Windows with the Vista release). These can also be used to prevent starvation by assigning high priorities to particularly old requests (e.g., Linux mostly uses SSTF but prioritizes disk requests over 5 seconds old).

## 93. Block device driver implementation

Because we want to be able to reorder requests, the simple approach to building a device driver where some process locks the device, executes its request, then unlocks the device doesn't work very well. So instead the usual approach is to set up a queue for operations and have the device driver draw a new operation from the queue whenever the previous operation finishes. Users of the driver interact only through the request queue, inserting new operations into the queue (and then possibly blocking waiting for the result). The driver itself is a thread or process that in its simplest form executes a loop that chooses an operation from the queue, signals the disk controller and/or DMA controller to begin executing it, then blocks until the operation completes.

## 94. Buffering, caching, and prefetching

Because the disk is so slow, a sensible driver will buffer disk blocks in main memory. This allows write operations to appear to finish immediately; if some other process asks to read the block just written, the disk driver returns the buffered block without waiting to write it to disk (or read it back). The disk driver can also maintain in the same space a pool of recently read blocks, so that it is not necessary to go to the disk if the some process asks for one again. Most modern kernels will use any spare available memory for this purpose—it's cheap, because we have to read the incoming disk blocks in somewhere, and since the data is backed by the disk we can always through these cached blocks away.

A further extension of this approach is **prefetching**. If the disk driver (or higher-level filesystem) has a good model of what blocks will be needed, it can fetch and cache these blocks off of the disk when it is cheap to do so or the disk is otherwise idle. This can vary from simple approaches like reading the next  $N$  sectors after the one that is requested (under the assumption that many file reads grab contiguous blocks of data) to recording detailed statistical data about what blocks are needed when and prefetching them into memory (e.g., Windows Vista's Wikipedia: SuperFetch). Even a small reduction in the number of unpredicted disk fetches can noticeably reduce latency.

## 95. Network-attached storage

- Move disks to the other side of a network.
- Disk access now goes a network protocol instead of through the system bus.
- Disks can be run by another general-purpose computer or by an NAS device.
- Most common approach is to do this at the FileSystem level e.g. with NFS.

## 96. Removable storage

- E.g., USB keys, Flash/SD/MMC cards, tapes.
- Mounting and unmounting: need to flush buffers before removing the device.
- Historically mostly used as slow but cheap **tertiary storage**; also used for data portability.
- Nothing beats the bandwidth of a station wagon full of tapes—*still true?*

## 97. Multiple disks

- *Striping: split sectors across  $N$  different disks.*
- *Now we can read them  $N$  times as fast!*
- *Except failures happen  $N$  times more often.*
- *Solution: RAID (Redundant Array of Inexpensive Disks)*
  - *Elaborate catalog of RAID architectures or **levels** (see SGG §12.7.3).*
  - *Basic idea: duplicate data so we can reconstruct it after one or two disks fail.*
  - *Naive approach: make extra copies*
  - *Sophisticated approach: use error-correcting codes*
  - *Real-world example: the Google File System uses the naive approach.*

## 98. Other block devices

Things that look like disks but aren't:

- RAM disks.
- Loopback devices.

---

CategoryOperatingSystemsNotes

## 99. FileSystems

A **filesystem** presents an abstract interface to one or more underlying **BlockDevices**. It may also provide a namespace for other OS mechanisms.

*It is possible to have an OS that doesn't provide a filesystem. We let user processes access disks directly, possibly with some very minimal access control like assigning different ranges on the disk to different processes. But essentially all usable operating systems provide a filesystem as a basic resource, and most use it enough internally that they couldn't run without it.*

# 100. Interface

*We'll start with the user's view and then move on to implementation issues later.*

## 100.1. Files

### 100.1.1. Structure

- Big bag of bytes
  - No OS-enforced internal structure to files
  - Modern default
  - Popularized by Unix
- Records
  - File is a sequence of records
  - Records have fixed length (ancient mainframe OSes) or variable length
    - Fixed length allows fast random access
  - Survives in vestigial form even in modern OSes
    - `gets`, `puts`
    - Standard record separators: `"\n"` (Unix), `"\r\n"` (MS-DOS/Windows), `"\r"` (old MacOS).
- Tree structure
  - File consists of named components, each of which in turn may consist of named components
  - Replaced by directory trees in modern OSes
  - Possible return via XML?
- Text vs binary files
  - Some OSes distinguish between text and binary files
  - Allows for special handling of text files, e.g. record separator translation.

### 100.1.2. Naming

- Structured names
  - MS-DOS style 8+3: `MZFIRFOX.EXE`, `WDDOMPLN.TXT`
    - Case-insensitive
    - Limited alphabet: alphanumerics plus `'.'` and `'_'`
    - Three-character extension indicates type
      - Can be used to choose which program can work on the file
    - Selling point: fit nicely in 16-byte directory table slots
  - VMS: `KREMVAX::BIGDISK:DOSSIERS:GORBACHEV.DOC;37`
    - Starts with node name and disk name (allows references to remote files).
    - Extension (e.g. `DOC`) indicates type.
    - Number at the end is a version number, automatically maintained by the filesystem.
    - Directories can be tucked in the middle.
- Less structured names
  - Unix filenames: up to 255 arbitrary characters
    - Case-sensitive: `file`, `FILE`, and `File` are all different files
    - Extensions if any are just more characters, interpreted by programs at their option
    - `foo.c.archive.tar.bz2.jpg`
    - Name may give no hint about contents or purpose of a file!
  - NTFS
    - Filenames in Unicode: `φύλε`
    - Some characters reserved
- See [WikiPedia: Comparison\\_of\\_file\\_systems](#) for many more exotic examples

### 100.1.3. Metadata

- Data associated with a file but not part of a file
- Core system stuff
  - Permissions
  - Access times
  - Location
  - Size
  - File content type (text/binary)
- Type information
  - E.g. Windows extensions
  - MacOS forks/NTFS Alternate Data Streams
    - Original purpose: tell the GUI how to depict the file
  - Unix kludge: magic numbers
    - Fixed header bytes at the start of a file
    - Not provided by the file system
    - Not centrally administered
    - `file` program can make guesses about file types based on well-known magic numbers
- User-defined metadata

### 100.1.4. Operations

- *creat, open, close, read, write, lseek, remove, truncate*
- *Locking?*

### 100.1.5. Access methods

- *Sequential: read/write the next block*
- *Direct: read/write block number N*
- *Distinction is usually weak in practice*
  - *Can fake sequential access given direct access by tracking position ourselves*
  - *Most sequential access implementations provide seek operation that can be used to simulate direct access*
- *WikiPedia: ISAM*

## 100.2. Directories

*Also called folders.*

- *Basic model: maps names to files*
  - *Usual hash-table operations: insert, delete, lookup, list*
- *Multiple directories: flat vs tree vs graph structure*
- *Can a file appear in more than one directory (or twice in the same directory)?*
  - *If so, who gets the metadata?*
    - *Associate most metadata with the file.*
    - *Name information goes in directory.*
  - *May require garbage collection*
- *Directories and naming*
  - *Root directories*
  - *Pathnames interpreted by the kernel*
  - *Relative vs absolute pathnames*
- *Symbolic links*
  - *Directory entry that refers to a pathname rather than a physical file*
  - *Used in Unix as work-around for limitations on hard links*
    - *Can symlink to a directory*
    - *Can symlink across devices*
  - *Problem: no guarantee that target exists*
  - *Interpretation is not necessarily fixed*
    - *E.g., Windows Vista allows remote symlinks to refer to local files*
    - *On Linux: `ln -s http://www.yale.edu/ yale-www`*
      - *This works, but filesystem doesn't know how to follow the link.*
      - *Nobody actually does this.*
  - *Many early OSes didn't have them and actively resisted including them*

## 100.3. Mount points

- *Idea is to combine multiple filesystems in a single directory tree.*
- *A filesystem is **mounted** on a **mount point***
  - *E.g. `mount /dev/hda2 /overflow`*
  - *Mount point is an existing directory in an already-mounted filesystem*
  - *Mount operation initializes new filesystem, sets up internal kernel data structures*
  - *Kernel redirects operations on this directory to new filesystem*
  - *Root file system must be handled specially*
- *Unmounting*
  - *Typically requires closing any open files on the filesystem*
  - *Flushes buffers and clears internal kernel data structures*

## 100.4. Special files

*In addition to symlinks and mount points, various other special files may appear in a filesystem.*

- *Device files e.g. `/dev/hda`, `/dev/console`: map pathnames to (major,minor) device number pairs.*
- *Named pipes (`mknifo` in Unix, pipefs filesystem in Windows)*
- *Unix-domain sockets*

*Mostly these allow using the standard file interface to access devices or other mechanisms directly, e.g.*

```
echo "now you're in trouble" > /dev/hda,1 wc /dev/mem.
```

## 100.5. Special file systems

*Nothing in the file system interface says that the files have to actually exist. Assuming the kernel's internal structure is flexible enough, this allows for many kinds of virtual file systems that extend the basic files-on-disk model. Some examples:*

### 100.5.1. `/proc`

*Originally developed for the experimental OS WikiPedia: Plan\_9\_from\_Bell\_Labs, but now present in most Unix-like OSes.*

The `/proc` filesystem contains a subdirectory for each running process and virtual files representing system information. Process subdirectories allow reading command lines, detecting the executing program, looking at open files (all subject to access control). System files allow reading system information, e.g. here is `/proc/cpuinfo` for `pine.cs.yale.edu`:

```
processor      : 0
vendor_id     : GenuineIntel
cpu family    : 15
model         : 4
model name    : Intel(R) Pentium(R) 4 CPU 3.80GHz
stepping      : 3
cpu MHz       : 3790.778
cache size    : 2048 KB
physical id   : 0
siblings      : 2
core id       : 0
cpu cores     : 1
fpu           : yes
fpu_exception : yes
cpuid level   : 5
wp            : yes
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
fxsr sse sse2 ss ht tm syscall nx lm constant_tsc pni monitor ds_cpl est tm2 cid cx16 xtpr
bogomips      : 7588.59
clflush size   : 64
cache_alignment : 128
address sizes  : 36 bits physical, 48 bits virtual
power management:

processor      : 1
vendor_id     : GenuineIntel
cpu family    : 15
model         : 4
model name    : Intel(R) Pentium(R) 4 CPU 3.80GHz
stepping      : 3
cpu MHz       : 3790.778
cache size    : 2048 KB
physical id   : 0
siblings      : 2
core id       : 0
cpu cores     : 1
fpu           : yes
fpu_exception : yes
cpuid level   : 5
wp            : yes
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx
fxsr sse sse2 ss ht tm syscall nx lm constant_tsc pni monitor ds_cpl est tm2 cid cx16 xtpr
bogomips      : 7581.31
clflush size   : 64
cache_alignment : 128
address sizes  : 36 bits physical, 48 bits virtual
power management:
```

None of these files actually exist as permanent stored objects; instead, when a process opens and reads them the data is generated on the fly by the `/proc` filesystem driver.

### 100.5.2. Archival file systems

E.g. Plan 9's `WikiPedia`: `Fossil`, allowing `cat /snapshot/2004/02/01/123000/home/user/important-document.txt`. Here the first few directory names are actually timestamps, with the interpretation of finding the most recent copy of `/home/user/important-document.txt` available as of 12:30:00 on 2004-02-01.

### 100.5.3. Distributed file systems

E.g. `NFS`, `SMB/CFIS`, `AFS`. Here files are stored on a remote server (possibly several remote servers). Local requests for file operations are redirected across the network. We'll talk more about these when we talk about `DistributedSystems`.

### 100.5.4. Unions

Some filesystems allow several directories (or several filesystems) to be mounted in the same place; this presents to the user the union of the contents of these directories, with typically a priority scheme to determine what to do in the case of a conflict.

### 100.5.5. Encrypted filesystems

(Exactly what they sound like.)

## 100.6. Access control

It's natural to limit users' access to particular files or directories in the filesystem, both for privacy on shared systems and to keep users from accidentally or maliciously damaging system components that happen to be stored in the filesystem. Any modern filesystem will provide at minimum some approximation to `POSIX` access control bits (the classic `rw-rw-rw-` inherited from `Unix`), where each bit controls whether the owner of the file, the members of a particular group of users associated with the file, or an arbitrary user can read/write/execute the file. Users are identified by user ids, which are set by the system at login time.

Additional bits are provided to support other features, like the **setuid bit** that allows a user to execute a file with the effective permissions of the owner or the **sticky bit** that allows only the owner of a file to remove it from an otherwise world-writable directory.



POSIX-style permissions are a compromise between generality and simplicity. One disadvantage is that it is difficult to grant fine-grained access to a file to specific users. If I want George and Martha to be able to edit one of my files, I can give them group access provided there is a group already defined in the system that contains just George and Martha (and possibly me). But defining groups typically requires special privileges not available to normal users.

Many systems thus extend the basic POSIX framework by adding **access control lists** or ACLs, specific lists of per-user or per-group permissions associated with individual files or directories that override the basic access control rules. Thus I can specifically grant George the ability to read or write my file while granting Martha only the ability to read it. Such features are required for certain security certifications, but are often left out in Unix-like systems. This has led to a lack of standardization of ACL interfaces; attempts to include access control lists in POSIX failed for precisely this reason. It also means that users don't expect to be able to use ACLs, which lets implementers off the hook for building them, an example of the chicken-and-egg problem that prevents new operating system features from being widely deployed unless they provide functionality much more useful or efficient than the default.

When the filesystem provides access control, this gives a natural way to provide access control to hardware devices that are themselves accessed through the filesystem. So for example we can turn off all the speakers in the Zoo machines by the simple expedient of turning off write access to `/dev/audio` and `/dev/dsp`; or, in a context where speaker use is less annoying, we might restrict access only to members of group `audio`.

## 101. Implementation

### 101.1. Disk layout

- **Master Boot Record (MBR)** sits at standard location, lists **partitions**.
- Each partition acts like a miniature disk with its own filesystem.
  - **Boot block** for bootable partitions
  - **Superblock** with key filesystem info
  - Remaining blocks allocated to directories, inodes, free lists, files.

#### 101.1.1. Block size

- Want to amortize cost of reading over many bytes
- Fixed-size blocks make life easier
- Trade-off on block size
  - Big blocks: lots of empty space
  - Small blocks: lots of pointers
- Frequently-used 4K block size fits nicely with typical file size.

#### 101.1.2. Tracking blocks in a file

##### Contiguous allocation

- Each file is stored as a sequence of consecutive blocks
- No need for indexing: just track first and last blocks
- Very good for fast sequential reads
- Very bad for fragmentation

##### Linked lists

- Each block stores a pointer to the next block.
- Doesn't work well for random access.
- Blocks end up with weird sizes, e.g.  $2^{12}-4 = 4092$  bytes, forcing filesystem to do division instead of bit operations.

##### File-allocation table

- Separate linked lists from files
- Still not very good for random access
  - But maybe OK if we store the whole FAT in memory
- FAT can be very big

##### Index nodes:

- Give each file an **inode**, an array of pointers to blocks
- Supports random access
- Good for memory: don't need to load inodes for closed files
- Not so good for short files (inodes will be mostly empty)
- Maybe not so good for big files: run out of space in the inode!

##### Multilayer Unix approach:

- Inode stores file attributes (but not the filename, which goes in a directory)
- Inode also stores pointers to first few blocks of file.
- 3rd to last pointer points to **single indirect block**, which points to blocks
- 2nd to last pointer points to **double indirect block**, which points to blocks that point to blocks
- Last pointer points to **triple indirect block**, which points to blocks that point to blocks that point to blocks. (After this we are out of address bits.)
- Still not too good for very short files
  - Inodes are in fixed location: can run out of inodes before running out of disk space

Fancier mechanisms:

- E.g. B+-tree in *Wikipedia: ReiserFS*

### **Log-structured filesystems**

- Will get their own page, see *LogStructuredFilesystem*

#### **101.1.3. Tracking free blocks**

The basic choice here is a linked list vs a bit vector. Bit vectors have the advantage of allowing smarter allocation (since the block allocator can see all the blocks that are free), but the disadvantage of requiring more space when there aren't many free blocks. Linked lists can be stored in any already-existing file-allocation table if we have one, or can be stored in the free blocks themselves.

A tempting solution that doesn't work so well is to allocate all free blocks to one giant dummy file; the problem is that we are likely to have to go through several layers of indirection (and thus multiple slow disk accesses) to find a free block.

#### **101.1.4. Directories**

- Table mapping names to inodes/FAT entries/other directories.
- May also contain file attributes.
- Implementation issues
  - Filename length
    - Fixed length: makes for simple structure, but wastes space
    - Variable length: often implemented as ISAM structure with pointers into packed table of names
    - Hybrid approach (e.g. as used in Windows FAT): store short names in main table, but allow pointers to extensions stored elsewhere if appropriate bit is set.
      - Largely a retrofit in FAT, but has led to entertaining patent battles.
  - Data structure
    - Classic approach: unsorted array
      - Low overhead
      - Slow for big directories
    - Better approaches: hashing, B-trees

#### **101.1.5. Physical layout**

- To be fast, must respect disk geometry
- Berkeley Fast File System
  - Cylinder groups
    - Contain backup superblock, inodes, file blocks
    - Inodes and related file blocks are contained to a cylinder group if possible
- Windows
  - Move frequently-accessed files to middle of the disk

### **101.2. Pathname translation**

- Root directory (typically a mount point stored in mount table)
- Each pathname component traverses one more directory
- Mount points are intercepted before examining corresponding directory
- Virtual file systems may take over pathname translation and do their own nastiness

### **101.3. Caching**

- Maintain pool of cached disk blocks
  - Typically done below filesystem level
- Consistency issues
  - Flush important blocks first (e.g. superblock, inodes)
  - Write-through: flush all dirty blocks immediately
    - MS-DOS approach
    - Maintains consistency with removable media
  - Sync: flush all dirty blocks from time to time
    - Unix approach
    - `sync` system call called by user, or periodically by `update` daemon
    - Guarantees all data is updated after at most  $k$  seconds for some small  $k$ .
    - Doesn't guarantee consistency if user can remove the disk or if the power fails.
  - Log-structure filesystems (we'll come back to these)

## **102. Consistency checking**

- Filesystems are subject to bit rot.
- Consistency checker (`fsck`, `chkdisk`) fixes inconsistencies
  - Examples of inconsistencies:
    - Same block in multiple files
    - Blocks not in free list
    - Directory pointers to nowhere
  - Often run at boot time (interacts badly with mounted disks)
  - Similar to garbage-collection

- Note: only guarantees consistency, doesn't guarantee you get your data back

## 102.1. How inconsistencies arise

- Rare: media failure
- More common: disk operations are not atomic
  - E.g. consider deleting a file:
    - Remove directory entry
    - Add blocks to free list
    - Update statistics in superblock
  - Failure during this process can leave inconsistent state

## 102.2. Recovering from inconsistencies

- Run `fsck`
  - Essentially a garbage collector
  - Detects inaccessible blocks that aren't in free list
  - Detects accessible blocks that are in free list
  - Puts orphaned files in some standard location (e.g. `/lost+found`).
  - Checks other file attributes
    - E.g. bizarre modification times

## 102.3. Preventing inconsistencies

- Be careful about order of disk operations
  - E.g. in deleting a file:
    - Update directory entry first
    - If we get a failure, worst that happens is a storage leak
  - E.g. expanding a file:
    - Update free list
    - Then write new blocks
    - Then update inode/indirect blocks
  - May conflict with disk scheduling optimization
  - **Soft updates**
    - Allow disk scheduler freedom to order writes for operations on different files, but enforce consistency for individual files
  - Not perfect if disk hardware reorders writes
- Enforce atomicity:
  - Journaling
  - `LogStructuredFilesystem`

## 103. More info

There is a detailed on-line book on filesystem implementation at <http://www.letterp.com/~dbg/practical-file-system-design.pdf>. (Thanks to Philip Taff for pointing this out.)

---

CategoryOperatingSystemsNotes

## 104. LogStructuredFilesystem

A **log-structured file system** is a method for organizing blocks in a filesystem so that writes are always appended to the end of the filesystem. This has several advantages over traditional approaches:

- We get **journaling** automatically: all file operations are effectively **atomic**, reducing the need for consistency checking.
- We may get **versioning** as a bonus: old versions of files are still present in the log, so we can undo mistakes and/or recover old versions of files.
- For write-intensive applications, the fact that writes all go to the same location may improve write performance substantially (by an order of magnitude in some cases).
- There is no free space fragmentation.

On the other hand, we do pay a price:

- There is quite a bit of data fragmentation, since updating a few blocks in a file places the new copies at the end of the log, wherever it happens to be.
- Some sort of garbage-collection processes is necessary to recover free space from the beginning of the log.

Log-structured filesystems were proposed by Rosenblum and Osterhout in a 1991 SOSP paper (See [lfsSOSP91.ps](#)). To date they have not been widely adopted, although some of the ideas have shown up elsewhere.

## 105. The first step: journaling

Suppose we want to guarantee that file operations maintain consistency even in the presence of power failures, sudden media removal, etc. One approach, derived from techniques used for databases, is to maintain a **journal** of pending actions. So now if we want to update a file

(say by adding a block to the end of it), we

1. Write an entry in the journal describing the change,
2. Implement the change in the main filesystem, and
3. Mark the journal entry as completed, so that the space it occupies can eventually be reused.

Suppose now we pull the plug on the computer somewhere in the middle of this process. At recovery time, instead of scanning the entire disk to look for inconsistencies, we can just look at the uncompleted journal entries and carry out whichever ones have not actually taken effect in the main filesystem. (Any partial journal entries are ignored.) This is the approach, for example, taken in the default Linux ext3 filesystem.

By adopting this approach, we can guarantee consistency with minimal recovery time, but we pay a high price: since we have to write an update to the journal before writing it to disk, every change we make gets written twice. In practice, this problem is usually dealt with by limiting journaling to metadata: directory structure, free lists, etc., but not the actual contents of files. This is a compromise that greatly reduces the size of the journal (and thus the amount of writes that need to be done to maintain it) while still protecting the consistency of the filesystem's internal data structures. However, it allows for the possibility that user data is invisibly corrupted (consider what happens if we journal adding a block to the end of a file before we actually write the block). Some of this corruption can be avoided by carefully scheduling the order of disk operations, but this may conflict with other disk scheduling goals (like not moving the head too much).

## 106. Ditching the main filesystem

The key idea of Rosenblum and Osterhout's paper was to go one step further and get rid of the second step of updating the main filesystem; indeed, they drop the main filesystem completely, by folding it into the journal. In order to avoid having to reimplement too much, they keep the overall structure (superblock, free list, inodes, indirect blocks, blocks) of the Berkeley Fast File System, but most of these blocks are no longer updated in place but instead are appended to the end of the log in large multi-block segments that can be written very quickly in bulk. This means that particular filesystem elements like inodes may appear in several versions in the log, but only the last version counts. It also means that we need a mechanism to track down blocks that previously lived at fixed locations (e.g. the inode table) but that are now scattered throughout the log.

Below we will describe the specific mechanisms used in SpriteFS, the filesystem described in the paper. Other approaches are possible.

### 106.1. The inode map

As always, these problems are solved by adding more indirection. An **inode map** gives the current location of all the inodes. This inode map is itself stored in the log, and updates to the inode map require writing new versions of inode map blocks.

### 106.2. Checkpoints

So how do we find the inode map? SpriteFS uses a **checkpoint region** at a fixed location on the disk (specified in the superblock, which is also fixed). This checkpoint region contains pointers to the most recent blocks in the inode map and to the front of checkpointed portion of the log. It does not need to be updated after every log operation; it is possible during recovery to replay any part of the log that extends beyond the last checkpointed entry. So the checkpoint region acts primarily as a backup copy of the real data kept in memory.

In the worst case, if the checkpoint region is lost or damaged, it is possible to recover it from a full scan of the log.

## 107. Space recovery

With an infinitely large disk, we don't have to worry about recovering space. But we don't have infinitely large disks. So a log-structured filesystem requires a mechanism for recovering free space from old segments at the start of the log. An approach that was considered but discarded by Rosenblum and Osterhout was threading the log through free space within early segments: turning the log into a linked list that overlaps itself in physical block locations. The disadvantage is that the log becomes fragmented and it is no longer possible to use bulk writes. Instead, SpriteFS adopts the approach of **cleaning** old segments by copying live data to the end of the log and then reclaiming the entire segment.

### 107.1. Segment summary data

To facilitate this process, each segment contains a summary header that describe which blocks belong to which versions of which inodes. This allows the cleaner to quickly detect out-of-date blocks (since it can check to see if the corresponding inodes are current by looking at the in-memory inode map).

### 107.2. Data compaction

Live blocks in old segments are copied to the front of the log. There is an opportunity here for **data compaction**: blocks from the same file can be sorted together to increase locality for later read access. This works especially well if the segment cleaner can process many old segments at once, since it increases the opportunity to sort blocks together.

### 107.3. Which segments to clean?

The segment cleaner can be selective about which segments it attempts to clean. Cleaning a segment that consists of mostly live data won't gain us much space. So SpriteFS adopts a policy of cleaning segments that are mostly junk first (if there aren't any, the disk is probably full). This means that segments are not necessarily cleaned in the order they are generated, which slightly complicates the process of choosing segments to clean. But it also means that static data doesn't add much to the cost of cleaning, making cleaning costs close to a linear function of update costs. The effect is similar to generational garbage collection strategies in programming language runtimes, where static data is considered less and less often by the GC system.

## 108. Performance

Log-structured filesystems assume that write performance is more of a constraint than read performance. The justification for this assumption is that read performance can be improved by increasing cache sizes. For write-heavy loads on disks that aren't too full, a log-structured filesystem produces much faster performance than a traditional filesystem because it avoids seeks between block writes—this is even enough to pay for the overhead of segment cleaning and maintaining duplicate log entries. Read performance as observed by Rosenblum and Osterhout was comparable to that for a traditional filesystem. However, for very full disks (requiring frequent cleaning) or for read-heavy loads that could otherwise take advantage of locality, a traditional filesystem would give lower overhead.

For media (like flash drives) that don't have to worry about moving around physical heads, the advantages of doing bulk writes largely evaporate. However, there may still be an incentive to adopt a log-structured approach here since (a) it evenly distributes writes across the medium, which can be an issue for many solid-state devices, (b) it makes it easier to pack variable-sized blocks together, and (c) it allows for the possibility of on-the-fly compression (which otherwise would produce lots of internally fragmented partial blocks). So paradoxically log-structured filesystems like Wikipedia: JFFS are currently more likely to be found on flash drives than on hard drives.

## 109. Why aren't all filesystems log-structured?

Short version: Journaling is good enough for most purposes while maintaining backward compatibility: throw away the journal and you still have a familiar filesystem. E.g. Linux ext2 -> ext3 transition.

---

CategoryOperatingSystemsNotes

## 110. Networking

We'll talk about **computer networks** from an OS perspective.

## 111. Local-area vs wide-area networks

A **local-area network** or **LAN** connects machines in the same room or building. A **wide-area network** or **WAN** connects machines on the same planet. Both provide a mechanism for doing message-passing between computers (see *InterProcessCommunication*); the main difference from the OS view is speed and possibly reliability.

### 111.1. LANs

For a LAN, the connection between two machines is likely to be either direct or through a single **switch** or **hub**. Most LAN transports now are implemented using radio signals, either sent through the air as in 802.11 or along twisted pairs of wires as in Ethernet. Networking hardware takes responsibility for transmitting **packets** generated by the participating machines and detecting **collisions**, which are attempts by two machines to transmit at the same time. Lost packets are retransmitted, with increasing random delays to avoid congestion.

There are various configuration details that may complicate this picture. Although Ethernet is designed to allow computers to be connected directly across a single shared line, it is common to instead connect machines to a **hub** (dumb version that echoes packets to everybody) or **switch** (smart version that echoes packets only to intended recipients); the hub or switch may also act as a **gateway** that passes packets between separate LANs or to a larger WAN. Gateways are particularly necessary for LANs built from multiple transports; for example, a given local-area network may include both wired and wireless connections, and packets from the wired side must be translated through a gateway to the wireless side and vice versa.

Speeds on local area networks can be very high, with bandwidth ranging from 1 Mib/sec to 54 Mib/sec for wireless connections to 10 Mib/sec to 1 Gib/sec for wired connections using current consumer-grade hardware (that's bits, not bytes; to get bytes, divide by 8). Latency is typically on the order of a few hundred microseconds. Note that bandwidth is usually shared between multiple computers and applications, which can reduce the maximum effective bandwidth substantially.

### 111.2. WANs

When our pile of gateways and hubs become large enough, we have to start exercising still more intelligence in where we send packets, and our gateways/switches/etc. graduate to being **routers**. A router may be connected to many transports, and has a **routing table** that tells it where to forward packets based on their destination addresses (see below). In a wide-area network, a packet may be forwarded through dozens of routers and intermediate transports before reaching its destination.

Speeds on wide-area networks are limited by the effective bandwidth of the slowest link and delays incurred in translating and forwarding packets between transports. Even though the transports used for the core of a WAN may have very high bandwidth (e.g. 14 Tib/sec for long-distance optical fiber), this bandwidth is typically divided between many packets, so the actual bandwidth available to a single application is likely to be much more limited—and proportional to what the user is willing to pay for. Consumer and business connections typically involve running a line to an **Internet Service Provider** or **ISP** (often multiple ISPs in some cases for businesses that worry about reliability or maintaining competition among their suppliers). The capacity of this line is generally capped by the limits of the transport medium or by a contract between the purchaser and the ISP. Bandwidths here may range from 56 kib/sec or worse for modem connections across the telephone network to 1-2 Mib/sec for DSL connections, 6-12 Mib/sec for cable modem connections, or 100 Mib/sec and up for fiber optic connections (largely legendary in the US, but rumored to be big in Japan). Actual effective bandwidths are likely to be lower because of congestion later in the network, and because of asymmetries in the transport mechanism; for example, typical upload bandwidths on DSL or cable modem lines are an order of magnitude smaller than download bandwidths.

Latency for wide-area networks is necessarily much higher than for local-area networks, both because of delays inside routers and because of fundamental limitations of physics: a signal traveling 20,000 kilometers around the surface of the earth will take approximately 70 ms to arrive even traveling at the speed of light; this is 2-3 orders of magnitude slower than we would expect from a signal traveling across a local-area network, where travel time is much less of an issue. Signals going through roundabout routes (e.g. via satellite transmission) will

be even slower. The difference in speed between LANs and WANs has similar effects to the difference in speed between various levels of the memory hierarchy, and encourages the use of similar tools like caching to work around it.

## 112. Addressing: IP

How do we know where to send a packet to? At the bottom layer, every wired or wireless Ethernet card ever built has a unique 48-bit MAC address that is baked into by its manufacturer (different manufacturers get different prefixes of this address space); for local-area networks, this address is included in the header of each packet and polite Ethernet cards drop packets that aren't addressed to them.

For wide-area networks, pretty much every network in the world now uses **Internet Protocol**, in either its "Classic" IPv4 or "New Coke" IPv6 incarnations, to do addressing and routing. IPv4 provides 32-bit address (the classic 128.36.29.174 style addresses you've probably seen before); IPv6 provides a much larger 128-bit address space (e.g. fe80::213:72ff:fe07:b068/64, an abbreviated address written mostly in hexadecimal with blocks of zeros omitted).

IP packets consist of some small number of bytes (the **payload**) prefixed with a short header containing routing information. This information is essentially just the source and destination addresses, together with a checksum (for the header only) and some status flags. A router or machine attempting to deliver an IP packet will translate the address using its routing table (which must be initialized somehow, often directly by a human being or using some distributed route-discovery algorithm) and send the packet out across the appropriate transport. Here's a simple routing table for one of the Zoo machines: ignoring the two middle lines, it says to send any packets for the 128.36.232 subnet out across the Ethernet and send any other packets to anger.net.yale.edu (the router for the CS department) so that anger can figure out where to send them next.

```
$ route
Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
128.36.232.0    *                255.255.255.0   U        0      0        0 eth0
link-local      *                255.255.0.0     U        0      0        0 eth0
loopback        *                255.0.0.0       U        0      0        0 lo
default         anger.net.yale.  0.0.0.0         UG       0      0        0 eth0
```

Assuming that the network engineers did their job right, each packet is eventually forwarded to its correct destination. Or not: IP only guarantees best effort delivery, meaning that a router is perfectly entitled to drop, reorder, or duplicate packets if it gets overwhelmed, and transport links are perfectly entitled to damage the data inside packets in various ways if it would be more expensive to avoid such damage. (Except for the possibility of duplication, this is similar in many ways to the situation with physical mail delivery.) IP also only delivers packets to a specific machine, and not to any particular service or process running on that machine. So most applications use higher-level protocols built on top of IP.

## 113. UDP and TCP

The two most common protocols built on top of IP are **UDP** (User Datagram Protocol) and **TCP** (Transmission Control Protocol). These add their own headers (which IP thinks of as part of its payload) that can be used to provide a saner interface to the user.

For UDP packets, a checksum stored in the header on the UDP payload detects most accidental corruption of the data (but not all—it's a pretty weak checksum). The UDP header also provides 16-bit source and destination **ports**, which are used to distinguish different processes or services at the endpoints. Finally, UDP packets are guaranteed to be delivered in one piece or not at all; this is an improvement in theory on IP packets, which can be fragmented if necessary for transport media that can't handle large packets, but is not necessarily an improvement in practice since a common way of enforcing this condition is to discard over-large UDP packets. UDP packets do not add any more reliability; like IP packets, UDP packets may be dropped, reordered, or duplicated, and are only useful for applications that can tolerate such abuse. (If you are curious, you can find the specification of UDP at <http://tools.ietf.org/html/rfc768>.) The selling point of UDP packets is that they are lightweight—no state needs to be maintained within an operating system to handle sending or receiving UDP packets (except perhaps to record which process is interested in receiving packets on which port).

TCP packets add a much larger header that includes sequencing information used to simulate reliable, FIFO delivery of data. The TCP protocol describes elaborate mechanisms for acknowledging successful packet delivery and retransmitting packets that fail to be delivered in a sufficient amount of time. Lost packets are further used as a signaling mechanism to control transmission rates; by dropping excess TCP packets, an overloaded router will cause polite TCP stacks to cut back their rate and resolve the overload (there is a Wikipedia: tragedy\_of\_the\_commons situation here if TCP stacks become impolite). The interface presented to the user is one of two-way, byte-oriented stream connections essentially identical to local pipes. So for example, TCP connections can be accessed directly by C's `stdio` library; after opening a TCP connection to a remote host, you can run `fscanf` or `fprintf` on it just as you would on any open file.

Reliability is still an issue for TCP connections, but it is now all-or-nothing: so long as the TCP connection stays up, bytes are delivered without corruption and in the correct order, but if the connection drops (because packet loss or corruption gets so bad that acknowledgments can't get through within the maximum timeout), some suffix of the transmitted data may be lost. It is generally not possible to detect exactly which bytes get through and which are lost, which creates some interesting theoretical problems for things like purchasing goods over the Internet exactly once, which in practice doesn't seem to come up except on really low-budget small business websites.

TCP has evolved over the years and is now specified by a number of RFCs; the page Wikipedia: TCP gives a list of some of the more important ones. The basic guts of TCP are in <http://tools.ietf.org/html/rfc793>.

### 113.1. TCP: more details

The basic mechanism for ensuring in-order packet delivery is the use of a 32-bit **sequence number** on each packet. The sequence number specifies the position in the stream of the first byte in the packet; so sending a 100-byte packet increases the sequence number for the next packet by 100. (The reason for labeling bytes rather than packets is that it allows retransmitted packets to be consolidated or split.) Packets traveling in the other direction contain an **acknowledgment number** that indicates the next byte that the receiver expects to receive. When the sender emits a packet it places a copy in a local queue, which is retransmitted after a timeout unless an acknowledgment is received first. For streams with high traffic in both directions, no extra packets are needed for acknowledgments; instead, the acknowledgments are

piggy-backed on data packets. If there is no outgoing data, acknowledgments can be sent using otherwise empty **ACK** packets. Note that sequence numbers for each direction of the connection are independent of each other.

### 113.1.1. Connection setup

In order to use a TCP connection, the connection must first be established using a three-step handshake protocol. The main purpose of this protocol is to initialize the sequence numbers. Because a TCP connection is identified only by the source and destination addresses and source and destination ports, it is possible that a sender may attempt to re-establish a TCP connection that was previously closed. While this is not a problem in general, it does mean that we have to be careful to make sure any old packets from the previous connection don't cause trouble with the new connection (e.g. by getting delivered as part of the new stream or by being treated as acknowledgments for data that hasn't actually gone through). So a connection initiator chooses a new starting sequence number that must be greater than all previous sequence numbers and sends it to the receiver using a special **SYN** (synchronization) packet. The receiver chooses its own starting sequence number (again subject to the constraint of avoiding reuse) and sends it back in a **SYN-ACK** packet. A final **SYN-ACK-ACK** packet from the sender (actually just an ordinary **ACK** packet) acknowledges receipt of the **SYN-ACK**; at this point the connection is established and either end can send data packets.

Because **SYN**, **SYN-ACK**, and **ACK** packets can all be lost, timeouts are used to retransmit them just as with regular data packets.

One complication with connection startup is that it is possible that the sender or receiver has no idea whatsoever what packets might still be floating around; for example, a host might have just brought up its network interface after recovering from a crash that wipes its memory. There is essentially nothing that can be done about this other than waiting; TCP defines a **maximum segment lifetime** of two minutes, and demands that the underlying network not deliver any TCP packet older than this. This means (a) that a crashed machine can't establish any new connections for at least two minutes, and (b) that you can't use TCP to talk to anybody more than 120 light-seconds away, or across communication links with particularly long delays.

### 113.1.2. Receive window

One of the goals of TCP is to allow data to be shipped in bulk with low overhead. So TCP defines a **receive window**, which is the amount of data a sender is allowed to send before receiving an acknowledgment. This window is under control of the receiver and is included in the TCP header of each outgoing packet, allowing the receiver to dynamically adjust the window as needed; the trade-off is that large windows allow for higher throughput (particularly on high-latency connections) but may require more buffering at the receiver. The original TCP specification allowed window sizes between 2 and  $2^{16}$  bytes; a later extension (negotiated during the setup handshake) allows window sizes up to  $2^{30}$  bytes. One downside of using a very large window is that it is possible to lose data early in a stream causing all subsequent packets to be retransmitted (since the acknowledgment number only indicates the last byte successfully received, and not whether subsequent packets need to be retransmitted). Further extensions allow **selective acknowledgment**, which uses extensions to the basic TCP header to allow acknowledging particular ranges of packets.

### 113.1.3. Retransmission control

Retransmission is controlled by setting a **retransmission timeout**. An ideal retransmission timeout for maximizing throughput will be slightly larger than the typical **round-trip time** for the network connection, since this the first time at which the sender can conclude that a packet has in fact been lost. The round-trip time can be estimated by the delay between when a packet is first sent and when it is acknowledged. More sophisticated mechanisms can adjust the round-trip timeout to achieve better congestion control by responding to packet loss; how to do this in a way that optimizes throughput while minimizing congestion (i.e. overloaded routers tossing packets), maintaining fairness (each stream gets a fair share of the underlying pipes), and being compatible with previous widely-deployed mechanisms is an area of continuing active research.

### 113.1.4. Shutting down a connection

There are two ways to shut down a TCP connection:

#### Failure

If a sender doesn't get an acknowledgment after retransmitting a packet too many times, it gives up and sends a **FIN** packet, which initiates the standard shutdown procedure described below.

#### Shutdown

A sender can send a **CLOSE** packet, the TCP equivalent of EOF, to indicate it has no more data. If both endpoints send a **CLOSE** packet, this initiates a shutdown handshake using **FIN** packets (which must in turn be acknowledged). A particularly impatient sender can also send a **FIN** packet without **CLOSE** to hang up on a remote receiver who just won't shut up.

If the shutdown procedure works, both endpoints can clear out the connection (although they may need to retain some state for the connection for 2 MSL intervals to handle delayed **FIN** acknowledgments).

## 114. Higher-level protocols

Higher-level protocols (e.g. HTTP, SMTP, SSH, NFS) are usually built on top of UDP or TCP. This may involve defining yet another header-and-payload structure, so that for example an HTTP request might consist of a packet with several layers of headers, all of which need to be unwrapped to process it:

Ethernet frame header | IP header | TCP header | HTTP request header | HTTP request data

Protocols implemented on top of HTTP (e.g. XML-RPC or SOAP) will contain even more data.

## 115. OS implications

An OS can treat most of the network as one gigantic blob of communication sitting on the other side of its network hardware, and doesn't need to worry too much about the specific details of routing protocols. However, it at minimum needs to provide a device driver for its



network hardware and a dispatch facility that delivers incoming packets to the appropriate processes, and most likely will provide a kernel-level implementation of the TCP protocol to maximize performance.

Some local routing may also be the responsibility of the OS; for example, to route IP packets across a local Ethernet it is necessary to translate IP addresses to Ethernet addresses using the **address resolution protocol** (ARP), which essentially involves broadcasting the message "Hey! Who is 128.36.229.174?" and waiting for somebody to pipe up. For efficiency these ARP responses are typically cached (so we don't need to keep asking) for limited times (in case 128.36.229.174 moves or changes its MAC address), which implies all the usual overhead and maintenance of any cache system.

Back in the days of 300 baud modems, the network subsystem did not have to be terribly efficient, since the network itself acted as the main performance bottleneck (this is still somewhat true in the modern era of distant, underpowered web servers). But for high-speed local-area networks running at speeds only a few orders of magnitude slower than main memory, bad design choices in the OS can significantly cut into network performance, which in turn can produce visible, painful slowdowns for users of e.g. remote filesystems. The main design principles here are similar to those for any other I/O: to the maximum extent possible, avoid context switches (arguing against, for example, having a microkernel-style "networking daemon" that processes all network traffic) and avoid copying data unnecessarily. This last design goal is met for very high-performance systems by mapping the internal buffers in network hardware directly into the address space of the sending or receiving processes, so that data is never copied; but more typical systems accept one layer of copying by the kernel between the network hardware and the user process's address space.

## 116. Socket interface

Essentially all modern OSes provide access to networking through a standard API known as the **Berkeley socket interface**. Historically, this was the networking interface developed for BSD Unix, which spread to other Unices and Unix-alikes as well as Windows along with the Berkeley TCP stack that implemented it after the TCP stack became free for general use in 1989.

A **socket** is an abstraction of an endpoint of a network connection or service as a file. Berkeley sockets are implemented using several different system calls, which between them allow very detailed control over the construction and handling of sockets representing TCP streams, servers capable of accepting TCP connections, servers capable of accepting incoming UDP packets, etc. This control is provided at a very low level; using the socket interface to create a TCP connection often feels like building a car out of individual Lego blocks, and is suspiciously similar to the setup and teardown sequences for TCP connections. Part of the reason for this may be that the C calling syntax (especially in the form of system calls) limits the ability to specify default or conditional arguments, so that basic socket setup operations are handled by initial system calls while later specialized operations (which would probably be methods on socket objects or effects of subclassing in an object-oriented language) require subsequent specialized system calls.

Some of the major system calls provided by the socket API are:

### **socket**

Used to create a socket. Arguments specify a domain (e.g. IPv4, IPv6, various more exotic options), type (e.g. sequential vs datagrams) and protocol (e.g. UDP vs TCP); there is a certain amount of overlap between then last two arguments. Unlike file descriptors returned by `open`, newly-created sockets aren't really attached to anything, and with a few exceptions (sending UDP packets using `sendto`) can't do much.

### **bind**

Binds a socket to a local address. This is used to specify what address and port a server will listen for incoming connections on, or what port a remote recipient will see for packets originating from this socket. It is still not enough to actually allow you to send or receive data on TCP connections. For that, you need to do one of the following two system calls, depending on whether you want to be a **server** (accepting incoming connections) or a **client** (connecting to some remote server).

### **listen and accept**

For servers, the `listen` system call declares that we will accept incoming connections, which we can then turn into new sockets representing our end of the TCP connection using still another system call `accept`.

### **connect**

Attempt to connect to a remote server. If successful (i.e. the remote serving is listening and executes `accept`), the socket becomes one endpoint of a TCP connection.

### **shutdown**

Like `close` for sockets; politely signals the other end of our connection that we want to leave. Of course, we still have to call `close` after the shutdown succeeds (or fails, in case of network trouble).

You can read the details of each of these system calls by typing `man 2 socket`, `man 2 bind`, etc. The 2 is important, since similarly-named (and often more useful) functions show up elsewhere in the Unix manual pages.

Many of these functions require highly-processed and magical address arguments that are very different from the nice clean addresses humans are used to. So there is a whole family of additional library routines for translating human-readable names like `www.google.com` to the low-level crap that the socket library demands. See for example `getaddrinfo`.

Sensible people generally don't use any of these calls directly unless they are implementing some higher-level client or server library. Instead, they use higher-level client or server libraries already implemented by somebody else.

## 117. Effect of failures

Failures create fundamental problems for applications running across machines, which can be mitigated by sufficiently sophisticated protocols but cannot be avoided outright. For example, the TwoGenerals problem means that it is impossible to guarantee with an unreliable network that both the sender and receiver of a packet agree in all circumstances whether it has been delivered. Failures at the machine level can create even bigger headaches, allowing communication protocols to be driven to arbitrary states (Jayaram and Varghese, Crash failures can drive protocols to arbitrary states, PODC 1996; <http://portal.acm.org/citation.cfm?id=248052.248104>) in the worst case, and rendering a wide variety of perfectly ordinary tasks impossible in theory (see CS425/Notes). In practice most of these problems can be worked around with enough use of timeouts and redundancy, if we are willing to accept a small probability of failures at the application level.



## 118. Network File Systems

A **network file system**, **distribute file system**, or **remote file system** allows files stored on a remote server machine to be accessed as part of the filesystem of a local client machine. In order to do this, the local filesystem must translate what would otherwise appear to be local file operations into remote procedure calls working on the remote filesystem. There are two network file systems in widespread use today: **NFS**, originally developed by Sun, which is the dominant system in Unix and Linux systems, and **CIFS** or **SMB** for Windows systems. We'll mostly talk about the development and features of NFS, and then talk about some less dominant systems that provide other interesting features like detached operation.

For more information on distributed file systems in general see SGG Chapter 17; this also includes a very nice discussion of AFS, a particularly sophisticated distributed file system that never really caught on because it wasn't free. The original design considerations for NFS in particular are documented in [nsf-design.pdf](#).

## 119. Naming issues

Given an existing hierarchical file system, the simplest approach to naming remote files is to mount the remote filesystem as a subdirectory of the local filesystem. So, for example, in the Zoo machines `/home` is a mount point for `artemis.cs.yale.edu:/home`, an NFS file system exported by `artemis.cs.yale.edu`, the Zoo fileserver. A goal in adopting this approach (as discussed, for example, in the NFS design paper) is to obtain **location transparency**: a local process shouldn't have to know what machine actually stores the files it is using, so it can freely refer to `/home/some-user/.bash_profile` or some such without making any adjustments for remote access.

The price for this location transparency is that remote file systems must be explicitly mounted (e.g. at boot time based on being listed in `/etc/fstab`), so that the local machine can translate addresses as needed. This limits use to servers that are known in advance, and is very different from the distributed-file-system-like functionality of HTTP, where a user can talk to any server that they like. An alternative approach is to encode the remote server address directly in the pathname, e.g. `/@remote.server.com/usr/local/bin/hello-world` or `/http/pine.cs.yale.edu/pinewiki/422/Schedule`, and rely on symbolic links or similar mechanisms to hide the location dependence. It is not clear why the NFS approach of requiring explicit mounting came to dominate in the Unix world; perhaps it was because of a combination of the security dangers of eroding the separation between local resources (or at least resources on a server under the control of the local system administrators) and remote resources, and the problems that arise when programs that expect to be able to read files without errors encounter the misbehavior of resources accessed across a WAN.

## 120. Caching and consistency

Caching is less of an issue for distributed file systems running across a LAN than one might think: the bottleneck in the filesystem is likely to be the disk rather than the intervening network, so assuming the network stays up there is not much of an incentive to cache files on a local disk to improve performance. However, **consistency** of in-memory caches may be an issue: since the client machine can't necessarily see what changes are being made to a remote file, it can't easily guarantee that any data it has cached in memory will be up to date.

As with other problems in operating systems, there are a range of solutions to this problem. The simplest approach is to either avoid caching data locally (beyond minimal buffering inside `stdio` or similar libraries) or accept that local data may be out of date. A more sophisticated approach requires running some sort of cache-consistency protocol, either by having the client query the server for updates on each access or by having the server send callbacks to clients when data changes. The client-initiated model puts a lot of load on the server if reads are common but writes are few, and it's not clear that it is much faster than simply passing read operations through to the server unless network bandwidth is severely limited. The server-initiated approach will be less expensive when writes are rare, but requires that the server maintain state about what files each client has cached.

Changes in the semantics of the file system can affect cache costs. For example, the **write-on-close** policy of the Andrew File System (AFS) ensures that all updates to a file are consolidated into a single giant write operation—this means that the server only needs to notify interested clients once when the modified file is closed instead of after every write to a part of the file. The cost here is that it is no longer possible to have interleaved write operations on the same file by different clients.

The ultimate solution to consistency problems is typically locking. The history of NFS is illustrative here. Early versions of NFS did not provide any locking at all, punting the issue to separate lock servers (that were ultimately not widely deployed). Practical solutions involved the use of lockfiles based on the fact that creating a file under NFS always involved talking to the server and that Unix file semantics provided an `O_EXCL` operation to the `creat` call that would refuse to create a file that already existed. The problems with such ad-hoc solutions (mostly the need to rewrite any program that used locks to use lockfiles instead) eventually forced NFS to incorporate explicit support for Posix-style advisory `fcntl` locks.

## 121. Stateful vs stateless servers

The original distributed version of NFS (NFS version 2) used a **stateless** protocol in which the server didn't keep track of any information about clients or what files they were working on. This has a number of advantages:

### Scalability

Because the server knows nothing about clients, adding more clients consumes no resources on the server (although satisfying their increased requests may).

### Consistency

There is no possibility of inconsistency between client and server state, because there is no server state. This means that problems like TwoGenerals don't come up with a stateless server, and there is no need for a special recovery mechanism after a client or server crashes.

The problem with a stateless server is that it requires careful design of the protocol so that the clients can send all necessary information along with a request. So for example a Unix-style file descriptor that tracks a position in the file must be implemented at the client (since the

server has no state with which to track this position), and a client `write(file_descriptor, data, count)` operation on the client translates to a `write(file_handle, offset, count, data)` on the wire to the server. The inclusion of an explicit offset, and the translation of the local file descriptor to a **file handle** that contains enough information to uniquely identify the target file without a server-side mapping table, means that the server can satisfy this request without remembering it.

A second feature we want with a stateless server is **idempotence**: performing the same operation twice should have the same effect as performing it once. This allows a client to deal with lost messages, lost acknowledgments, or a crashed server in the same way: retransmit the original request and hope that it works this time. It is not hard to see that including offsets and explicit file handles gives us this property.


## 122. Data representation

An issue that arises for any network service but that is particularly tricky for filesystems is the issue of machine-independent data representation. Many of the values that will be sent across the network (e.g. file offsets or buffer size counts) are binary values that might (a) be stored in different widths by default on different machines, and (b) might be stored with different byte order or Wikipedia: Endianness. So an x86-based client talking to a PowerPC-based server will need to agree on the number of bytes (called **octets** in IETF RFC documents, to emphasize 8-bit bytes as opposed to the now-bizarre-seeming non-8-bit bytes that haunted the early history of computing) in each field of a data structure as well as the order in which they arrive.

The convention used in most older network services is to use a standard **network byte order** which is defined to be **big-endian** or most significant byte first. This means that a hypothetical x86 client will need to **byte swap** all of its integer values before sending them out. Such byte-swapping is usually done in a library routine, so that client or server writers don't need to know about the endianness of the target machine when writing code. If the target machine is already big-endian, the byte-swapping routine can be a no-op. (See for example `man ntohs` on any Unix or Linux machine.)

More recent services have tended to move to text-based representations. Byte order doesn't come up in HTTP because a content-length field is the ASCII string `Content-length: 23770` instead of a 4-byte integer value packed at position 17 in the some hypothetical binary-format HTTP response header. This approach has the advantage of making programmer error harder, at the cost of consuming more network bandwidth, since a decimal digit packs only about  $3\frac{1}{3}$  bits of information in an 8-bit byte. For things like HTTP headers that are attached to large documents, the additional cost in bandwidth is trivial. An extreme example of this is self-documenting XML-based encodings like XML-RPC and its successor SOAP.

## 123. Case study: NFS version 2

See  [nsf-design.pdf](#).

---

Category: Operating Systems Notes

## 124. Distributed Systems

There are three main ways to look at distributed systems, each of which provides a slightly different perspective from the others.

From a very theoretical perspective, we can build a model of a distributed system in terms of e.g. communicating automata and ask what tasks we can do in this model under various assumptions about scheduling, failures, and so forth. Much of the research in this area sits right on the edge of impossibility, where dropping a single beneficial assumption makes what you want to do impossible and adding a few more beneficial assumptions makes what you want to do easy (or at least not hard). (To find out more about this, take CS425.)

From a more practical perspective, we can say that we don't mind assuming properties of our system that the real world actually has (like timeouts or reliable clocks). This gives us a more useful model and allows the use of standard coordination algorithms of the sort described in SGG Chapter 18.

From a still more practical perspective, we can adopt the principle that "If we build this right, we won't need any algorithms"<sup>2</sup>, and aim for distributed systems that don't require much coordination or global consistency. Most of the distributed systems we actually use fall in this last category.

## 125. Examples of distributed systems

Here are the four main distributed systems normal people actually use:

1. The vast collection of routing mechanisms that make up the Internet. Most of these are very local, in the form of routing tables maintained by humans or by (fairly straightforward) route-discovery algorithms.
2. The **domain name system** (DNS). Essentially a large hierarchical database for translating domain names like `www.cs.yale.edu` into IP addresses like `128.36.229.30`. The underlying mechanism consists of the user propagating RPC-like queries up through a tree of nameservers, to find first the nameserver for `.edu`, then for `.yale.edu`, then for `.cs.yale.edu`, and then finally to find the target host. Fault-tolerance is obtained through replication, scalability through caching, and sound economics through carefully assigning most of the cost of a query to an organization that cares about the user getting the answer (i.e. Yale ITS runs the `.yale.edu` nameserver). Note that very little coordination is required: domain records don't change very fast, and can be published in a central location.
3. The World Wide Web. Structurally not all that different from the bottom layer of DNS, with web servers replacing nameservers.
4. The SMTP-based email system. This is to packet routing what the web is to DNS: a store-and-forward system implemented at a high level in the protocol stack rather than down in the routers. Reliability is important (TwoGenerals comes up here, as my mail server can't drop an outgoing message until it is sure that your mail server has picked it up), but for the most part there is no need to do any sort of global coordination or consistency enforcement.

The common pattern in each case: to build a successful Internet-scale distributed system, it appears to be that case that you need an application that doesn't require centralized coordination, that allows newcomers to join easily, and that scales in economic terms by lining up costs and benefits so that they are roughly balanced for most potential users. This also characterizes other large-scale distributed systems like UUCP (in the old days) or various peer-to-peer content distribution systems (more recently).

## 126. Distributed coordination

For some distributed systems, it's necessary to get more coordination between components. The classic example is banking: we want to ensure that if I send you \$100, both our banks agree that the transaction took place. Such coordination turns out to be surprisingly difficult if we are not careful.

In the absence of failures, the problem of distributed coordination is essentially just the problem of ConcurrencyControl in a distributed setting, and we can adapt the approaches we've already seen for multiprocessors, primarily mutual exclusion. But we need to be careful to make sure that whatever we use works when our only underlying communication mechanism is message passing. We also have to be careful in that we no longer have as much control over the ordering of events: where a write to a memory location can be treated as atomic operation (assuming our hardware is set up right), transmitting a message across a network necessarily takes time. This can lead to confusion between widely-distributed processors about when particular events happen, or even the order in which they happen.

## 127. Timestamps

We can clear up the confusion somewhat by assigning our own synthetic times, or **timestamps** to all events. We'd like this assignment to be consistent with what we observe: in particular, the timestamps of successive events at the same machine should be increasing (a property called **monotonicity**) and the timestamp at which a message is received should exceed the timestamp for when it is sent (**consistency**). One way to do this would be to use very carefully synchronized clocks. However, there is a simpler approach—known as a **logical clock**—that just uses counters.

Suppose each machine has a counter for the number of events it has processed; this counter always rises by 1 at each event, so we get the monotonicity property. Whenever we send a message, we attach the timestamp at the sender; the receiver updates its local clock to  $\max(\text{message\_timestamp}, \text{local\_clock}) + 1$ . This process also preserves monotonicity (since the new value of local clock is at least as big as the old value plus 1) but in addition gives us consistency (since the message is received later than it is sent). The only real downside of this system is that the local clocks may advance very quickly if some process goes nuts and starts sending out huge timestamps, but we can probably defend against this by dropping messages if the timestamps are implausibly large.

## 128. Distributed mutual exclusion

So now we'd like to block off exclusive access to some resource for some interval of time (where time is now a potentially very squishy logical time). There are several options:

### Centralized coordinator

To obtain a lock, a process sends a request message to the coordinator. The coordinator marks the lock as acquired and responds with a reply message. When the initial process is done with the lock, it releases it with a release message. Bad things happen if any of these messages are lost or either process fails (we can use retransmissions and timeouts to work around this). An advantage is that the coordinator can implement any scheduling algorithm it likes to ensure fairness or other desirable guarantees, and that only 3 messages are sent per entry into the critical section.

### Token passing

We give a unique token to some process initially. When that process is done with the lock (or if it didn't need to acquire it in the first place), it passes the token on to some other process. Repeat forever. Advantage: no central coordinator. Disadvantage: have to organize the processes into a ring to avoid starvation; token can be lost (or worse, duplicated!); if nobody needs the lock the token still spins through the ring. This is a still a pretty common algorithm for human collaborators not blessed with good distributed version control systems.

### Timestamp algorithm

This is similar to the ticket machine approach used in delicatessens (and bakeries, although that risks confusion with the similar Bakery algorithm for shared-memory). To enter a critical section, a process  $p$  generates a timestamp  $t$  and sends  $\text{request}(p, t)$  to all of the other processes in the system. Each process  $q$  sends back a  $\text{reply}(p, t)$  message provided (a)  $q$  is not already in a critical section, and (b) either  $q$  is idle (not attempting to enter a critical section) or  $q$  has a higher timestamp than  $p$ . If  $q$  doesn't send back a reply immediately, it queues  $p$ 's request until it can. This algorithm has the property of ensuring mutual exclusion since for each pair of conflicting machines  $p$  and  $q$ , the one with the smaller timestamp won't send a reply to the other until it is done. It guarantees deadlock-freedom and fairness because the process with the smallest timestamp always wins (in the absence of cheating, which we have to assume here). It uses  $2(n-1)$  messages per critical section, which is more expensive than a centralized approach, but could be cheaper than token-passing. The main difficulty with the algorithm is that it doesn't scale well: each process needs to know the identity of all the other processes, so it works best for small, stable groups.

## 129. Distributed transactions

If we are worried about failures, we may want to go beyond a simple mutual exclusion approach and provide actual **atomic transactions**. Here atomicity means that the transaction either occurs in full (i.e. every participant updates its local state) or not at all (no local state changes). Mutexes are not enough to guarantee this because the process holding the critical section might fail in the middle—and if this occurs, there may be no way to recover a consistent state even if we can break the lock and restore access to the underlying data.

Instead we need a **distributed commit protocol** that allows all the processors participating in a transaction to agree when the transaction has completed—if this protocol fails, we will do a **rollback** of each processor's state to what it was before the transaction started (this requires keeping enough history information around to do this).

The simplest distributed commit protocol is known as **two-phase commit** and uses a central coordinator. However, it does not require that the coordinator survive the full execution of the protocol to ensure atomicity (but recall that aborting the transaction ensures atomicity). It

assumes the existence of stable storage (e.g. disk drives) whose contents survive crashes.

The algorithm proceeds as follows, when committing a transaction  $T$  (see SGG §18.3.1 for more details).

#### Phase 1

1. Coordinator writes  $\text{prepare}(T)$  to its log.
2. Coordinator sends  $\text{prepare}(T)$  message to all the participants in  $T$ .
3. Each participant replies by writing  $\text{fail}(T)$  or  $\text{ready}(T)$  to its log.
4. Each participant then sends a message  $\text{fail}(T)$  or  $\text{ready}(T)$ .

#### Phase 2

1. Coordinator waits to receive replies from all participants or until a timeout expires.
2. If it gets  $\text{ready}(T)$  from every participant, it may **commit** the transaction by writing  $\text{commit}(T)$  to its log and sending  $\text{commit}(T)$  to all participants; otherwise, it writes and sends  $\text{abort}(T)$ .
3. Each participant records the message it received from the coordinator in its log. In the case of an abort, it also undoes any changes it made to its state as part of the transaction.

Failure of sites is handled through a recovery process. A non-coordinator can detect whether the transaction committed or aborted from its log entries except if the log contains only a  $\text{ready}(T)$  record. In this case it must either ask the coordinator what it did (assuming the coordinator has come up), or wait for some other site to tell it that it has a  $\text{commit}(T)$  or  $\text{abort}(T)$  record in its log.

Failure of the coordinator is trickier. Temporary failures are not too bad; the coordinator can consult its log when it recovers to decide if  $T$  was committed or not. Permanent failures are trickier. Here in the worst case each participant in  $T$  has a  $\text{ready}$  message only in its log, and it is impossible to detect whether the transaction committed without waiting for the coordinator to recover. It is possible to demonstrate theoretically that under certain plausible assumptions, any distributed commit protocol has this property, that the permanent failure of some process in an asynchronous system may cause the protocol itself to fail (this is the well-known Fischer-Lynch-Paterson impossibility result).

## 130. Agreement protocols

There are various ways to get around the FLP impossibility result; the most practical use mechanisms that in the theoretical literature are modeled as abstract *FailureDetectors* but that in practice tend to specifically involve using timeouts. The problem usually solved is the problem of **agreement**, where we want all the participants to agree on some value (the **agreement condition**) after some finite amount of time (the **termination condition**), where the value is one initially proposed by at least one of the participants (the **validity condition**). It is easy to see that if we have such an agreement protocol we can solve the distributed commit problem (run the protocol to agree on committing vs aborting). In some circumstances we may even be able to use an agreement protocol more directly to agree on what update was performed to the shared data structure.

There are many protocols for implementing agreement. The best practical protocol for systems with crash failures may be Paxos, a voting-based protocol due to Leslie Lamport. More sophisticated protocols are needed if nodes in the system can misbehave in an attempt to subvert the protocol, a condition known as a **Byzantine fault**. We won't talk about these in CS422, but you can read about them on the ByzantineAgreement page from CS425.

---

CategoryOperatingSystemsNotes

## 131. Paxos

For more up-to-date notes see <http://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf>.

The Paxos algorithm for consensus in a message-passing system was first described by Lamport in 1990 in a tech report that was widely considered to be a joke (see <http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos> for Lamport's description of the history). The algorithm was finally published in TOCS [Lamport, The part-time parliament](#), ACM Transactions on Computer Systems 16(2):133-169, 1998, and after the algorithm continued to be ignored, Lamport finally gave up and translated the results into readable English [Lamport, Paxos made simple](#), SIGACT News 32(4):18-25, 2001. It is now understood to be one of the most efficient practical algorithms for achieving consensus in a message-passing system with *FailureDetectors*, mechanisms that allow processes to give up on other stalled processes after some amount of time (which can't be done in a normal asynchronous system because giving up can be made to happen immediately by the adversary).

We will describe only the basic Paxos algorithm. The *WikiPedia*: *WikiPedia* article on Paxos gives a remarkably good survey of subsequent developments and applications.

## 132. The Paxos algorithm

The algorithm runs in a message-passing model with asynchrony and less than  $n/2$  crash failures (but not Byzantine failures, at least in the original algorithm). As always, we want to get agreement, validity, and termination. The Paxos algorithm itself is mostly concerned with guaranteeing agreement and validity while allowing for the possibility of termination if there is a long enough interval in which no process restarts the protocol.

Processes are classified as *proposers*, *accepters*, and *learners* (a single process may have all three roles). The idea is that a proposer attempts to ratify a proposed decision value (from an arbitrary input set) by collecting acceptances from a majority of the accepters, and this ratification is observed by the learners. Agreement is enforced by guaranteeing that only one proposal can get the votes of a majority of accepters, and validity follows from only allowing input values to be proposed. The tricky part is ensuring that we don't get deadlock when there are more than two proposals or when some of the processes fail. The intuition behind how this works is that any proposer can effectively restart the protocol by issuing a new proposal (thus dealing with lockups), and there is a procedure to release accepters from their old votes if we can prove that the old votes were for a value that won't be getting a majority any time soon.

To organize this vote-release process, we attach a distinct proposal number to each proposal. The safety properties of the algorithm don't depend on anything but the proposal numbers being distinct, but since higher numbers override lower numbers, to make progress we'll need them to increase over time. The simplest way to do this in practice is to make the proposal number be a timestamp plus the proposer's id to break ties. We could also have the proposer poll the other processes for the most recent proposal number they've seen and add 1 to it.

The revoting mechanism now works like this: before taking a vote, a proposer tests the waters by sending a  $\text{prepare}(n)$  message to all accepters where  $n$  is the proposal number. An accepter responds to this with a promise never to accept any proposal with a number less than  $n$  (so that old proposals don't suddenly get ratified) together with the highest-numbered proposal that the accepter has accepted (so that the proposer can substitute this value for its own, in case the previous value was in fact ratified). If the proposer receives a response from a majority of the accepters, the proposer then does a second phase of voting where it sends an  $\text{accept}(n, v)$  to all accepters and wins if it receives a majority of votes.

So for each proposal, the algorithm proceeds as follows:

1. The proposer sends a message  $\text{prepare}(n)$  to all accepters. (Sending to only a majority of the accepters is enough, assuming they will all respond.)
2. Each accepter compares  $n$  to the highest-numbered proposal for which it has responded to a  $\text{prepare}$  message. If  $n$  is greater, it responds with  $\text{ack}(n, v, n_v)$  where  $v$  is the highest-numbered proposal it has accepted and  $n_v$  is the number of that proposal (or  $\perp$ , 0 if there is no such proposal). (An optimization at this point is to allow the accepter to send back  $\text{ack}(\text{higher number})$  to let the proposer know that it's doomed and should back off and try again—this keeps a confused proposer who thinks it's the future from locking up the protocol until 2037.)
3. The proposer waits (possibly forever) to receive  $\text{ack}$  from a majority of accepters. If any  $\text{ack}$  contained a value, it sets  $v$  to the most recent (in proposal number ordering) value that it received. It then sends  $\text{accept}(n, v)$  to all accepters (or just a majority). You should think of  $\text{accept}$  as a command ("Accept!") rather than acquiescence ("I accept")—the accepters still need to choose whether to accept or not.
4. Upon receiving  $\text{accept}(n, v)$ , an accepter accepts  $v$  unless it has already received  $\text{prepare}(n')$  for some  $n' > n$ . If a majority of accepters accept the value of a given proposal, that value becomes the decision value of the protocol.

Note that acceptance is a purely local phenomenon; additional messages are needed to detect which if any proposals have been accepted by a majority of accepters. Typically this involves a fourth round, where accepters send  $\text{accepted}(n, v)$  to all learners (often just the original proposer).

There is no requirement that only a single proposal is sent out (indeed, if proposers can fail we will need to send out more to jump-start the protocol). The protocol guarantees agreement and validity no matter how many proposers there are and no matter how often they start.

## 133. Informal analysis: how information flows between rounds

Call a round the collection of all messages labeled with some particular proposal  $n$ . The structure of the algorithm simulates a sequential execution in which higher-numbered rounds follow lower-numbered ones, even though there is no guarantee that this is actually the case in a real execution.

When an acceptor sends  $\text{ack}(n, v, n_v)$ , it is telling the round  $n$  proposer the last value preceding round  $n$  that it accepted. The rule that an acceptor only acknowledges a proposal higher than any proposal it has previously acknowledged prevents it from sending information "back in time"—the round  $n_v$  in an acknowledgment is always less than  $n$ . The rule that an acceptor doesn't accept any proposal earlier than a round it has acknowledged means that the value  $v$  in an  $\text{ack}(n, v, n_v)$  message never goes out of date—there is no possibility that an acceptor might retroactively accept some later value in round  $n'$  with  $n_v < n' < n$ . So the  $\text{ack}$  message values tell a consistent story about the history of the protocol, even if the rounds execute out of order.

The second trick is to use the overlapping-majorities mechanism that makes ABD work (see *SharedMemoryVsMessagePassing*). If the only way to decide on a value in round  $n$  is to get a majority of acceptors to accept it, and the only way to make progress in round  $n'$  is to get acknowledgments from a majority of acceptors, these two majorities overlap. So in particular the overlapping process reports the round  $n$  proposal value to the proposer in round  $n'$ , and we can show by induction on  $n'$  that this round  $n$  proposal value becomes the proposal value in all subsequent rounds that proceed past the acknowledgment stage. So even though it may not be possible to detect that a decision has been reached in round  $n$  (say, because some of the acceptors in the accepting majority die without telling anybody what they did), no later round will be able to choose a different value. This ultimately guarantees agreement.

## 134. Safety properties

We now present a more formal analysis of the Paxos protocol. We consider only the safety properties of the protocol, corresponding to validity and agreement; without additional assumptions, Paxos does not guarantee termination.

Call a value chosen if it is accepted by a majority of accepters. The safety properties of Paxos are:

- No value is chosen unless it is first proposed. (This gives validity.)
- No two distinct values are both chosen. (This gives agreement.)

The first property is immediate from examination of the algorithm.

For the second property, we need some invariants. The intuition is that if some value is chosen, then a majority of accepters have accepted it for some proposal number  $n$ . Any proposal sent in an  $\text{accept}$  message with a higher number  $n'$  must be sent by a proposer that has seen an overlapping majority respond to its  $\text{prepare}(n')$  message. If we consider the process that overlaps, this process must have accepted  $v$  before it received  $\text{prepare}(n')$ , since it can't accept afterwards, and unless it has accepted some other proposal since, it responds with  $\text{ack}(n', v, n)$ . If these are the only values that the proposer receives with number  $n$  or greater, it chooses  $v$  as its new value.

Worrying about what happens in rounds between  $n$  and  $n'$  is messy, so we'll use two formal invariants (taken more or less directly from Lamport's paper):

### Invariant 1

An acceptor accepts a proposal numbered  $n$  if and only if it has not responded to a prepare message with a number  $n' > n$ .

#### Invariant 2

For any  $v$  and  $n$ , if a proposal with value  $v$  and number  $n$  has been issued (by sending accept messages), then there is a majority of accepters  $S$  such that either (a) no accepter in  $S$  has accepted any proposal numbered less than  $n$ , or (b)  $v$  is the value of the highest-numbered proposal among all proposals numbered less than  $n$  accepted by at least one accepter in  $S$ .

The proof of the first invariant is immediate from the rule for issuing acks.

The proof of the second invariant follows from the first invariant and the proposer's rule for issuing proposals: it can only do so after receiving ack from a majority of accepters—call this set  $S$ —and the value it issues is either the proposal's initial value if all responses are  $\text{ack}(n, \perp, 0)$ , or the maximum value sent in by accepters in  $S$  if some responses are  $\text{ack}(n, v, n_v)$ . In the first case we have case (a) of the invariant: nobody accepted any proposals numbered less than  $n$  before responding, and they can't afterwards. In the second case we have case (b): the maximum response value is the maximum-numbered accepted value within  $S$  at the time of each response, and again no new values numbered  $< n$  will be accepted afterwards. Amazingly, none of this depends on the temporal ordering of different proposals or messages: the accepters enforce that their acks are good for all time by refusing to change their mind about earlier rounds later.

So now we suppose that some value  $v$  is eventually accepted by a majority  $T$  with number  $n$ . Then we can show by induction on proposal number that all proposals issued with higher numbers have the same value (even if they were issued earlier). For any proposal  $\text{accept}(v', n')$  with  $n' > n$ , there is a majority  $S$  (which thus overlaps with  $T$ ) for which either case (a) holds (a contradiction—once the overlapping acceptor finally accepts, it violates the requirement that no proposal less than  $n'$  has been accepted) or case (b) holds (in which case by the induction hypothesis  $v' =$  the value of some earlier proposal numbered  $\geq n = v$ ).

## 135. Learning the results

Somebody has to find out that a majority accepted a proposal in order to get a decision value out. The usual way to do this is to have a fourth round of messages where the accepters send  $\text{chose}(v, n)$  to some designated learner (usually just the original proposer), which can then notify everybody else if it doesn't fail first. If the designated learner does fail first, we can restart by issuing a new proposal (which will get replaced by the previous successful proposal because of the safety properties).

## 136. Liveness properties

We'd like the protocol to terminate eventually. Suppose there is a single proposer, and that it survives long enough to collect a majority of acks and to send out accepts to a majority of the accepters. If everybody else cooperates, we get termination in 3 message delays.

If there are multiple proposers, then they can step on each other. For example, it's enough to have two carefully-synchronized proposers alternate sending out prepare messages to prevent any accepter from ever accepting (since an accepter promises not to accept  $\text{accept}(n, v)$  once it has responded to  $\text{prepare}(n+1)$ ). The solution is to ensure that there is eventually some interval during which there is exactly one proposer who doesn't fail. One way to do this is to use exponential random backoff (as popularized by Ethernet): when a proposer decides it's not going to win a round (e.g. by receiving a nack or by waiting long enough to realize it won't be getting any more acks soon), it picks some increasingly large random delay before starting a new round; thus two or more will eventually start far enough apart in time that one will get done without interference.

A more abstract solution is to assume some sort of weak leader election mechanism, which tells each acceptor who the "legitimate" proposer is at each time. The accepters then discard messages from illegitimate proposers, which prevents conflict at the cost of possibly preventing progress. Progress is however obtained if the mechanism eventually reaches a state where a majority of the accepters bow to the same non-faulty proposer long enough for the proposal to go through.

Such a weak leader election method is an example of a more general class of mechanisms known as *FailureDetectors*, in which each process gets hints about what other processes are faulty that eventually converge to reality. (The particular failure detector in this case is known as the  $\Omega$  failure detector; there are other still weaker ones that we will talk about later that can also be used to solve consensus.)

---

CategoryDistributedComputingNotes CategoryOperatingSystemsNotes

## 137. ComputerSecurity

The problem of computer security can be divided roughly into two parts: a local part, consisting of **protection mechanisms** built into an operating system, and a more global part, consisting of higher-level defenses spanning multiple machines or entire networks against insider and outsider attacks. We'll start by talking about local protection mechanisms, and then continue with a discussion of practical attacks and defense against them.

## 138. Goals of computer security

Computer security has traditionally been seen (especially in a military context) as motivated by a broader goal of **information security**. There are a number of ways to define information security; a classic definition involves the so-called **CIA triad of confidentiality, integrity, and availability**. Confidentiality says that users who are not supposed to receive secret information don't get it. Integrity says that information should be preserved against tampering. Availability says that data should be accessible by those who are entitled to it when they need it.

It's easy to imagine these issues coming up in a military context: we want to keep the enemy from knowing where our forces are and what they are doing (confidentiality), while keeping track of what the enemy is doing (integrity/availability). But the general approach of the CIA model also applies in civilian contexts where we care about the usefulness of information: credit histories are generally subject to confidentiality or privacy protections (so, for example, we can't base your 422/522 grade on your crummy credit history), and are only useful if they can't easily be modified (so you can't escape bankruptcy by paying the neighborhood script kiddie \$100 to fix your record) and are available when needed (by say the loan officer at the local bank). Similar considerations apply to other private data, such as medical histories or video rental records (in the United States), or just about any record with your name on it (in Europe).



Other classes of attacks don't exactly fit in the CIA model (but can be made to fit with enough pushing). Some examples from SGG §15.1 include theft of service (where I consume your valuable resources through impersonation or some other breach of your security) and denial of service (where I temporarily prevent you from using some resource; SGG distinguishes this from breach of availability by using the latter only for permanent destruction).

## 139. Protection

Basic idea: A computer system contains many resources, some of which we want to protect from malicious or foolish users. A protection mechanism restricts what some processes can do. Examples include **filesystem protections** used to keep users from accidentally trashing system files (or, on timesharing systems, from reading or damaging each other's files), **memory protections** used to keep insane user processes from damaging system resources, and **digital rights management** used to make would-be consumers of valuable copyrighted material pay what they owe.

### 139.1. Principle of least privilege

A common feature of most protection systems is the **principle of least privilege**, the computer security equivalent of the **need to know principle** in traditional military security. The idea is that no users or programs should be given any more power than they need to get their tasks done. So a web server, for example, should have access to the network port it listens to and the files it is serving, but should probably not have access to user home directories or system configuration files. This is true even if the server solemnly promises not to abuse this access; if the system enforces that the web server can read my collection of embarrassing love letters, there is no possibility that a bug or misconfiguration of the server might expose them to Google's all-powerful search engine. Similarly, it is generally best to protect the installed software base of a system from ordinary users, who might damage or corrupt the software by accident on a single-user system (possibly by downloading malicious trojan horses or viruses), or who might deliberately corrupt software on a shared system to obtain access to other users' data. A reasonable security system will allow users to be locked out of such access under normal conditions.

There is a conflict between the principle of least privilege and the goals of simplicity and accessibility. Taken to an extreme, the principle of least privilege would demand that every access to every file be constrained by the specific needs of the task at hand: my word processor, for example, might be allowed to read its own word processing files but would be prevented from examining my web browser cache. Conversely, my web browser might not be allowed to look at my word processing documents. Such a system would prevent several possible attacks on my data (e.g. a corrupted web browser secretly uploading all my valuable trade secrets), but in practice nobody uses it because the cost of keeping track of which programs can legitimately access which data exceeds the security benefits for all but the most paranoid users, and additional complications arise with unanticipated uses (like wanting to upload my documents to a coursework submission website). So in practice the principle of least privilege is usually implemented approximately.

### 139.2. Users, roles, and groups

The mechanism for such approximate privileges is the notion of a **user** or sometimes more specifically a **role**. Each user represents an agent with interests distinct from that of other users; these may correspond to real human beings, or may consist of virtual users representing some subsystem with special privileges (or especially restricted privileges) like a web server or the filesystem. A typical user-based protection system will by default provide the same privileges to each program running under control of a particular user regardless of what task the program is executing; the assumption is that users can be trusted to look out for their own interests, and that a user's right hand does not need protection from their left.

For some purposes it makes sense to subdivide a user further into distinct roles with different privilege levels. So, for example, the owner of a particular machine might normally work as a normal user with no special privileges (thus providing protection against accidents or malicious code), but will occasionally adopt a more powerful role to install new software or perform system maintenance. From the point of view of the underlying system, there is not much difference between a single user with multiple roles and multiple users, but a particular implementation may condition adopting particular roles on already identifying as a particular user (e.g. **sudo** in Unix-like systems or administrator privileges in Windows).

For convenience, it may also be useful to categorize users into **groups**, where all members of a given group are given the same privileges with respect to particular objects. So, for example, users on a Windows machine might be members Administrator group, allowing them to invoke Administrator privileges when needed. A Linux machine might put users in the **audio** group to give them access to the speaker and microphone, or the **cdrom** group to give them access to the CD-ROM device. A **daemon** group might include system daemons that need powers common to all system daemons (like the ability to append to log files).

### 139.3. Protection domains

A **protection domain** defines operations a process may perform at a given time; the power to execute a particular operation on a particular object is called an **access right** and a protection domain is essentially a bundle of access rights. In a user-based system the domain of a process will generally be determined by the identity of the user that runs the process; however, more complex systems might allow per-process protection domains or even per-procedure protection domains.

Protection domains may be **static** or **dynamic**. In the former case, the access rights of a process are fixed; in the latter, there may be some mechanism for granting a process new access rights or removing rights it already has. Dynamic protection domains are often necessary if we are fanatical about least privilege: the access rights that a process needs at one point in its execution may be different from those it needs later. But because of the complexity of fine-grained dynamic control of access rights, a more practical solution might be to combine static or near-static domains with **domain switching**, where a process can obtain a new bundle of rights when needed. Examples of domain switching are the switch to kernel mode in a system call (or the more general ring gateway mechanism in Multics), the setuid bit in Unix that allows a program to specify that it should be run with the privileges of the program owner in addition to those of the user that calls it, and mechanisms for switching roles as described above.

### 139.4. Access matrices

An access matrix specifies the relation between protection domains and access rights. In full generality we could have a system that allows each entry in an access matrix to be specified separately; in practice, most systems establish simpler protection policies from which the access matrix is implicitly derived.

By convention, each row of the access matrix corresponds to a protection domain, and each column to an object. Each entry gives a list of access rights to that particular object. So we might have a matrix that looks like this:

Tetris high score list		A's diary	Printer
A	append	read, write, delete	print
B	append	read	print, disable

Here B can read A's diary but not write to it or delete it. Both A and B can append to the Tetris high score list, and both can print, but only B is allowed to turn the printer off.

#### 139.4.1. Who controls the access matrix?

Typically each column of the access matrix corresponds to an object that is **owned** or created by a particular user. It is natural to give this user control over the entries in the column. The right to modify entries in the access matrix may also be controlled by the access matrix itself, through an explicit owner right.

#### 139.4.2. Copy rights

An access right may provide the right to copy or transfer the right to new domains. So, for example, I might have *read\** access to an object, which means that not only may I read the object but I may grant read access to other domains. There are several variants of this: a transfer right allows me to copy my access right to a different domain provided I give it up myself; a limited copy right allows me to copy the underlying right to a new domain without also granting the right to copy it further.

#### 139.4.3. Confinement

One problem with copy or owner rights is that they can allow privileges to escape to domains that shouldn't have them. We may want, for example, to guarantee that none of the TOP SECRET data on our system is ever visible to a user with only SECRET clearance, but if we allow an owner to adjust access rights arbitrarily the owner can easily grant rights to the data to anybody they like. In this particular case we can impose a ring structure on data where high-confidentiality data is never expose to low-confidentiality domains, but in more general cases it may be very difficult to enforce exactly the restrictions we want. We also quickly run into computational limits: once a protection system allows users to grant each other rights under reasonably sophisticated conditions, it may become NP-hard to detect if a particular user can obtain a particular right through normal execution of the system. So the problem of confinement is generally considered be unsolvable at the software level, and for applications where it is critical is usually enforced externally (e.g. via locked rooms).

### 139.5. Implementation

There are basically two standard mechanisms for specifying access rights: **access control lists** and **capabilities**. The difference is that an access control list is associated with an object, while capabilities are associated with a domain.

#### 139.5.1. Access control lists

In the most general form, an access control list or ACL describes exactly the column in the access matrix associated with a particular object. So a file might state that it can be read by any administrator and appended to by any system daemon (using in this case groups as protection domains). Similarly, a subdirectory in a version-control system might be readable by John Q. Tester, readable and writable by Jane C. Developer, and allow updates to its permissions by Hieronymous J. Administrator. These values would most likely be stored in a data structure along with the protected object, e.g. in the inode in a filesystem.

Some systems provide weaker versions of access control lists. For example, standard Unix file permissions are divided into read, write, and execute access rights (the last has a special meaning for a directory, allowing a user to find a file by name in a directory but not list all files). For each file, these can be set separately for (a) the owner of the file, (b) a single group that the file belongs to, and (c) arbitrary users. So it is possible to create a file that anybody can read but only the owner can write to, but it is difficult to create a file that A and B can both read but only C can write to (it's necessary to create a new group containing only A and B), and it's impossible to create a file that only A and B can read but only B and C can write to (since you would need two groups, and each file can only be assigned to one group). So Unix necessarily violates the principle of least privilege in some plausible scenarios. In practice, these issues are usually worked around by building setuid daemon processes that enforce arbitrarily convoluted access control, but this can be expensive.

#### 139.5.2. Capabilities

An alternative approach is to store rows of the access matrix instead of columns. Here each protection domain carries with it a list of **capabilities**, tokens that indicate particular access rights for particular objects. These capabilities may be transferable to other protection domains with or without the supervision of the operating system, depending on the implementation.

An example of a capability is a Unix file descriptor. When a file is opened, the kernel checks the current user against its access permission. But once the kernel returns a file descriptor, further access to the file requires no additional permission checks, and indeed changes to the permission bits at this point will not affect use of the file descriptor. File descriptors are mostly not transferable (unless you are running Mach); the only case where they are transferred is as the result of a **fork** or **exec** system call.

A more sophisticated capability mechanism allows transfer of capabilities between processes. This allows for very fine-grained control of access rights. For example, if I want to run a word-count program on one of my valuable files, instead of running the program with all of my privileges (possibly allowing it to read or write any of my other files), I could give it only the capability representing read access to the particular file I want it to look at.

The usual difficulty with capabilities is that they are harder than ACLs to keep track of. Given a particular resource, it is easy to figure out exactly which users can access it. This is especially true if capabilities are transferable or if they are represented externally (say by using cryptographically signed certificates).

## 140. Implementation



Protection domains and access matrices only describe what we want to protect. The question remains of how we actually do it.

## 140.1. Authentication

The first problem that arises is: how do we know which user we are dealing with? This is the problem of **authentication**. Some options:

### 140.1.1. Something you know

Like a **password**. Here the idea is that if only I know what my password is, the system can trust anybody who types in my password to be me. There are some well-known problems with this:

- Simple passwords may be easy to guess. Early work on cracking passwords showed that the vast majority of passwords could be easily found by trying a few thousand possibilities, with the most common password being **password**. Later systems have tried to avoid this problem by demanding, for example, 8 or more character passwords with a mix of letters and numbers; according to legend, this has led to a new most common password **password1**.
- Passwords stored or transmitted in cleartext can be stolen. The usual way of dealing with this is with a **cryptographic hash**, a function  $H$  that disguises its argument in a consistent (and hopefully irreversible) way. So now instead of storing **password1** in my server's authentication database, I store  $H(\text{password1})$ . Further security can be added by storing  $H(\text{some-server-secret, password1})$ ; this makes it harder to keep a single list of hashed passwords to compare against multiple servers' databases. Of course, preventing an attacker from seeing the hashed passwords is also a good idea.
- Passwords transmitted across a network are vulnerable to replay or man-in-the-middle attacks. See below for more details.

The moral: Passwords provide only minimal security under typical circumstances. And yet (a) we still use them for everything, and (b) the most valuable passwords (like 4-digit banking PINs or non-resettable Social Security Numbers) are often the weakest! Why is this? (My guess: There is much psychological comfort in letting users think they are in control of their own security.)

### 140.1.2. Something you have

We can strengthen passwords somewhat by replacing them with cryptographic authentication mechanisms using e.g. public-key cryptography. Many public-key cryptosystems can be adapted to give **digital signatures**, where I can sign a document using my **private key** and anyone who knows my **public key** can verify the signature. So now instead of a password dialog that looks like this:

Computer: please send password

Me: **password1**

we have something that looks more like this:

Computer: please send signed authentication for service X at time 2007-04-25T13:22:07

Me: <digitally signed document that says "I want to access service X at time 2007-04-25T13:22:07">

The difference is that the digitally signed document is presumably unforgeable by anybody who doesn't have access to my private key, and can't be reused at a later time to cause trouble. Note that this type of system may still be vulnerable to man-in-the-middle attacks.

Secret keys have to be stored somewhere; for typical public-key systems the length of a secure key is long enough (starting at around 1024 bits) that it is impractical to have a user remember one. So this means storing the keys on the user's personal computer or better yet on some device that the user carries around with them. Various smartcards, dongles, and bits of jewelry have been marketed for this purpose, and it would be trivial to implement a system that keeps cryptographic keys on a USB device. Nobody I know carries one. So to the extent that we currently use cryptographic authentication (at least in civilian contexts), we mostly are relying on the security of our personal computing devices.

There is a long history of authentication tokens other than cryptographic keys: physical keys, ATM cards, signet rings and other seals (as found in Europe) and chops (as found in East Asia) all work on the something-you-have principle.

### 140.1.3. Something you are

If a computer could identify a user directly, we wouldn't need any other authentication. Possibilities here generally involve some form of **biometrics**, such as fingerprint or retinal scanning, face recognition, or more exotic measurements like measuring typing characteristics, voiceprints, or walking gait. These all share several serious problems:

- Biometrics is hard to do well. Face recognition (the worst of the lot) runs to 1% false positive rates even under good conditions. Other methods have lower failure rates but can still be inaccurate. All such methods (except perhaps typing analysis) require special hardware.
- Biometrics can be spoofed. Fingerprints and voiceprints are both vulnerable to replay attacks, where an image or recording substitutes for the real thing. Retinal scanners claim to be less vulnerable to this (they can look for dynamic properties like saccades or blood circulation), but spoofing is still likely to be possible with more sophisticated techniques.
- Biometrics can't be reset when compromised. If you lose your password, you can change your password. If somebody copies your thumbprint, you can't do much about it.
- The difficulty of stealing biometric authenticators may actually work against the owner. If some deranged mugger with a machete demands access to my bank account, I'm going to be happier if he can get it with my ATM card than if he needs my eye and my thumb.

### 140.1.4. Two-factor authentication

Because individual authentication mechanisms are weak, a common approach is to require **two-factor authentication**, where the user must demonstrate their identity in two different ways (that hopefully have different vulnerabilities). So an ATM machine demands both an ATM card (that is hard to copy) and a PIN (that is hard to steal); this protects me against pickpockets and against muggers who aren't willing to actually drag me to the nearest ATM to make sure the PIN I gave them isn't bogus. Because of the strength of two-factor authentication, current US banking regulations demand that on-line banking be done using it. Unfortunately, many banks have successfully convinced regulators to interpret requiring a username (factor 1) and a password (factor 2) entered on separate web pages as qualifying.

## 140.2. Authorization

Here the issue is the design and implementation of access policies. The main problem as discussed previously is the trade-off between convenience and least privilege. For users, this usually means separating roles, so that the identity I use to buy books from Amazon is not the same identity I use to tell Yale where to send my paycheck. However, there are still vulnerabilities here, since somebody (or some program) I delegate my authority to may misuse it: a compromise web browser or web proxy, for example, may use my Amazon password to send large amounts easily-fenced jewelry to the summer home in Moldova I didn't know I had.

## 140.3. Enforcement

Even if we know who a user is and what access rights they have, we still have the problem of enforcement. It doesn't matter if my filesystem correctly determines that user `evil` isn't allowed to read file `sensitive.txt` if it then goes ahead and lets them read it anyway. This can be particularly difficult dealing with an operating system containing millions of lines of code of variable quality, some obtained from questionable sources, and all running with full kernel privileges; any flaw can potentially be exploited to obtain access to protected objects.

To the extent that there is a solution here, it is made up in practice of several pieces:

### Minimize the size of the trusted computing base

If we can reduce the amount of code with full access to the hardware, we reduce the problem of eliminating bugs. This is often used as an argument for microkernel designs (where the kernel can be as small as a few thousand lines of code) or exokernel designs (where hardware is partitioned between processes directly and protection is limited to restricting access to particular memory regions or disk cylinders); however, this may just push the issue of enforcement off into user-level processes that implement their own security policies. An alternative is to implement security directly in hardware: for example, digital rights management for HDTV is implemented in part by only doing the final decryption of the video image inside the display device itself, reducing reliance on the trustworthiness of path leading to it.

### Partition the system to contain faults

That is, apply the principle of least privilege by separating components into distinct protection domains as much as possible. So for example a web server can be run inside a restricted subsystem that hides most of the normal filesystem (a chroot jail in Unix terms) or even inside a virtual machine with no access to the underlying hardware. Similar techniques can be applied to device drivers, filesystems, etc.: if there is no reason for my wireless driver to touch the superblock on my filesystem, then I probably shouldn't let it.

### Rely on trusted code

If I'm worried about buggy code breaking my machine, then I should only run thoroughly-debugged code that is certified as such by somebody I trust. To a first approximation, this is the security mechanism most users rely on most, and ultimately any user of somebody else's code (including assemblers or compilers—see the famous compiler hack by Thompson) must do this. Violating this trust is also the most effective practical way to subvert security, since it doesn't rely on finding bugs.

## 140.4. Intrusion detection and recovery

Try as we might to secure a system, there is still a possibility of compromise. We'd like to detect such a compromise and recover from it as quickly as possible. Detecting breaches comes under the heading of **intrusion detection**, which typically consists of extensive **logging and accounting** (so we can see what our system has been doing) and **auditing** (so we can check what is going on against what we think should be going on). Logging ideally occurs in a form that can't be disabled and can't be erased by attackers,<sup>3</sup> although even a protected file in a filesystem that might be compromised my help against stupid attackers. Auditing typically involves studying log files, the filesystem, and/or kernel memory to look for signs of a breach. This can be done by looking for positive signs of compromise (e.g. virus scanners looking for virus signatures) or by looking for unexpected changes (e.g. virus scanners looking for files that have changed but shouldn't have). The ultimate difficulty with auditing is the problem of weeding out false positives: often, human intervention is required to detect if some file really should have changed (or if I really did mean to send lots of jewelry to my summer home in Moldova).

Recovery from intrusion is often difficult. Because it is hard to tell exactly what has been compromised, and because the tools we use to observe the system may themselves have been compromised, often the only safe course is to reinstall from a more secure backup.

## 141. Practical attacks and defenses

### 141.1. Attacks on individual machines

#### 141.1.1. Buffer overflow and other code injection exploits

An attackers' goal on a single machine is **privilege escalation**, where I execute code in a limited protection domain that somehow obtains access to a more powerful domain. One common method for doing this is **code injection**, where I can convince some privileged process to execute code of my choosing. Some variants:

#### Buffer overflow

A program reads some parameter from the user that it stores in a 512-byte buffer allocated on the stack. The attacker supplies a 1037-byte value for the parameter that overwrites some other part of the stack, seizing control of the process by changing some other variable or possibly even code. Solution: Don't write anything critical in a programming language that doesn't check array bounds (this generally means C and C++). Problem: Everything is already written in C or C++. A less effective but more feasible solution is to carefully insert array bounds checks everywhere in your existing C or C++ code.

#### Code injection

I cleverly replace all my buffer-overflow-prone C/C++ code with code written in some exciting scripting language like Perl, Python, TCL, Ruby, Visual Basic, or (since I'm using a database anyway) SQL. Part of my program generates scripting commands on the fly of the form  
`do something; do something else; open database record for <username>; do something with it; etc,`  
where commands are separated by semicolons and `username` is supplied by the user. Everything goes great until some smart-aleck named `HaHa`; `delete all database records` signs up for my free web service. Solution: Be very careful about quoting and using user-supplied input.

### **Symlink attacks**

Long ago (1970s and 1980s), it was possible to break into a BSD machine by creating a symbolic link `/tmp/foo` to some `setuid` shell script. Since shell scripts work by having (a) the shell program start (under the uid that the script is `setuid` to) and then (b) open and read the shell script file, a clever attacker could redirect the symbolic link between steps (a) and (b). Most Unixes no longer allow `setuid` shell scripts for this reason.

### **141.1.2. Viruses**

A **virus** copies itself into executable programs on disks so that it obtains new privileges when these programs are run by a different user. Before widespread networking, these were the main automated security threat. Solution: Carefully limited access rights, integrity scanning for executables (including not executing unsigned executables, since a virus hopefully can't forge signatures), user education.

### **141.1.3. Trojan horses**

A **trojan horse** is a program that claims to do something good but really does something evil. These are often automated versions of **social engineering attacks**: for example, a user may receive an executable file that claims to be a self-extractive archive of a notice of impending court proceedings or something equally threatening. Solution: Don't let users run programs they find on the street unless they can really convincingly argue that the program is trustworthy.

### **141.1.4. Worms**

Automated program that uses local privilege escalation exploits, network buffer overrun exploits, and trojan horse trickery to propagate itself from one machine to another. First appeared in fiction (*The Shockwave Rider* by John Brunner). Now all too real. Solution: Close the holes the worms rely on.

## **141.2. Network attacks**

### **141.2.1. Replay attacks**

I record your password going over the wire and reuse it. Solution: encrypt everything.

### **141.2.2. Man-in-the-middle attacks**

I pretend to be your bank to you and pretend to be you to your bank. Anything your bank asks for to authenticate you, I from you, and vice versa. Once we are authenticated to the bank, I hijack your connection and drain your accounts.

To execute a man-in-the-middle attack, I have to be able to insert myself in the middle. Some ways to do this are compromising DNS (so that `bank.com` goes to my IP address), tricking you via phishing email into following a bogus URL (to say, `bank.com`), compromising your network connection by compromising some intermediate router (it may help if your Netgear wireless access point still has the default administrative password `netgear`), or compromising your computer or web browser so I can intercept your credentials directly (don't download Firefox extensions from `bank.com`).

Man-in-the-middle attacks are particularly difficult to deal with because no matter how clever your authentication protocol, the attacker can simulate it from both ends. The only solution is to have some shared secret that allows the remote end to authenticate itself; hence the security certificates in web browsers (which don't actually help much against typical users, who have been trained to happily click accept this certificate buttons).

## **142. How much of a problem is security?**

For normal users, the history of computer security looks like this:

1. A popular operating system or program ships with an easily exploited, catastrophic vulnerability.
2. Nefarious bad guys exploit the vulnerability to do something highly costly and visible.
3. Either vulnerability is quickly patched, or damage is contained through some other means.
4. Process repeats.

The effect of this is that the most dangerous vulnerabilities tend to be the ones that are fixed first, or at all. There is a sense in which this is economically optimal. It also fits well with the pattern for physical security, where a plausible rule of thumb is that the number of security devices on the front door of a house is roughly equal to one or two plus the number of times the house has been broken into.

But: The efficiency of computers also means they are efficient at propagating worms, viruses, etc. It is theoretically possible to infect every vulnerable Windows machine on the open Internet using a new remote exploit in less than a minute. If the worm that does this is sufficiently malicious (e.g. erasing all infected hard drives and then doing denial of service on all nearby routers), the damage to the world economy could be quite high. Perversely, our best hope against such nightmares may be the trickle of annoying but not very dangerous worms, which highlight existing vulnerabilities and force them to be fixed.

---

CategoryOperatingSystemsNotes

## **143. Virtualization**

## **144. A brief history of operating systems**

- Single-application computers: one CPU, one program.
- Timesharing: one CPU, one program at a time.
- Virtual machines: one CPU, multiple programs (that all think they are running alone).

- *Processes*: abstract CPU, multiple programs (that all think they are running at the same time).
- *Threads*: two layers of processes, with lightweight processes running inside heavyweight processes.
- *Virtualization*: virtual machines running complete operating systems inside processes.

The last step can be seen as a return to the virtual machine days of yesteryear—possibly even as a precursor to going back to one CPU per program. Or it can be seen as an attempt to build fully recursive processes.

## 145. Why virtualize?

- Simulate hardware you don't own (e.g. SoftPC/SoftWindows/Virtual PC for Macs in the 1980's and 1990's).
- Simulate hardware you don't own any more.
- Share resources with full isolation (e.g. rented web servers).
- Run programs that expect incompatible OS environments (e.g. using VMWare to run Office under Linux).
- Run programs that you don't trust with access to the underlying hardware.

## 146. Virtualization techniques

The goal is to run a **guest operating system** on top of a **host operation system** so that the guest OS thinks it is running on bare hardware. There are basically two ways to do this: using an **emulator** or a **hypervisor**.

### 146.1. Emulation

This is the simplest conceptually. We write a program (in C, say) that simulates all of the underlying physical hardware, including the CPU. CPU registers, the MMU, virtual memory, etc. are all represented using data structures in the program, and instruction execution involves a dispatch loop that calls appropriate procedures to update these data structures for each instruction.

Examples: bochs, SoftPC, many emulators for defunct hardware like Apple II's or old videogames.

Advantages: Runs anywhere, requires no support from host OS.

Disadvantage: Horrendously slow. When emulating old hardware, this is not necessarily a problem: A 2 GHz Pentium doing up to 4 instructions per clock cycle can do a pretty good job of faking a 1 MHz 6502 doing 2-3 clock cycles per instruction. But it's less convincing when emulating recent hardware.

### 146.2. Hypervisors

A **hypervisor** or **virtual machine monitor** runs the guest OS directly on the CPU. (This only works if the guest OS uses the same instruction set as the host OS.) Since the guest OS is running in user mode, privileged instructions must be intercepted or replaced. This further imposes restrictions on the instruction set for the CPU, as observed in a now-famous paper by Popek and Goldberg published in CACM in 1974, "Formal requirements for virtualizable third generation architectures" (see <http://portal.acm.org/citation.cfm?doid=361011.361073>).

Popek and Goldberg identify three goals for a virtual machine architecture:

1. *Equivalence*: The VM should be indistinguishable from the underlying hardware.
2. *Resource control*: The VM should be in complete control of any virtualized resources.
3. *Efficiency*: Most VM instructions should be executed directly on the underlying CPU without involving the hypervisor.

They then describe (and give a formal proof of) the requirements for the CPU's instruction set to allow these properties. The main idea here is to classify instructions into **privileged** instructions, which cause a trap if executed in user mode, and **sensitive** instructions, which change the underlying resources (e.g. doing I/O or changing the page tables) or observe information that indicates the current privilege level (thus exposing the fact that the guest OS is not running on the bare hardware). The former class of sensitive instructions are called **control sensitive** and the latter **behavior sensitive** in the paper, but the distinction is not particularly important.

What Popek and Goldberg show is that we can only run a virtual machine with all three desired properties if the sensitive instructions are a subset of the privileged instructions. If this is the case, then we can run most instructions directly, and any sensitive instructions trap to the hypervisor which can then emulate them (hopefully without much slowdown).

The bad news: Most CPU architectures contain sensitive but unprivileged instructions, known as **critical instructions**. For example, IA32 architecture allows unprivileged programs to read the Global and Local Descriptor Tables, so if the hypervisor is lying about the interrupt vectors the guest OS can find this out. (A more complete list of bad instructions on IA32 can be found at [http://www.floobydust.com/virtualization/lawton\\_1999.txt](http://www.floobydust.com/virtualization/lawton_1999.txt).) So some mechanism is needed to trap these instructions.

#### 146.2.1. Using breakpoints

One approach is to use the CPU's breakpoint mechanism to trap on critical instructions. This requires scanning code to be executed so we know where to put the breakpoints. The tricky part is that typically we don't have enough breakpoints to cover all critical instructions, so in practice we can only execute natively code in chunks, where we trap anything that escapes from the chunk we have covered (this is not as hard as it sounds, since we can use the virtual memory system to mark any page outside the current one as non-executable). This requires that when we switch to a new chunk we rescan it, adding quite a bit of overhead to executing straight-line code.

#### 146.2.2. Using code rewriting

A more efficient method is to rewrite the code itself. If we replace every occurrence of a critical instruction with a system call, we can emulate the critical instruction directly without any sneakiness. We can do the same for all privileged instructions as well, which may slightly increase performance just using protection faults. The problem is that now the guest OS may notice that its code isn't what it thought it should be.

Fortunately, the virtual memory system again comes to our rescue: by marking each rewritten page as executable but not readable, any attempt by the guest OS to read a page can be trapped. We can then supply the data from the original page (which we presumably kept around somewhere).

### 146.2.3. Using paravirtualization

A third approach is to let the guest OS do its own code rewriting. Here we use a modified guest OS that replaces privileged instructions with explicit hypervisor calls. We still need to detect and trap any sensitive instructions, but the cost of doing so is likely to be small (if we are lazy, we can simply ignore the issue of critical instructions, since we have already given the game away by asking for a modified guest OS). This is the best approach if we can do it, but since it depends on modifying the guest OS, it doesn't work in general.

### 146.2.4. Using additional CPU support

The ultimate solution is to fix the CPU so that there are no critical instructions. We can't change the instruction set if we want to run old programs unmodified, so instead we have to expand the CPU to move control registers into virtual machines implemented in hardware (again, back to the past). This is done in recent Intel and AMD CPUs; a description of the Intel approach can be found [\[here\]](http://www.intel.com/technology/itj/2006/v10i3/1-hardware/5-architecture.htm). Such support allows for **ring aliasing**, where unprivileged code thinks it is running with higher privileges, and allows for executing many privileged instructions that control CPU state without faulting (because they now execute the fake state in the virtual machines, which can be modified without causing trouble). Quite a bit of work is still needed to translate operations on the fake machine to operations on the underlying machine; for operations that actually affect the system (I/O, changes to virtual memory), the CPU must trap to the hypervisor running on real hardware.

## 147. Applications

We've already mentioned some of the main applications. Broadly speaking, there are three main reasons to use a virtual machine:

1. Emulating hardware or operating systems that would otherwise not be available.
2. Timesharing with full OS isolation.
3. Security.

Timesharing mostly comes up with systems that expect to have full control of the machine. For example, web servers and database servers typically expect to be the only one running at a time. So if you want to rent out webserver space, it makes sense to split your single real server among multiple virtual machines that can be configured to the tastes of your various clients. This also provides isolation, always a good thing.

Isolation can also be an issue for programs that you don't trust. If you worry that your webserver can be compromised, running it inside a virtual machine prevents it from escaping and compromising the rest of your system; a VM thus acts as a perfect jail. Conversely, bad guys can use virtualization to produce near-perfect rootkits: having a compromised machine appear indistinguishable from an uncompromised machine is the definition of successful virtualization. Such techniques may also be used to subvert software-only DRM mechanisms.

---

CategoryOperatingSystemsNotes

## 148. UsingBochs

**Bochs** is a 386 PC emulator that runs on top of a variety of operating systems. Full documentation can be found through the [Sourceforge project page](#). These notes are to get you started with using bochs on the Zoo machines.

## 149. Basic use

Type **bochs** in a terminal emulator window. This will pop up a text menu with several options. Unless you have a **bochsrc** or **.bochsrc** file in the current directory, you will probably need to specify at minimum a disk image file using **Edit Options/Disk Options**. You can then run the emulator using **Begin simulation** from the main menu.

Since having to edit options all the time is annoying, you can save your current options out to a **bochsrc** file once you have the setup you like. **Bochs** will load options from a file with this name by default, or you can tell it to load from a different file with the **-f** option, e.g. **bochs -f bochsrc-unusual**. For CS422 assignments, we will generally supply you with a standard **bochsrc** file along with the assignment files.

## 150. Debugging

**Bochs** runs in debugging mode if compiled with the appropriate files. Luckily, if you are taking CS422 we have already compiled it for you. Look for **common/bin/bochs-debug** in your user or group directory; this should run on any recent Intel-architecture Linux machine (e.g., Zoo nodes).

The **bochs-debug** program works exactly like stock **bochs**, except that it gives a gdb-style command prompt after initializing. Type **help** to get a list of commands or see <http://bochs.sourceforge.net/doc/docbook/user/internal-debugger.html> for documentation. If you just want to run your simulation, type **c** (for continue).

### 150.1. Breakpoints and breakpoint gotchas

There are several breakpoint commands in **bochs-debug**. The most useful early on is probably the stock breakpoint command **b**, which lets you specify a physical memory address (e.g. **b 0x7c00** will set a breakpoint at the start of your bootloader). One annoying feature of **bochs** as currently implemented appears to be that the **step** command will not step over a breakpoint! So you may need to delete your breakpoint (e.g. **d 1** for the first one you set) before you can start stepping through your program.


## 151. Alternatives

*Qemu is generally faster than bochs and may be more forgiving. Typical usage is `qemu -fda ./image`. Unfortunately, qemu doesn't come with a debugger, and we won't be testing your software with qemu by default.*

---

CategoryOperatingSystemsNotes

## 152. UsingSubversion

*These are very sketchy notes on using Subversion for CS422 students. For more details see the Subversion web site at <http://subversion.tigris.org> or the online book  Version Control with Subversion.*

## 153. Basic concepts

*Subversion is a centralized [WikiPedia: Distributed\\_version\\_control\\_system](#), which in this context is a fancy way to say a distributed file system with versioning. Files are stored in a repository, which is either a local directory accessible to the user or on a server somewhere. To use the files, you need to check out a local working copy that you can tinker with. When you are done, you commit your changes back to the repository, supplying a log message so that other users who may be sharing the same files will know what you did.*

## 154. Subversion commands: basics

*All Subversion commands go through the `svn` executable. Type `svn help` for a list.*

*Typical use of Subversion involves only three commands: `svn co` to get the initial working copy, `svn up` to pull changes from the repository, and `svn commit` to put your changes back.*

### 154.1. `svn co [url]`

*Check out a working copy from the given URL. This will create a copy of the most recent version in the repository under the current working directory.*

*Example: `svn co http://pine.cs.yale.edu/422/user/some-user-name` creates a new directory `some-user-name` in the current directory. If the repository is password-protected, you may need to supply a username and password for this to work. You can also access some repositories directly through a web browser (but you will generally not be able to commit changes or get at past history).*

### 154.2. `svn up`

*Bring the current working copy up-to-date, incorporating (by merging) any changes that were made in the repository. Any local changes will not be lost; instead, Subversion will attempt to merge non-overlapping changes together. If it fails, you will have to clean up after it (see `svn resolved` below); this should only happen if somebody else commits a change since the last time you did.*

### 154.3. `svn commit`

*After changing something, you can put your changes back with `svn commit`. This will pop up your default editor to write a log message. If you want to avoid dealing with the editor, you can supply a message on the command line with the `-m` switch, e.g. `svn commit -m "fixed life-threatening scratch monkey remount bug"`.*

## 155. Getting information

### 155.1. `svn log`

*Prints all log entries touching the current directory to `stdout`.*

### 155.2. `svn status`

*Tells you what files in your working copy are modified, what files are unknown, etc., what files have unresolved conflicts, etc.*

### 155.3. `svn diff`

*Tells you the difference between your working copy and the checked-in version. Can also be used with the `-r` or `-D` switches to get differences between the working copy and older revisions, e.g. `svn diff -r1031` or `svn diff -D yesterday`. Can also be used on individual files: `svn diff broken.c`.*

### 155.4. `svn cat`

*Send a copy of a file to `stdout`. Mostly useful with `-r` or `-D`: `svn cat -D 2006-05-01 may-day-parade-schedule.txt` is the best way to recover an old version of some file. (Don't be tempted to use `svn up` for this, despite what the documentation says: you will get your working copy stuck in some ancient state.)*

## 156. File operations

Subversion doesn't pay attention to files unless you tell it to, and for this reason any changes you make that go beyond editing files Subversion already knows about require you to tell Subversion about them.

**Note:** You should **not** add any files to the repository that can be regenerated automatically (e.g. binaries). Not only will this take up a lot of space in the repository, but it will also lead to confusion later when **make** refuses to rebuild some file because **svn** up already gave you a "new" copy.

### 156.1. `svn add [file]`

Tell Subversion to track some new file it wasn't tracking before. Also works on directories (it adds all contents recursively).

### 156.2. `svn mkdir [directory]`

Like `mkdir dir; svn add dir`.

### 156.3. `svn rm [file]`

Tell Subversion to delete a file. For reasons that are not at all obvious, Subversion insists that you delete the file yourself before calling `svn rm`; the fix to this is to use the `-f` switch, e.g. `svn rm -f annoying-extra-file`. Also works on directories.

### 156.4. `svn cp [source] [destination]`

Like stock `cp`, only does an `svn add` on destination for you. Also copies log information, so that the new file inherits the version history of the old. This is cheaper for the repository than copying the file yourself, because it stores only the changes (if any) between the two copies rather than both copies.

### 156.5. `svn mv [source] [destination]`

Pretty much equivalent to `svn cp source destination; svn rm -f source`.

Note that if you want to move a file on top of another one, you have to delete the target (using `svn rm`) and commit the change (using `svn commit`) first.

## 157. Fixing things

### 157.1. `svn revert [file]`

If you screw up editing something, you can throw away your changes with `svn revert`, e.g. `svn revert broken-file`. Be careful with this.

### 157.2. `svn resolved [file]`

If a merge fails, you will get a file that looks something like this:

```
This is my file.

<<<<<<< .mine
This is a change in my working copy.
=====
This is an incompatible change somebody else made.
>>>>>>> .r6258

This is the rest of my file.
```

There will also be several extra copies of the broken file corresponding to the various versions Subversion tried to merge together. Fire up your editor, fix the file, and tell Subversion you did so using `svn resolved filename`. This will clean up all the extra files and let you run `svn commit` again. (Hopefully you will not need to use this command much.)

---

CategoryOperatingSystemsNotes

1. Don't do this. (1)
2. Attributed to WikiPedia: Rick\_Rashid. (2)
3. The one case of logging that I personally had some unhappy involvement with many years ago took the form of a line printer in a locked machine room. (3)

---

2014-06-17 11:58