Stanley Wei

CS 111

Reiher, Su 2024

Operating System Principles 3
Sec. 1: Operating Systems 3
Principles 3
Operating Systems Basics 3
Sec. 2: OS Abstractions & Services 7
OS Abstractions 7
OS Services 9
OS Interfaces 12
Sec. 3: Processes, Execution, and State 15
Processes 15
Address Spaces 16
Handling Processes 21
Limited Direct Execution 23
Sec. 4: Scheduling 29
Scheduling Processes 29
Scheduling Performance 32
Scheduling Algorithms 33
Non-Preemptive Scheduling 34
Preemptive Scheduling 36
Memory Management 42
Sec. 5: Memory Allocation 42
The Memory API 44
Memory Allocation Strategies 46
Fixed Allocation 46
Dynamic Allocation 46
Buffer Pools 50
Memory Allocation Problems 51
Memory Leaks 51
Fragmentation 52
Sec. 6: Virtual Memory 55
Virtual Memory 55
Memory Management Techniques 58
Swapping 58
Paging 59
The Memory Management Unit 59

Demand Paging 60
Page Replacement 64
Threads and Synchronization 69
Sec. 7: Threads and IPC 69
Threads 69
Inter-Process Communication 72
Types of IPC 75
Sec. 8: Synchronization 77
Synchronization 77
Critical Sections 78
Mutual Exclusion 80
Implementing Locking 81
Asynchronous Completion 83
Condition Variables 84
Waiting Lists 85
Semaphores 89
Sec. 9: Synchronization Problems 92
Locking Problems 94
Deadlock 97
Avoiding Deadlock 98
General Synchronization Bugs 101
Operating System Features 104
Sec. 10: Devices and Device Drivers 104
Device Drivers 104
Device Utilization 106
Sec. 11: File Systems 111
File Systems and the OS 112
File System Control 114
File System Structure 115
Allocation and Performance Strategies 118
Naming and Reliability 121
Sec. 12: Virtual Machines 128
Sec. 13: Operating System Security 132

Operating System Principles

Sec. 1: Operating Systems

Principles

- Complexity management principles
    - Hierarchical decomposition & layered structure
    - Modularity and functional encapsulation
    - Abstractionand information hiding
    - Interface contracts
    - Progressive refinement
- Architectural paradigms
    - Separation of policy (how to decide which clients get which resources) vs mechanism (how to actually effect these decisions) for resource managers
    - Indirection, federation, and deferred binding (in the case of multiple implementations of a general object class)
    - Dynamic equilibria for resource allocation
    - Importance of data structures for performance
- OS history
    - Standardization

Operating Systems Basics

- Running program
    - Program execution: fetch an instruction, decode, then execute
        - Von Neumann model of computing
- Basic parts of a computer
    - Registers (?): Read/write binary words
    - CPU
        - Basic operations: Move, add, jump; fancy: SQRT
    - Bus: Read/write data
        - Request sense [input from input devices]

    - Flash drive: Read/write blocks of data
        - Blocks: Fixed-size chunks of memory
    - Input/output (I/O)
        - Mouse: Report X/Y movements, button presses
        - Screen: Write to pixels
        - Hard drive, external drives
- Computing: Want to use simple pieces to perform complex operations
    - Operating systems: Translates complex operations to simple hardware operations
        - Sits between hardware and everything else; abstracts away complexity
        - Controls network interactions
        - Allocates computing power to many applications at once
- Operating systems are systems software - used to provide support for higher-level applications, rather than individually on its own
    - Higher-level applications: higher-level system software, user processes
    - Is typically the only piece of software interacting with the hardware
        - On rare occasions: may allow other software to interact with hardware
        - Provide callable interfaces (via system calls or standard libraries, e.g.) for application use
        - Device drivers - software specifying low-level operations for interfacing with external devices
- Importance of operating systems
    - Every computing device has an operating system, even things not typically thought of as computers
    - Offer virtualization: transform physical resources (e.g. processor, memory, etc.) into more powerful, easy-to-use virtual versions
        - OS acts as a virtual machine
    - Acts as resource manager - handles allocation of resources (e.g. memory, disk) across many concurrent processes
- OS functions
    - Virtulizing CPU - scheduling
    - Virtualizing memory - virtual memory
    - Concurrency (multithreading)
    - File system - software for disk management, storing user files

- Binary distribution model: OSes typically distributed in binary
    - OS written in source, compiled to binary; binary = ready to run
        - 1+ binary distributions per ISA
    - Binary model for platforms support: device drivers added after-market, depends on binary distribution
        - Device drivers vary between different devices (e.g. different keyboards)
- Binary configuration model helps eliminate manual/static configuration
    - Allows one distribution to support all devices
    - Hardware can be discovered automatically, corresponding drivers loaded
        - Via self-identifying buses: PCI, USB, etc.
    - Automatic resource allocation: rather than fixed-size resource pools, dynamically (re)allocate resources on demand
        - For allocation of any shared resources, e.g. RAM, memory
- Functionality in the OS: as much as necessary, as little as possible
    - OSes must be free of crashes - very expensive to develop, maintain
    - Functionality in the OS if:
        - Requires use of privileged instructions
        - Requires manipulation of OS data structures
        - Must maintain security, trust, or resource integrity
        - Performance excuse: some things faster if done within OS
    - Functionality in libraries if:
        - Are services commonly needed by applications, but do not need to be implemented inside the OS
- Popular OSes:
    - Windows - most popular PC OS, some servers/small devices
    - MacOS - Apple products
    - Linux - industrial servers, embedded systems
- Software generation
    - Classes of files
        - Source modules - later translated into machine language by compiler
        - Relocatable object modules - compiled/assembled from source modules, but not yet complete programs

        - Libraries - collections of object modules
        - Load modules - complete programs, e.g. from combining object modules by assembler; are ready to be loaded (via OS program loader) & executed
            - Linkage editing: given a collection of relocatable object modules, to turn into a program, first need to:
                - Resolution - resolve unresolved external references (i.e. Locate in specific libraries)
                - Loading - lay text/data segments into single virtual address space & note locations of each symbol
                - Relocation - go through all relocation entries in loaded object modules and set to correct addresses
    - Executable and linkable format (ELF) - object module format for Unix/Linux
        - Contains consecutive sections: header (describing type/size/locations of other sections), code/data, symbol table, and set of relocation entries

Sec. 2: OS Abstractions & Services

OS Abstractions

- Major function of OS: offers abstract versions of resources, rather than physical resources
    - OS implements abstract resources via physical resources
        - Ex: processes (abstraction) via CPU/RAM (physical), files via flash drive
    - Abstractions simpler, more suited to programmers/users
        - Easier to use, compartmentalize complexity, eliminate behavior that is irrelevant to the user
            - Make behavior more convenient
        - Hide variations between hardware/software of different machines, provide common unifying model
            - Ex: different printer drivers look the same from user perspective
- Serially reusable resources - can be used by multiple clients, but only one at a time (time multiplexing)
    - Require access control (ensuring exclusive use)
        - Analogy: printers
    - Requires graceful transitions between users - switches to hide the fact that resources used, belong to someone else
        - Prevents users from accessing until it is “their turn”; make resources appear “as-new” when accessed
- Partitionable resources - divided into disjoint pieces for multiple clients (spatial multiplexing)
    - Requires access control to ensure containment (cannot access resources outside of your own partition), privacy (nobody can access your resources
        - Ex: RAM, flash drives
    - Still requires graceful transitions when reallocating resources
        - Namely: RAM used by a process may later belong to another process; want RAM cells to become “as new” when transferring
- Shareable resources - usable by multiple concurrent clients
    - No notion of owning resources, no waiting for access to resource
    - May involve (effectively) limitless resources

            - Ex: copy of operating system, shared by processes
- Critical/core OS abstractions: memory, process, communications abstractions
    - Memory abstractions - lots of applications need memory
        - Complicating factors:
            - Persistent vs transient memory
            - Size of memory operations: size requested vs physical memory
            - Coherence and atomicity
                - Coherence - putting related functionality close together, e.g. within the same modules
            - Latency - CPU is much faster than RAM, is much faster than most hardware devices
            - Many different physical devices
        - OS has to work with physical devices, not abstractions
    - Interpreters - something that performs commands
        - Interpreter components
            - Instruction reference - tells interpreter which instruction to do next
            - Repertoire - set of things interpreter can do
            - Environment reference - current state on which next instruction should be performed; registers, e.g.
            - Interrupts - situations in which instruction reference pointer is overridden
        - Physical level: processors/CPUs are interpreters
        - OS provides higher-level abstractions - processes
            - OS maintains program counter (instruction reference) for each process
            - Source code specifies repertoire
            - Stack, heap, register contents are environment; OS maintains pointers to all
            - No other interpreters can interfere with process's resources
        - Implementing process abstraction in the OS:
            - Difficult for multiple processes - OS has limited physical memory, only one set of registers (or one per core)
                - Processes need to share CPUs/cores

            - Schedulers share CPU among various processes
                - Use special memory management hardware, software to multiplex memory use; give each process, illusion of full use of memory
                - Access control mechanisms for other memory abstractions
    - Communication: OS may need to provide communication links between interpreters (on the same or different machines)
        - Physical level: memory/cables
        - Abstract levels: networks, interprocess communication mechanisms
        - Communication links distinct from memory
            - Highly variable performance, often asynchronous
            - Receiver may only perform operation because send occurred, unlike a typical read
            - Remoet machine - additional complications (e.g. losing communication)
        - Implementing communication link abstractions in OS
            - If both ends are on same machine: use memory for transfer by copying (relatively simple)
                - May play memory management tricks to optimize costs of copying memory

OS Services

Operating services offers important services to other programs

- Typically offered via abstractions
- Basic categories:
    - CPU/Memory abstractions:
        - Processes, threads, virtua machines
        - Virtual address spaces, shared segments
    - Persistent storage abstractions
    - Communication abstractions:
        - Sockets, pipes

OS layering - modern OSes offer services via layers of software/hardware

- Higher level abstract services offered at higher software layers; lower-level services deeper in the OS
    - Effectively: OSes on top of other OSes
    - Everything mapped down to hardware
- Layers:
    - Highest (Applications): user/system applications, OS services
    - Middle (Application binary interface): General libraries, drivers, OS kernel
    - Lowest (ISA): devices, privileged instruction set, general instruction set

Service delivery

- Service delivery via subroutines: access services via direct subroutine calls
    - Implementation
        - Push parameters, jump to subroutines, return values in registers on the stack & return to previous stack frame
    - Typically operates at high layers
    - Advantages: extremely fast, run-time implementation binding
    - Disadvantages: all services implemented in same address space, limited ability to combine different languages, can't use privileged instructions
- Service delivery via libraries
    - Library - collection of object modules; single file containing many files (analogous to zip/jar), usable directly without recompilation
        - Most systems come with standard libraries: system services, encryption, etc.; more via add-ons
        - Can build custom libraries
        - Linux: can use ar() syscall to create, update, or examine
    - Libraries live below application binary interface, above OS kernel
- Service delivery via system calls: forces entry into operating system
    - Parameters/returns operate similar to subroutine calls; implementation in shared/trusted kernel
    - Advantages: able to allocate/use new/privileged resources, share/communicate with other processes
        - Can do anything the computer can, effectively
    - Disadvantages: 100-1000x slower than subroutine calls

    - Typically done for functions that require privilege
        - Privilege:
            - Privileged instructions (e.g. interrupts, IO)
            - Allocation of physical resources, e.g. memory
            - Ensuring process privacy/containment
            - Ensuring integrity of critical resources
    - OS may out-source called operations to other applications (system daemons - background processes w/ no direct control, server processes)
    - Some plugins (device drivers, file systems, network protocols) may be less trusted by the OS, given less permissions
        - Typically consist of additional software "plugged into" kernel
    - System calls operate at the OS kernel layer

Characteristics of libraries

- Advantages: reusable code makes programming easier, act as building blocks (encapsulate complexity)
    - Multiple bind-time options: static (include in load module at link time), shared (map into address space at exec time), dynamic (choose and load at runtime)

Sharing libraries

- **Static libraries** added to a program's load module
    - Each load module has its own copy; increases size of each process
    - Programs must be re-linked if library is changed
- Instead, make each library sharable code segments (**shared libraries**)
    - One in-memory copy, shared by all processes; kept separate from load modules, OS later loads library along with program when program is executed
    - Advantages:
        - Reduced memory consumption, faster start-up
        - Libraries are easier to update
    - Limitations:
        - Not all modules work in a shared library
            - No global data storage in a shared library, since all processes can read the same memory

        - Shared libraries always added into program memory, whether they are actually needed or not
        - Called routines must be known at compile time
- **Dynamic libraries** - similar to shared libraries, but are only loaded into memory (via dlopen) when dynamic libraries are actually called in runtime
    - May never be loaded if library is never called
        - If a library was previously loaded but an app never cals it, will never be linked in memory
    - Implicit vs explicit loading - the specific library loaded may be automatically loaded without application attention (implicit), or application may request specific library to be loaded (explicit)

Some services may be offered outside the kernel

- Not all trusted code must be in the kernel; may not need to access kernel data/privileged instructions
    - Ex: email clients
- Some services are somewhat privileged processes (can modify user credentials, directly execute I/O); some are just trusted (given only limited sets of operations)

OS Interfaces

- OS interfaces - interfaces between OS and other programs allow OS to offer services
    - OS meant to support other programs via services; need interfaces to allow users/programs to access
- Two important interfaces: API
    - Application Program Interfaces (APIs): specified at source level, allows programmers to invoke services with parameters via system calls
        - For programmers; allows for writing programs for OS
        - System calls - expose certain pieces of functionality (e.g. accessing file system, creating/destroying processes) to user applications
        - Ex: allows opening/closing files
        - Advantages

            - Software portability - can recompile for any architecture, high-level system calls are later linked with OS-specific libraries
            - Expect an API-compliant program to compile and run on any compliant system (ISA)
    - Application Binary Interfaces (ABIs): binary interface, is both ISA-specific + OS-specific
        - Allows for specifying dynamically loadable libraries (DLLs), data formats, calling sequences, etc.
            - Binds API to a hardware architecture - specifies binary-level behavior/instructions of a system call, e.g.
            - Commonly used by compiler and linker programmers, e.g.
        - Any ABI-compliant program will run on any compliant system (i.e. sharing the same ISA and OS) without any modifications (i.e. same bytes)
            - Different ISA/OS -> no longer applies
        - Example contents: binary representation of data types, instructions for subroutine/syscall calls, stack-frame structure, etc.
        - Benefit: installed programs (binaries), once compiled for a given ABI, is guaranteed to work on any computer sharing the same ABI; ABIs for users
        - Data type, load module format, exception propagation mechanism, system call invocation mechanism
            - Not: include files, routine names, parameter details, return details, error return codes, operation semantics
- Libraries & interfaces
    - Normal libraries (shared & otherwise) accessed through API
        - Source-level definitions of how to access the library; readily portable between different machines
    - Dynamically loadable libraries also called through an API
        - However, dynamic loading mechanism is ABI-specific
            - Due to how differences in ISA, OS affect behavior
- Interfaces & interoperability
    - Strong, stable interfaces allow programs to be distributed, operate together

        - Allows OS evolution (adding new features), while preventing OS upgrades from breaking existing systems so long as OS-program interface does not have features removed
    - When operating outside of interfaces, may be tempted to rely on side effects of actions not specified by API
        - OS developers only required to maintain what is specified by API; side effects (even beneficial ones) may change version-to-version
        - May result in unexpected behaviors + causes hard to find bugs
- Updating interfaces
    - A few ways to ensure backwards compatibility
        - Interface polymorphism
        - Versioned interfaces

Sec. 3: Processes, Execution, and State

Processes

- Prev. lecture: interpreter as an abstraction of "something that does work"

Process - running version of a program

- Is a type of OS **object**; characterized by properties (**state**) and **operations**
    - States distinguish different objects; describes current condition of an object
        - Content of state depends on object type, potentially complex; however, all states representable as set of bits -> can be saved, restored from bits
            - Can talk about state subsets (e.g. scheduling states)
        - Potential OS object states: scheduling priority of a process, current pointer into file, completion condition for an I/O operation, list of allocated memory pages to a process
        - OS object states primarily maintained & managed by OS, not directly managed by user code
            - User code must ask OS to access, alter state of OS objects
- Note: OS itself is not a process; processes are built by OS

Each process has some memory addresses reserved for private use; called its **address space**

- Address space made up of all memory locations the process can access
    - A process can access a memory location/address if and only if that location is in its address space
    - Address space located in RAM, typically
- Modern OSes: pretend a process' address space includes all of memory (virtual memory); serves as abstraction of physical memory (virtualizing memory)
    - Process sees virtual addresses, not physical addresses

Programs vs processes

- Program - compiled version of an application, but not currently running
    - Contains: ELF header (executable and linkable format; specifies useful information for converting program -> process, e.g. target ISA), additional load and info sections

        - Load sections - information that needs to be stored in address space of the process
            - Specified by type, load address, and length
            - Ex: compiled code segments (as machine instructions), initialized data values
        - Info sections - useful information that is not directly loaded in/used by the process
            - Ex: symbol table (converting variable names to memory locations) for debugging purposes
- Process - running versions of a program
    - Don't exist until the program is actively run
    - Have an allocated address space; in addition to contents of program, contains shared libraries and private stack

Address Spaces

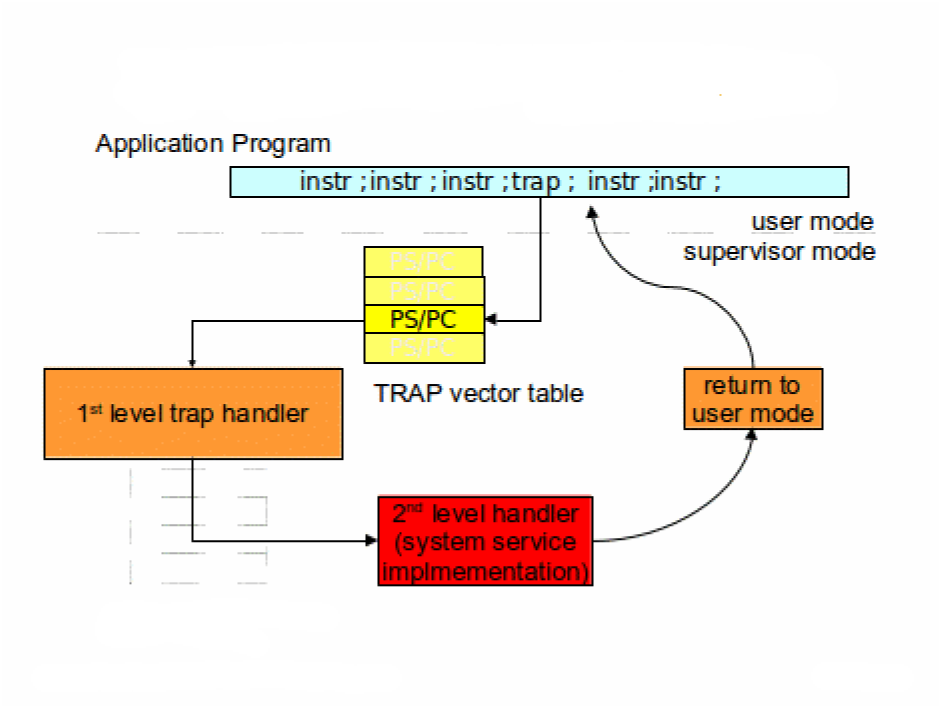
- Address space contain: shared code (when two copies of the same program are running, code is shared between both), private data, shared libraries, private stack
    - Heap/data - contains malloc'd data, dynamic data structures
    - Stack - contains local variables, arguments to routines, return values, etc.
- Address space layout: different types of memory elements may have different requirements
    - Stack - private, readable, writable, size managed by OS
    - Data/heap - private, readable, writable, size managed by syscall
    - Code - sharable, fixed-size, executable
- Each OS has its own strategy for how to place process memory segments
    - Ex (Linux): stack starts at highest address, grows downward to lower addresses; data segment starts at lowest, grows upward [toward stack]
        - Code segments statically/fixed sized, immediately before data
        - OS ensures data and stack do not intersect
        - Differs between OSes
            - Rec: is an abstraction (stack and data might be stored in completely separate locations in physical memory

- Address space code segments
    - Start with a load module (from a linkage editor, typically - all external references resolved, all modules combined into a few segments, everything except DLLs),
    - Code segments are loaded into memory [RAM], may be copied - read in from load module (possibly with small changes - memory addresses, e.g.)
        - Are read/execute only, sharable
- Address space data segments
    - Data segments also initialized in address space
        - Initial contents of data segments copied from load module, BSS segments (chunks of memory needed by process - known size, unknown values) initialized to all zeroes
        - Data segments are read/write, process-private; may be grown or shrunk by process via sbrk system call
    - Contain global variables
- Processes and stack frames
    - Modern PLs are stack-based - every procedure call allocates a stack frame
        - Stack [LIFO] grows & shrinks as functions are called & return
            - Always have one stack frame - active stack frame (furthest out)
                - Only active stack frame is being used at any given moment, changes as stack frames added
    - Stack frames contain local variables, function parameters, return values
        - Behind the scenes: stack frames contain save, restore registers
    - Stack is part of process state, must be preserved as part of process state
    - Most modern CPUs have ISA stack support for facilitating stack operations
- Address space stack segment can grow and shrink
    - Size is unknown ahead of time; typically, OSes allocate large amount of empty space initially, allocate more if needed
    - OS manages process stack segments, tracks state; uses stack pointer to locate
        - Stack segment created at same time as data segment
        - Some OSes dynamically extend stack as program needs
    - Stack segments read/write, process-private; typically not executable
- Libraries

    - Static libraries added to load module as part of code, a new copy for each load module; must re-link to update
    - Shared libraries stored as a single shared in-memory copy, separate from load modules; loaded by OS along with program
        - Reduces memory use + speeds up program loads; makes library upgrades easier
- Other process state
    - Register values - storing register values when processes paused, e.g. + program counter, processor status, stack pointer, frame pointer
    - Process' own OS resources: open files, working directory, locks
    - OS-related state information: process execution time
    - Memory
- "Other process state" not part of any code/data/stack segments; needs OS data structure to handle
    - In OSes, stored in OS-managed **process descriptors** - basic OS data structure for dealing with processes
        - Stored for any active process, even idle ones
    - Stores all process-relevant information: state to restore, references to allocated resources, information to support process operations
        - Only managed by OS; used for scheduling, security decisions, allocation issues
    - Ex: Linux has Process Control Block (PCB)
        - Stores: unique process ID [PID], process state (e.g. running) & priority, program counter, registers, I/O information, address space information, + more
        - Not: code, stack, data

- Processes virtualize CPU - by swapping between running different processes (**time sharing**), OS creates illusion of many virtual CPUs existing
    - Context switch - OS stops running one program, starts running another on the same CPU
- Subroutine linkage/calling - ISA and potentially language-specific, but have some common elements

    - Common elements:
        - Parameter passing - done via dedicated registers, e.g.
        - Subroutine call - save return address of caller on stack, transfer to called entrypoint (via call instruction, e.g.)
        - Register saving - save all non-callee-saved registers on the stacks they can be restored upon return
        - Allocating space for local variables of callee
    - Similar elements when returning
        - Return value - return value placed in expected location (e.g. register)
        - Pop local storage (i.e. stack frame) of callee off of stack
        - Restore registers to callee values
        - Subroutine return - transfer control to saved return address (via ret instruction, e.g.)
- Traps and interrupts
    - Vs regular procedure calls:
        - Linkage conventions for procedure calls under software control; traps/interrupts under hardware control
        - Procedure calls may result in some value (e.g. return register) being changed; trap/interrupt typically restores state entirely
    - Traps issued by processes; interrupts via external devices


Application Program
instr ;instr ;instr ;trap ; instr ;instr ;
user mode
supervisor mode
PS/PC
PS/PC
PS/PC
PS/PC
TRAP vector table
1st level trap handler
return to user mode
2nd level handler (system service implmementation)

Handling Processes

- OS handles processes: creating, destroying, and running processes
    - Other actions: wait, misc. control (e.g. suspend), status, etc.
    - Are part of the process APIs for most OSes
- Creating processes
    - Processes created by OS, upon request of another process
        - Uses some method to initialize state, set up program to run
    - **Parent process** specifies program to run (**child process**), aspects of initial state
    - OS creates a new process descriptor
        - Placed in OS **process table** - OS data structure used to organize all active processes (only one process table, typically)
            - Process table contains one entry (e.g. PCB) for each active process
    - OS creates an address space for each new process - allocates memory for code, data, and stack
        - OS loads program code, data into new segments + initializes stack segment
        - Sets up initial registers: PC, PS, SP
        - Initializes other aspects, e.g. input/output
            - Unix - three open file descriptors (stdin, stdout, stderr)
- Choices for process creation:
    - 1: Start with a "blank" process - no initial state or resources
        - Have some way to fill in vital information: code, PC, priority, etc.
            - System call for creation provides info for process setup; at least code, typically more
                - System call includes name of program to run
                - Other critical info (e.g. environment info, priorities, etc.) given by parameter
        - Windows approach - CreateProcess() system call
            - Very flexible way to create new process
    - 2: **Process forking** - Use the calling process as a template (copy parent's process descriptor)
        - Copies code, PC, etc.
        - Unix/Linux approach - fork() system call

            - No parameters, simply clones parent process (identical, but with new ID and different address spaces)
            - Initially: parent and child have same code and program counter, will run same code
                - Need code to change child behavior
        - Advantage: makes creating pipelines easier (can run code between when child is created, and application is run; e.g. redirect STDOUT)
            - Otherwise, done for historical reasons
        - Forking and memory
            - Code is shared between parent and child; stack is private for both parent and child (but initially identical)
                - Data - potentially large and expensive to copy, has different operation (initially shared, but once either parent/child writes, copied out)
        - Want child process to do something different
            - Via exec() system call - "remakes" process, changes code associated + resets most of state (e.g. open files)
                - Replaces code, stack, data
                - Must load new set of code; initialize child stack, PC, and other structures to start fresh program run
        - Linux wait() syscall: causes parent to delay execution until child finishes executing and exits
- Destroying processes
    - Once process reaches last instruction (or is killed; by OS or by another process, e.g.), terminates
        - Most processes terminate; all do if machine shuts down
    - When process terminates, OS needs to clean it up - clean up and reclaim all of its resources
        - Reclaim any resources held: memory, locks, access to hardware devices
        - Inform any other process that needs to know - processes waiting for interprocess communication, parent/child processes
        - OS removes process descriptor from process table, reclaims its memory

Limited Direct Execution

- Running processes
    - Three possible states: running, ready, blocked
    - Processes must execute code/instructions to do their job
    - Issue: processes need access to processor core via OS; typically, more processes than cores
        - Processes need to share cores - can't all execute instructions at once, but each process will need to be put on a core eventually
    - Running processes on a core (CPU)
        - First, must initialize core's hardware - initialize registers, either to initial state (if process has not been run before) or whatever state the process was in last (if process is resuming)
            - Information copied from process descriptor
        - Must initialize more information: stack and stack pointer, memory control structures, program counter
    - After initialization: process is running
        - **Limited direct execution** - (most) instructions executed directly by process on the core without OS intervention
            - For performance reasons - removes overhead of OS (faster)
        - Some instructions cause **trap** to operating system - privileged instructions, require supervisor mode; done by OS
        - Generally: want to maximize direct execution, minimize time spent by OS
            - Only interrupted by traps (system calls), timer interrupts (for time sharing)
    - If process can't/shouldn't run an instruction, causes exception
        - May have routine exceptions: end of file, arithmetic overflow, conversion error, etc.; checked after each operation
        - May have unpredictable exceptions: segmentation fault (e.g. dereferencing NULL), user abort, hang-up, power failure
            - Called **asynchronous exceptions**; unpredictable, may be outside of program's flow, cannot be checked for
                - Some languages - try/catch operations

    - Traps - certain exceptions, when caught, cause explicit transfer of control to OS
        - Supported by hardware, OS
        - Used for running privileged instructions + system calls (requests for OS services)
            - Once complete, returns to instruction after system call
        - Modern processor design - explicit trap instructions designed into CPU
            - Are privileged instructions - once performed, causes a trap to move to operating system
                - Indicates to operating system that program wants to perform a privileged instruction
            - Have conventions for indicating desired system call
                - Ex: r0 register contains system call numbers, r1 register points to arguments
                    - Upon return: r0 contains return code, condition code (indicating success, failure)
                - System call indicated by number, not address; indirectness offers additional protection
        - Many system calls in modern OSes -> use **gates** to enter system calls
            - Every system call has its own gate
- Two levels of processor mode - user mode (running code has restrictions on actions) vs kernel mode (no restrictions; can perform privileged operations)
    - Privileged vs user mode determined by a single bit (**processor status word**)
        - Switches on special occasion
    - OS/kernel runs in kernel mode
        - Running traps
            - Program runs in user mode; hits trap instruction
            - Upon hitting trap, will enter supervisor mode and reference **TRAP vector table** to find information on how to deal with system call - find PS, PC, e.g.
                - TRAP table contains all ways to enter supervisor mode; also contains timer interrupts, e.g.
                    - Built at boot time; startup runs in kernel mode

        - TRAP table will point to 1st level trap handler - has determined instruction is a TRAP
            - 1st level trap handler will look at r0 and r1, use **system call dispatch table**
                - Contains one entry for each system call; referenced via number in r0
                - Each entry contains **trap gate** - pointer to a system call-specific piece of OS code
        - Next, moves to 2nd level handler - executes all code necessary to perform the system call in question
        - Once complete, returns to user mode (limited direct execution)
    - Trap handling divided between hardware (CPU), software
        - Hardware portion
            - Trap table lookup
            - Load new processor status word (to switch to and from supervisor mode)
            - Push PC/PS of program causing trap onto stack
            - Load PC with address of 1st level handler
        - Software portion
            - 1st level handler pushes all other registers, gathers info, selects 2nd level handler
            - 2nd level handler deals with the problem - handles the event, kills the process, returns, etc.
                - May require lots of OS code
    - Stack-based architecture -> when running OS code, needs its own stack
        - Issue: don't want application code to be able to view OS stack
    - For every process, have two stacks: **user-mode stack** (application's own code) and **supervisor-mode stack** (for system calls)
        - User-mode stack contains all stack frames from application computation; grows down
        - Supervisor-mode stack
            - Upon performing system call, pushes:
                - User mode PC and PS

                - Saved user mode registers
                - Parameters to syscall handler
                - Return PC
            - After: have system call handler stack frame
            - Upon finishing, supervisor mode stack is emptied and return to user mode
- Returning to user mode: via **return-from-trap** instruction
    - Return opposite of interrupt/trap entry:
        - 2nd level handler returns to 1st level handler
        - 1st level handler restores all registers from stack, use privileged return instruction to restore PC/PS
        - Resumes user mode execution at next instruction
    - Saved registers may be changed before return: change stacked user r0 to reflect return code, stacked user PS to reflect success/failure
- Some system call instructions require waiting (ex: accessing a peripheral flash drive)
    - Normally, application can't run until system call returns (is blocked)
- In some cases - may want to do something else (run another application, e.g.) while waiting
    - Need to remember that instruction will return at some point + which program called it
    - Use **event completion callbacks** - when event completes, rerun the application (points to PC to resume from)
        - Built into OS
- OS defines numerous types of signals: exceptions, operator actions, communication
    - Processes can control how they handle signals - ignore a signal, designate a signal handler
    - In many cases, have default actions for a signal (e.g. kill/coredump process), run in the absence of a signal handler
    - Are analogous to hardware traps/interrupts, but are OS-required and delivered to processes (rather than the other way around)
    - Can send signals to individual processes, or larger process groups

                - Process signal() syscall catches signals
    - Managing process state is important
        - Shared responsibility
            - Process takes care of its own stack, contents of its memory/heap (implication: can screw up stack if not careful, OS doesn't care)
                - OS must ensure screwed-up stack/memory will not cause problems for any other application
            - OS keeps track of process-allocated resources
                - Allocated RAM segments, open files/devices + application permissions, supervisor stack, etc.
    - Blocked processes - processes become blocked when they ask for something to happen
        - Means process cannot run next instruction yet, waiting for something not under process' own control
            - Process can ask to be blocked itself, via system call (would need to arrange to be unblocked; cannot unblock itself)
            - Ex: waiting for peripheral device I/O or system call result
                - Peripheral devices much slower than CPU, typically
        - OS keeps track of whether processes blocked in process descriptor
            - Not running =/= blocked; may just be idle or waiting to be scheduled
        - Blocked/unblocked act as notes to scheduler, tells schedule not to choose that process
            - OS responsible for unblocking processes
        - Blocking typically done by OS' resource manager
            - Blocks process if it asks for an unavailable resource
                - Changes scheduling state, calls scheduler and yields CPU
            - Unblocks process once resource is available
                - Resource available -> receives message from networking resource, performs lookup for receiving process
                - Changes scheduling state to "ready", notify scheduler of a change in state
- OSes store various info about processes
    - Register context - stores register contents for stopped processes

- User permissions delineate which processes can be controlled by a particular person
    - Generally, users only control their own processes; superusers can control all processes + have other permissions
- Limited direct execution (LDE) protocol
    - Boot time: ernel initializes trap table, CPU remembers location for later use
        - Done within kernel mode
    - When running: kernel sets up program (creates process, allocates memory, etc.); then uses return-from-trap to start process execution
        - Process syscalls trap back to OS; OS handles trap, returns again via return-from-trap instruction
        - Once process exits, exit() call traps to OS for cleanup
- Traps vs interrupts
    - Traps - issued by processes, interrupts - by peripheral devices

Sec. 4: Scheduling

- Multiprogramming - a computer has multiple processes ready to run at any given time, and the OS switches between running them
    - Time sharing - many users might concurrently be using a machine and waiting for response on their tasks
    - Require some mechanism on how to choose what process to run
- Scheduling: an OS often has to make choices about what to do next
    - Namely: for a resource that can only serve one client at a time (e.g. processor core, flash drive, external device, etc.), but with multiple potential clients, who gets to use the resource and for how long?
        - Mostly, but not all, hardware resources
        - Ex: jobs to run on an idle core, in what order to handle block requests for a flash drive, order to send messages over network
- General strategy: choose a goal to achieve, then design a scheduling algorithm that is likely to achieve those goals
    - Different scheduling algorithms can have different effects
        - Different goals have different effects

Scheduling Processes

- Scheduling processes - want to determine allocation of processes to cores
    - Potential scheduling goals - maximize throughput (time spent on user code, rather than OS code), minimize average waiting time, ensure “fairness” (minimize worst-case waiting time)
        - May have explicit priority goals - items labeled with relative priority
        - Real time scheduling - items may have associated deadlines to meet
    - Different systems, different scheduling goals
        - Time-sharing - users sharing a computer -> fast response time to interactive programs, equal distribution of CPU
        - Batch - want to complete a set of tasks -> maximize total system throughput, don’t worry about dleays of individual processes

        - Real-time - critical operations must happen on time, non-critical operation may not happen at all
        - Service Level Agreement (SLA) - share resources between multiple customers, make sure all agreements (e.g. cloud bandwidth) are met across all users
- Process queue - OS typically keeps queue of processes ready to run (**process/task list**)
    - Processes ordered by whichever runs next; order determined by scheduling algorithm used
        - First process in the queue is next to be scheduled
    - Processes not in the queue are either: not in the queue, at end of queue, or ignored by scheduler
- Scheduling contains **policy**, **mechanism** component
    - **Policy** - means of choosing which process should be run
        - Different scheduler -> different policy
    - **Mechanism** - once scheduling decisions are made, want to run processes [on CPU] according to the scheduler's plan (**dispatching**)
        - Dispatching - moving jobs onto/off of processor core based on policy
        - Typically policy-agnostic; independent of what policy decisions are made
            - Primarily consists of mechanical/hardware aspects
            - Ex: OS process queue structure is the same across all potential policies

Two types of scheduling: **preemptive** vs **non-preemptive**

1. **Non-preemptive**: Once a process is running on a core, will continue running until it terminates or voluntarily stops running
    - Process can use the core until it finishes, effectively
    - *Pros*: Low scheduling overhead, generally high throughput, conceptually simple
    - *Cons*: Poor response time, generally not "fair", may make real-time or priority scheduling difficult [less suitable]
        - Bugs (e.g. infinite loop within process) can cause machine to freeze
2. **Preemptive**: Processes may be interrupted partway through (temporarily halted) to run something else, based on scheduler decisions
    - Temporarily halted - save bits corresponding to process' state

    - *Pros*: Good response time, fair usage, good for real-time and priority scheduling
    - *Cons*: More complex, possibly higher overhead & lower throughput (more interruptions, context switching)
        - Requires ability to clean halt processes + save process state

- Scheduling the CPU
    - New process enters ready queue
    - Upon reaching front, once core is made available, dispatcher will prepare to run process on that core
    - Context switcher changes process running on core, to the new process
        - Initializes registers, e.g.
    - Process actions
        - Process may voluntarily yield the core
            - Typically, will return to end of ready queue
        - Process may want to use a hardware/software resource (to send a message, read file, etc.)
            - Process sends resource request to resource manager, needs to wait for resource manager to complete operation
            - Once resource is granted, re-enters end of ready queue and waits until it is scheduled again
- Regaining control
    - Consequence of LDE model: when process running on CPU, OS is not running; need a way for OS to regain control (to perform context switch, etc.)
        - Cooperative approach: wait for processes to use syscalls to transfer control of CPU back to OS; via yield syscall, e.g.
            - Historical approach, assumes processes will eventually yield
            - In case of infinite loops or never yielding, requires reboot to regain OS control
    - Non-cooperative: via **timer interrupts**
        - Can program timer to raise an interrupt at regular periods; when interrupt raised, process is halted and OS' pre-configured interrupt handler runs (OS regains control of CPU)
        - Timer initialized by OS during boot sequence (privileged operation)

Scheduling Performance

- Scheduling important system activities has major effect on performance
  - Various aspects for performance - may not be able to optimize across all
    - Want to be able to characterize performance of a scheduling system, operating system
    - Ex: response time, throughput, etc.
    - General comments on performance
      - Performance goals should be quantitative and measurable
        - Need to be both in order to optimize
      - Metrics - way and units in which we measure (very complex)
        - First choose a characteristic to measure, find a unit to quantify it, then define process for measuring
        - Ex: CPU seconds/hour for time spent running user code
  - Load matters - systems with less work to do (light load) generally have different behavior than systems with more work (heavy load)
- Quantifying scheduler performance
  - Candidate metrics
    - Throughput: processes/second
      - Flaw: different processes need different amounts of time to run
      - Process completion time is not controlled by scheduler
    - Delay: milliseconds
      - Q: what kind of delay?
        - Time to finish a job (turnaround time)
        - Time to get some response
      - Flaw: some delays are not the scheduler's fault
        - Ex: time to complete service request, time to wait for busy resource
    - Generally: software cannot optimize things that it doesn't control
  - Generally (in many contexts) - ideal is that throughput increases as load increases up to a certain point (maximum possible capacity), at which point throughput remains constant for all higher loads

        - More realistically/typically: throughput peaks sooner and at lower true maximum capacity, decreases as load increases thereafter
        - Reasons for non-ideal throughput:
            - Scheduling is not free - requires overhead to dispatch process, resulting in less time (per second) for running processes
        - Minimizing performance gap: reduce overhead/dispatch, minimize number of dispatches per second
        - True for many contexts that have to do with throughput (e.g. networking)
    - Response time: ideal is that response time (delay) increases linearly with load, reality is that response time increases exponentially (non-linearly)
        - Reasons for non-ideal response time:
            - Real systems have finite limits, e.g. queue size
                - Once limits are exceeded, requests are typically dropped -> infinite response time
                - Some requests may have automatic retries (e.g. TCP), but those may drop as well
            - In general: if load arrives much faster than it is serviced, lots of things get dropped
            - Heavy load may cause overheads to explode if not careful
                - Even making the decision to take/drop a request, takes time -> even queued-up process is never run due to overhead
                - Happens in networking, e.g.
        - ***Graceful degradation*** - once system is overloaded (i.e. no longer able to meet service goals), can continue service with degraded performance
            - Maintain performance by rejecting work
                - Hope: overload is temporary, can resume normal service once load drops
            - When not overloaded - never allow throughput to drop to zero (i.e. be spent on overhead instead), allow response time to go to infinity

Scheduling Algorithms

- Non-preemptive scheduling - scheduled process runs until it yields CPU
    - Works well for simple systems: small numbers of processes, natural producer-consumer relationships
        - Good for maximizing throughput
    - Relies on processes to voluntarily yield - otherwise, processes may starve others (piggy processes) or lock up entire system (buggy processes)

<u>Non-Preemptive Scheduling</u>

- First-come first-served (FIFO) - simplest scheduling algorithm
    - Ready queue is just ordered in the order by which processes are added
        - Potentially high and highly variable delays, depending on how long each process takes + order within ready queue
    - Advantage: all processes will eventually be served (barring bugs)
        - Very simple, but poor response time
            - Convoy effect - if a very demanding process is scheduled early on, will delay all processes behind it (even less demanding ones)
        - May be suitable in cases where:
            - Response time is not important, want to maximize throughput (e.g. batch)
            - Minimizing overhead more important than single job's completion time (e.g. running on expensive HW)
            - Systems where processes are well-understood, computation times are known to be low
                - Ex: embedded systems (computers within devices, meant to do certain things)
- Shortest job first (SJF) - run the shortest job first, then the next shortest, etc.
    - Also simple, better average turnaround time than FIFO

        - Optimal if all jobs arrive at the same time, but relies on knowing lengths of all jobs
        - Turnaround time =/= response time; SJF is bad for response time (need to wait for all preceding jobs to completely finish)
    - Issue: if jobs don't all arrive at same time, have same convoy effect
- Real-time schedulers - for certain systems, some things must happen at particular times (e.g. industrial control systems - things not happening -> wasted product)
    - Deadlines based on real-time clock; may be hard or soft
    - Hard real-time schedulers: system absolutely must meet deadlines
        - Unmet deadline -> system failure (ex: power plant)
        - To ensure no missed deadlines, need very, very careful analysis
            - Need to make sure no possible schedule causes missed deadline, by analyzing ahead of time
            - Scheduler needs to rigorously enforce deadlines
        - Ensuring hard deadlines
            - Must have deep understanding of code in all processes - know exactly how long each process takes
                - Vital to avoid non-deterministic timings, even if it results in speedup sometimes, so long as they can potentially result in slowdowns
            - Typically requires turning off interrupts, since unexpected OS intervention can cause issues
                - Non-preemptive schedulers: set up predefined schedule, no runtime decisions
    - Soft real-time schedulers: highly desirable to meet deadlines, but some (or any) can occasionally be missed
        - Goal: avoid missing deadlines if possible
            - Ex: may use for playing video, e.g. (miss deadline -> buffering, frame skip, etc.)
        - May have different classes of deadlines - some harder than others
            - Soft requires less upfront analysis than hard deadlines
        - Soft real-time systems: less vital to meet every deadline

                - Non-preemptive -> may not be able to guarantee meeting every deadline
                    - If missed deadline, behavior depends on particular type of system; will generally be well-defined in each particular system
                    - Potential behaviors:
                        - Drop job whose deadline was missed
                        - Allow system to fall behind to complete the current one
                        - Drop some other job in the future to complete the current one
            - Soft real time algorithms
                - Earliest deadline first - each job has associated deadline, keep job queue sorted by those deadlines
                    - Prune queue to remove missed deadlines
                    - Goal: minimize total lateness

Preemptive Scheduling

- Preemptive scheduling - a process is chosen to run, then runs until either it terminates/yields, or the OS decides to interrupt it & run another process
    - Interrupted process/thread restarted later, typically
    - Implications of forcing preemption:
        - Processes may be forced to yield at any time: if more important process becomes ready (due to I/O completion interrupt, e.g.) or running process' important is lowered (due to having run too long, e.g.)
        - Interrupted process might not be in "clean" state - makes saving, restoring state more complicated
        - Allows scheduler to enforce "fair share" scheduling
        - Introduces context switches; are gratuitous (not required by the dynamics of processes), introduces additional overhead
        - Creates potential resource sharing problems
    - Implementing preemption

            - Need a way to get control away frmo process
                - Ex: if process makes sys call, or clock interrupt
            - Want to consult scheduler before returning to process
                - Any changes in priority of current or enqueued processes, e.g.
            - Scheduler finds highest priority-ready process; may return to current or yield current in favor of higher-priority process
    - Clock interrupts - way for modern OSes to stop a process
        - Modern processors contain clock (hardware device - counter for how much time has passed since last tick) - peripheral device with limited powers
            - OS can set up clock to generate an interrupt at a fixed time interval, temporarily halting any running process
            - Interrupt switches code to OS interrupt handler, allowing OS to re-evaluate scheduling
        - Important for preemptive scheduling (key technology)
            - Good way to ensure a runaway process (e.g. infinite loop) won't run infinitely
- Round robin - given N processes, give each process 1/N of total processor cycles
    - Goal: fair share scheduling
        - Every job experiences similar queue delays, receives equal shares of CPU
        - Good for highly interactive, but lightweight processes
    - Time-slicing: runs each process for some nominal time slice, runs until block or time slice expiry, then re-queues at end of queue
    - All processes get frequent chances at some computation, but no process will finish particularly quickly
        - Is fair - evenly divides CPU among active processes on a small time scale
            - Fair usually conflicts with turnaround time
    - Requires far more context switches - potentially more expensive
        - Even if only one process remains, clock interrupts will still occur -> context switches still happen
    - Advantage: runaway processes (infinite loop) does relatively little harm, since it receives only 1/N, low response time
    - Round robin and I/O interrupts

        - Process may block for I/O (or syscall) -> halts themselves, may not use whole time slice
            - If occurs, round robin acts same as FIFO
    - RR vs FIFO
        - RR requires more complex switches, and can take more time to complete first process; but lower average waiting time (more responsive)
    - Choosing RR time slices
        - Performance of preemptive scheduler depends on how long time slices are
            - Long time slices avoid context switches, but run processes for longer before interrupting -> lower response time, but higher throughput
    - Want ot balance response time (shorter slices), throughput (longer slices)
    - Costs of a context switch:
        - Need to enter OS - taking interrupt, saving registers, call scheduler
        - Need to run dispatcher to decide who to run next
        - If switched to different process
            - May need to OS context to new process - switch stack, process descriptions
                - Switch process address spaces (stack, page table, etc.): map out old process, map in new processes
            - Lose instruction, data caches (hardware) - greatly slow down first instructions after starting again
                - Most important cost nowadays
- Shortest time-to-completion first (STCF) - similar to shortest job first (SJF), but scheduler can preempt running jobs
    - Optimal for achieving lowest turnaround time, but poor response time (similar to SJF) & requires knowing job lengths
- Priority scheduling algorithms: preferentially run more important processes first
    - Assign each job a priority numer, run according to priority
        - Non-preemptive: priority scheduling is just about ordering processes within the queue
    - Preemptive: when new process is created, may preempt a running process if its priority is higher

        - Problems with priority scheduling:
            - Possible starvation - one process hogs CPU time, prevents all lower-priority processes for running
                - May make sense to adjust priorities: temporarily lower priority of long-running processes, temporarily raise priority for long-waiting processes
- Issue: for SJF/STCF and RR, generally can't predict future of how processes will act
    - I/O - some processes may interrupt early, before time slice is used up
        - Simple approach - treat each individual sub-job (between I/O interrupts) as an independent job; but how to schedule these?
- Multi-level feedback queue (MLFQ) scheduling
    - For different types of tasks, may want to incorporate different kinds of scheduling
        - Ex: tasks without interaction (mainly CPU-bound): want to run for a long time, maximizing throughput
            - Long quantum/background
            - Long, but low priority - minimize overhead
        - Ex: tasks with interactive behavior: do lots of I/O interrupts, thus don't need long time slices (will interrupt anyway)
            - Short quantum/foreground
            - Short, but high priority - minimize response time
    - May have multiple scheduling queues: each one has different-length time slices
        - Place each process in appropriate queue, depending on its needs
            - Low priority queues (longer-running jobs) may have longer time slices than high-priority queues (interactive jobs)
        - Perform round robin within each queue
        - Balancing different queues: various methods
            - Ex: each queue gets its own percentage of overall processor time, or simply highest priority jobs run first
    - Which queue to put a new process into?
        - If put in wrong queue, scheduling discipline can cause problems
        - Can start all processes in short quantum (high priority queue), give initial standard allocation (**allotment**) of CPU
            - Reduce allocation every time it runs

            - Once all its allocation has been used up, move to longer quantum (lower priority queue)
        - Periodically move all processes to higher [highest] priority queue in order to avoid starvation
            - Ensures jobs in lower priority queues get to run
        - (Alternative) reduce priority if job uses up entire allotment (doesn't exit early); reset allotment every time it is scheduled
            - Susceptible to quitting at 99% usage to game system
    - Benefits
        - Providdes acceptable response time fo rintercative jobs, other jobs with regular external inputs
            - Spend less CPU time, spend more time waiting for I/O -> stay in high priority queue for longer
        - Provides efficient, but fair use for non-interactive jobs
            - Run for long time slice without interruption; if starved, are eventually boosted in priority
        - Provides dynamic and automatic scheduling adjustment based on past behavior of jobs
            - Places each job in appropriate queue
    - Various parameters/options involved in MLFQ scheduling
        - Ex: number of queues, time slice sizes for each queue, frequency of priority boosting, allotments
            - Can even determine values, modify behavior based on mathematical formulas (rather than hard-coding)
        - Some systems have syscalls (e.g. advice) dedicated to modifying process priorities
- Real-time scheduling
    - Have unique metrics - timeliness (how closely schedule meet timing requirements), predicability - variation in delivered timeliness
        - Feasibility - whether task requirements are even possible to meet
        - Hard vs soft real-time - whether failing a requirement causes failure or simply reduced performance/recoverable failure
    - Often have various task characteristics for making problems easier

        - Task/job lengths may actually be known ahead of time
        - Starvation of low priority tasks may be okay
        - Workload may be relatively fixed; no bursts of traffic
        - For especially undemanding real-time tasks, can use regular scheduler
    - For simplest systems - may have no schedule (just have each task call the next directly), or fixed/static schedule
        - More complicated systems (varying workload, e.g. due to external events) - require dynamic scheduling

Memory Management

Sec. 5: Memory Allocation

- Memory (esp. RAM) - one of key assets used in computing
    - Have limited amount, but lots of processes in need - how to manage?
        - Simple strategy (give all memory to currently-running process; swap out every time a switch is made) is too slow
            - Memory operations are slow; want to leave processes in memory, simply switch between them
    - Physical memory divided between OS kernel, process private data (+ stack), and shared code segments + libraries
- Memory management goals
    - **Transparency** - each process sees only its address space, is unaware that memory (i.e. RAM) is being shared between processes
    - **Efficiency** - high effective memory utilization, low runtime cost for allocation and reallocation
    - **Protection** and **isolation** - want to prevent private data from being corrupted or seen by other processes
- Physical vs virtual addresses
    - Physical RAM cells have assigned **physical addresses** - locations on memory chip
        - Previously: physical address used to name memory locations
    - For many decades: processes use non-physical **virtual addresses**
        - Virtual addresses mapped to physical addresses, but:
            - One virtual address may be mapped to multiple physical addresses
            - The virtual address may be very different from the associated physical addresses
            - Virtual address =/= physical address, typically
        - Provides more flexibility in memory management, but requires virtual to physical addresses
- Aspects of memory management problem:
    - Most processes can't perfectly predict how much memory they will use
    - Processes expect to find their existing data where they left it (when they need it)

- Entire amount of data required by all processes may exceed amount of available physical memory
- Switching between processes must be fast - can’t afford much delay for copying data
- Cost of memory management itself cannot be too high

The Memory API

Two types of allocated memory in C - **stack/automatic memory** vs **heap memory**

1. **Stack memory**: allocations/deallocations managed implicitly by compiler
    - For local variables or function parameters/return values, e.g.
    - Stored in stack segment of address space
    - Freed automatically when exiting subroutines, terminating program
2. **Heap memory**: allocations/deallocations handled explicitly by programmer
    - Done via library calls, e.g. malloc() [C standard library - not a syscall!]
        - malloc() - takes requested size as argument, returns pointer to newly allocated space (or NULL, if failing)
            - In C: can specify sizeof(double), is replaced by compiler with correct argument at compile time (compile-time operator)
            - Can also pass a variable to sizeof()
            - malloc()-returned pointer has no type; programmer may choose to cast it (within C) to an int pointer, e.g.
        - free() - taking pointer from malloc() as argument, deallocates memory
            - Size of allocated region handled by memory-allocation library
    - Freed via free() library call or terminating program

Common errors with heap memory:

1. May forget to allocate memory -> program will attempt to access memory illegally, causes segmentation fault
2. May not allocate enough memory (e.g. buffer overflow) - may either seg fault, do nothing, or write into unintended portions of memory
    a. Can cause security vulnerabilities
3. May forget to initialize values for allocated memory -> will attempt uninitialized read, will return whatever value was previously in the heap
4. May forget to free memory (memory leak) -> can cause system to run out of memory in extreme cases, forcing a restart
    a. May never be garbage collected if a reference still exists
    b. Especially critical within OS/kernel code

5. May attempt to reference freed memory (dangling pointer) -> can either crash program (if memory was not later allocated), or will access memory intended for something else (if space was reallocated for something else)
6. May attempt to free freed memory (double free) -> undefined behavior
7. May pass in a bad value to free -> invalid free, can result in negative consequences

Heap memory in modern languages

- Some languages have garbage collectors - will deallocate memory that is no longer accessible (all references to it have been deleted)
- Some new languages have automatic memory management

Note: malloc(), free() are library calls, not syscalls; instead, they are built on syscalls

- Other library calls:
  - calloc() - allocates memory, zeroes it before returning
  - realloc() - given a region of memory, creates a larger region and copies old region into it -> returns pointer [for expanding an array, e.g.]
- Syscalls difficult to use directly; recommended to interface through memory-allocation libraries (e.g. cstdlib)
  - Ex: brk() syscall - changes location of program's break (location of end of heap)
    - Alt: sbrk() - increments location
    - Takes address of new break as argument, modifies heap accordingly (either increasing or decreasing size)
  - mmap() syscall - obtain memory from OS
    - Creates anonymous memory region within program - associated with swap space (not any particular file)
      - Can also treat and manage as a heap

Memory Allocation Strategies

Fixed Allocation

Fixed Partition Allocation: pre-allocate partitions for n processes

- One or more partitions per process, usable only by owning process
    - Reserve space for largest possible process
    - Partitions come in one or a few set sizes
- Very easy to implement - low overhead, cheap and easy allocation/deallocation
    - Common in old batch processing systems
    - Works for both physical, virtual memory
- Good for cases where all process memory needs are known

Memory protection & fixed partitions

- Need to enforce partition boundaries, prevent processes from accessing each other's memory
    - Typically, use hardware support - have special registers containing partition boundaries, only accept addresses within register values

Problems:

- assumes memory usage is known ahead of time, doesn't grow (inflexible)
- Limits number of processes supported - can only run number that fit within available RAM
- Not great for sharing memory
- **Fragmentation** causes inefficient memory use
    - High degree of internal fragmentation
- Not used in most modern systems, may be considered for special systems (e.g. embedded) where memory needs are exactly known

Dynamic Allocation

Dynamic partition allocation - like fixed partition, but partitions are variable size (usually almost any size requested)

- Each partition has contiguous addresses, processes have access permissions for partitions (similar to fixed size)
    - Each process could have multiple partitions (e.g. code area, heap area, stack area; different permissions on each partition, maybe)
    - Partitions potentially shared between processes
- Problems (physical memory dynamic allocation)
    - Not easily relocatable - once process has a partition, can't easily move contents elsewhere
        - Partitions tied to particular address ranges during execution
            - If contents of partition moved to another set of addresses, all pointers will be wrong - generally don't know which memory locations contain pointers
    - Partitions are not easily expandable
        - May not have any space "nearby" (to expand into available space) if adjacent spaces are taken up by other partitions
    - Can only run as many applications as fit in total available memory
        - Cannot support applications with larger address spaces than physical memory
    - Subject to external fragmentation
    - Maintaining free list more expensive than fixed-size
- Keeping track of variable sized partitions
    - Start with one large heap of memory, allocate it as processes request
    - Maintain **free list** - system data structure to keep track of pieces of unallocated memory (not necessarily a list)
    - When proces requests more memory:
        - Find large enough chunk of memory, carve off piece of requested size, put remainder back on free list
    - When process frees memory, return freed memory to free list
- Managing the free list
    - Easy with fixed-size blocks - bit map indicating which blocks are free
        - Pros: small, easy to find free space, efficient to allocate/free
        - 0 or 1 for every block, depending on status
        - Alternatively: list of fixed-size units

    - Variable chunks require more information: start address & size of each chunk
        - Implement as linked list of descriptors, one per chunk of memory
            - Descriptors list size of chunk, whether it is free
            - Each has pointer to next chunk of memory on list
            - Descriptors kept at front of each chunk, typically
                - Descriptor, then remainder of memory (allocated or unallocated)
        - May or may not have descriptors for allocated memory
    - **Free chunk carving/splitting** - find large enough chunk, reduce to requested size, create new header for residual chunk, insert residual chunk into free list, mark carved piece as in use
- Variable partitions less subject to internal fragmentation
    - Exception: unless requester asked for more than it will use (not uncommon)

Algorithm choices: best/worst/first/next fit

- Best fit - search for "best fit" chunk, smallest size greater than/equal to requested size
    - Advantage: might find perfect fit
    - Cons: have to search entire list every time, tends to create lots of small fragments
- Worst fit - search for "worst fit" chunk, largest size greater than/equal to request size
    - Look for biggest free partition
    - Pro: tends to create large fragments, but only for a while
        - Small fragments will still occur over time, as free chunks shrink
    - Con: still have to search entire list
- First fit - find the first chunk that is big enough
    - Pros: very short searches, random-sized fragments
    - Cons: first few chunks quickly fragment
        - Searches become longer, ultimately fragments as badly as best fit
- Next fit - similar to first fit, but set a "guess pointer" to start search from
    - Set guess pointer to chunk after last chunk used
        - Combination of first fit, worst fit; like first fit, but start from place-last-used rather than restarting from head every time
    - Pros: shorter searches (like first fit), spreads out fragmentation (like worst fit)
    - Con: fragmentation still occurs in the long run

    - Guess pointers - general technique for repeatedly searching within lists

- Tracking allocated region sizes
    - Most allocators store size information in in-memory header block, typically right before allocated region itself
        - Size of region: size of header + size of user-allocated space
- Allocating space for the free list
    - Need to build list inside a list
- Growing the heap
    - Traditionally: allocators start small, request memory as needed (via syscall, e.g.)
    - If out of space, may simply fail and return null
- Other approaches to memory allocation
    - Segregated lists - if a few sizes are popularly used by an application for requests, can keep a separate list just for objects of that size
        - All other requests go to more general memory allocator
        - For that particular size - less fragmentation, can service allocate/free requests very quickly (no list search needed)
        - Issue: How much memory goes to the specialized pool?
            - Slab allocator - when kernel boots up, allocates some object caches for likely-to-be-requested kernel objects (e.g. locks, I/O)
                - When a cache runs low, requests some slabs of memory from more general allocator
                - When a slab is empty, can be reclaimed by general allocator
    - [Binary] buddy allocator - can think of free memory as one big space of size 2^N
        - When request is made- search recursively divides free space by two until block big enough found (but cannot divide in half further), then returns the block to requester
            - When block is freed - can check whether the other “half” [buddy] is free and coalesce quickly; can be done recursively up the tree until an in-use “buddy” is encountered
        - Con: suffers from internal fragmentation
    - Many approaches suffer from scaling - llist searches are slow
        - Advanced allocators - complex data structures to address scaling costs

Buffer Pools

- Special case for fixed allocations
    - Internal fragmentation results from mismatches between chunk sizes and request sizes; but in real life, many memory request sizes may be one of a few “standard” sizes (e.g. 64, 128, 4K) requested more often than others
        - Reasons: many key services (e.g. file systems, network protocols, standard request descriptors) use fixed-size buffers
            - Account for much transient use, continuously allocated & freed - might be handled by OS specially
    - **Buffer pools**: In case of poplar sizes, reserve special pools of fixed-size buffers & satisfy matching requests using those pools
        - Only satisfy perfectly matching requests -> no internal fragmentation within buffer pool
        - More efficient - much simpler than variable partition allocation (removes searching/carving/coalescing), reduces external fragmentation
            - Bit map rather than free list for storing chunks
        - Problem: must know how much to reserve
            - Don’t want to reserve too much (causes unused space), want to reserve enough (otherwise, buffer pool doesn’t help much)
        - Using buffer pools
            - Process requests piece of memory (size may be specified explicitly or implicitly - determined by OS), system supplies element frmo buffer pool
            - Process uses it, completes, frees memory (may be freed explicitly, or implicitly - ex: message send triggers buffer free once message sent)
        - How big should buffer pool be?
            - Can resize buffer pool automatically and dynamically - helps adapt to changing loads

        - If run low on fixed-sized buffers, get more memory from free list and carve up more fixed sized buffers
        - If free buffer list gets too large, return some to free list
        - If free list gets low, ask major services with buffer pool to return space
    - Can tune various parameters: low/high space thresholds, nominal allocation
        - OS will periodically check in on buffer pool, determine if anything needs to be done

Memory Allocation Problems

Memory Leaks

Memory leaks occur when a process allocates space, but doesn't free it

- Long-running processes with memory leaks can waste large amounts of memory
- Can happen when a process manages its own memory space (i.e. allocates a large area, maintains its own free list)
    - Can also occur in buffer pools if a process is done with a buffer, but doesn't free it

One solution: **garbage collection** - don't trust processes to release their own memory

- Monitor amount of free system memory; when running low, search data space to find every object pointer
    - Note address/size of all accessible objects, compute the complement (i.e. all inaccessible objects)
    - Can then add all inaccessible memory to the free list
- Finding all accessible memory
    - Object-oriented languages can help - all object references tagged, descriptors include size information
    - Often possible for system resources where all possible references known (e.g. can easily determine who has which files open)
    - General garbage collection - problems
        - Locations in data/stack segments might look like addresses, but have to consider:

            - Are they pointers, or other data that simply look like memory addresses
            - If pointers, are they themselves still accessible
                - Might be able to infer (recursively) for pointers in dynamically-allocated structures, but what about statically allocated (potentially global) aeras
            - What is the size of the "pointed" region
            - Circular references - need to determine
- Can run garbage collection as ongoing background process

Fragmentation

Fragmentation occurs when memory is not being used effectively, some is not being used/can't be allocated & used (for various reasons)

- Is a problem for all memory management systems
    - May prevent system from running as many processes as it should be able to with its memory
- Results from inefficiencies in memory allocation
    - Ex (fixed partitions): if partition sizes are larger than process memory requirement, the remainder unused space is wasted
        - Requester given more than needed, unused part is wasted

Two forms of fragmentation - ***internal*** and ***external***

1. **Internal fragmentation**: Unusable pieces of memory within allocated partitions
    - Can always occur when allocating in fixed-size chunks
        - Typically caused by mismatches between the chosen size of a fixed-size block [larger] vs the actual size used by processes [smaller]]
        - Ex: fixed partition allocation, buddy allocation, sector-based memory management on external drives, any form of paging
    - Average waste: 50% of each block
        - No real general solution for fixed allocation
2. **External fragmentation**: pieces of memory between allocated partitions (but not inside) that are too small to be practically used by any program

    - Each allocation creates leftover free chunks become smaller and smaller over time
        - Occurs as memory is allocated & released
        - Leftover free chunks too small to satisfy requests -> become form of fragmentation waste

Solutions for external fragmentation:

- Be smart about which free chunk of memory is used when satisfying a request
    - Con: Being smart costs time
- **Coalescing** - recombine fragments into big chunks
    - All variable sized partition algorithms have external fragmentation, want a way to reassemble fragments
    - Check neighbors whenever a chunk freed - recombine when possible
        - Can design free list to make check more efficient: order list by chunk address to keep neighbors close
    - Fights external fragmentation

- Fragmentation and coalescing - opposing forces
    - Coalescing works better with more free space (higher chance of being able to coalesce)
    - How fast is allocated memory turned over? Held chunks cannot be coalesced
    - How variable are requested chunk sizes
        - Higher variability increases fragmentation rate
    - How long will program execute
        - Fragmentation worsens over time

- **Compaction and relocation** - garbage collection frees memory, but doesn't significantly affect fragmentation
    - Ongoing activity can starve coalescing - chunks might be reallocated before neighbors become free, e.g.
        - Don't want to stop accepting new allocations, otherwise processes that need additional memory would block until memory is freed
    - Want a way to rearrange active memory - repack all processes in one end of memory, create large chunk of free space on other end (**compaction**)

            - OS may choose to stop process and rearrange
            - Issue: takes processor time, compaction may make growing existing segments more difficult
        - Use swap device - other (not RAM) device, e.g. flash drive/hard drive
            - Copy all partitions onto swap device, then re-add to RAM but as adjacent partitions
        - Requires **relocation** - moving process' data from one location in memory, to new and different location
            - Challenges: all addresses in program will be wrong
                - Ex: references in code segment, calls and branches to other parts of code, references to variables in data segment, new pointers created during execution [pointing into data/stack segments]
                    - If reference moves out of program address space, may attempt to reference outside of program's own memory & cause fault
            - Not generally feasible to relocate process - similar challenges to garbage collection
                - Solution: can we make processes location independent?

- **Defragmentation** - changes which resources are allocated
    - In many cases - storing objects in contiguous memory makes operations (e.g. reads/writes) much faster
        - Issue: after many operations, files may become discontiguous, causing performance drops
    - To defragment - choose an area where we want to write contiguous files
        - For each file in that area, copy elsewhere to free space
        - Once done, can coalesce entire free space (erase existing data within) into one big chunk, then choose a set of files to copy into now-contiguous free space
            - Can repeat until all files, free space contiguous
    - Defragmentation is an ongoing process, similar to garbage collection

Sec. 6: Virtual Memory

Virtual Memory

Principle: Maintain separate **virtual address space** (only part seen by process), separate from **physical address space**

- Have "magical" **address translation unit** (in hardware) mapping virtual addresses to physical addresses
- Processes only ever issue, deal with virtual addresses; never physical

Segmentation - might store each memory segment (e.g. stack, data) into its own place on physical memory & use translation unit to match calls/references

- Uses segment as unit of relocation; natural model, long tradition in computing
- Use hardware **segment base registers** - point to start (in physical memory) of each segment
    - Set by operating system
    - Ex: stack, data, code base registers
    - CPU automatically adds base register to every address
        - Segment may grow upward or downward from base register
- OS uses segment base registers for virtual address translation
    - Set base register to start of region where program is loaded
    - If program is moved, reset base registers to new location
    - Result: program works no matter where segments are loaded
        - Segments no longer need to be adjacent in memory, may be placed independently
            - Some systems: put code, heap in same segment to have only two segments: code/heap and stack
                - Need only one bit to choose which segment
- Address translation unit
    - Computes: physical [addr] = virtual + base (in segment base register)
    - Note: offset may be either positive or negative (i.e. for the stack)
- Virtual addresses
    - Explicit - 1 (or 2) bits encoding which segment + offset

        - Issue: limits maximum size of a segment
    - Implicit - which segment, is determined from source (from PC -> code, from stack pointer -> stack, any other address -> heap)
- Code sharing
    - To support code sharing, add a few protection bits per segment - indicate whether program can read/write/execute a segment
        - Can set code to read-only
        - Done by hardware support
        - Note: protection bits are segment-wide, cannot apply only to subsets
- Growing segments - memory-allocation library may need to grow a segment
    - Via sbrk() Unix syscall, e.g.; hopefully not rejected, i.e. if no more physical memory is available
- Fine- vs coarse-grained segmentation
    - Coarse-grained - systems with only a few large segments (e.g. code, stack, heap)
    - Fine-grained - large numbers of smaller segments
        - Requires segment table in memory, but more flexible
        - Might chop up code, data into multiple segments (e.g.)

Memory segment relocation- move bits of stack (in physical memory) to new location, update value of segment base register

- Allows for relocation, coalescing segments - can move segments in physical memory without breaking references
    - Can now use variable-sized partitions with less external fragmentation, due to being able to moev partitions around for coalescing
    - Issue: still require contiguous chunks of area for each segment (e.g. contiguous area for stack), some external fragmentation remains
        - Want to get rid of requirement

Still need protection - want to prevent processes from reaching outside allocated memory

- For each segment, store length/limit register
    - Specifies maximum legal offset (from start of offset)
        - Any address greater than this is illegal, CPU reports via segmentation exception/segfault (trap)

    - Alternatively, might hold physical address of end of address space

- Dynamic relocation/base and bounds (summary)
    - CPU stores two hardware registers: base register and bounds/limit register
        - Can relocate address spaces (while running) simply by changing base register (dynamic relocation)
        - Registers require special privileged instructions to modify
- OS responsibilities
    - When new process created: OS searches free list to allocate room for new address space
    - Once process terminates, OS must reclaim its memory for user elsewhere (i.e. put back on free list, clean up any associated data structures as needed)
    - On context switch:
        - OS must save and restore base, limit registers upon context switch
            - May be saved in a per-process structure, e.g. process control block
        - While process is stopped, may choose to performed dynamic relocation
    - OS must provide exception handlers [e.g. trap/interrupt]
        - Installed at boot time as part of OS kernel via privileged instructions

Memory Management Techniques

Remaining Issues:

1. Segments still need contiguous chunks of memory
2. Still can't support more process data needs than we have physical memory

Swapping

Swapping - what if we don't have enough RAM to handle all (or even one) process' memory needs

- Can keep memory on some othe rlocation, e.g. disk (hard/flash)
    - Issue: can't directly use code/data on a disk
- Swapping to disk - when a process yields/is blocked, copy memory to disk
    - When scheduled, copy it back
        - If have relocation hardware, can put memory in different RAM locations (i.e. don't need to copy back to same place)
    - Consequently: in principle, each process could see memory space as big as total amount of RAM
    - Con: actually moving everything out (copying is expensive)
        - Context switching becomes very expensive
            - Need to: copy all of RAM out to disk, copy other stuff from disk to RAM before newly scheduled process can do anything
        - Process still limited to amount of total RAM
- Swapping - old strategy

Paging

Paging - previously segments required to be contiguous; causes external fragmentation

- Paging approach - divide physical memory/RAM ito units of single fixed size
    - Pretty small units, e.g. 1-4K bytes/words - called **page frames**
- Do same process for virtual address space - call each virtual unit a **page**
    - Pages same size as page frame
- For each virtual address space page, store its data in one physical address page frame
    - Store in any page frame, no specific page frame required
    - Can divide each virtual segment into pages, store each page into some page frames (need not be contiguous)
        - Segment implemented as set of virtual pages
            - Although pages are fixed-size, internal fragmentation averages only to half a page (half of last page)
            - All preceding pages are filled
    - No external fragmentation - never carve up pages
    - Reduces fragmentation costs significantly
- Needs per-page translation mechanism to convert virtual to physical pages
    - Need to be able to translate every memory reference (virtual address) to physical
        - Needs to be fast -> use hardware (MMU)
        - Page-level translation, not just segment-level (registers from before)

The Memory Management Unit

- **Memory management unit** (MMU) - hardware designed to perform page translation quickly
    - MMU stores **page table**: table mapping translations of virtual addresses to physical addresses
        - For each entry, also have **valid bit** - determines whether virtual page number is legal
            - If virtual page number maps to invalid bit entry, invalid
        - Also contains address translation unit, base and limit registers

    - Virtual address stores two parts: lower bits contain offset within a page, higher bits contain page number (encoding which page)
        - Ex (4K): lower 12 bits are offset, 52 are page #
    - Process issues virtual address
        - Hardware will use top-level bits to index page table (check valid bit), selected entry contains *physical* page number
        - Physical page number becomes higher-order bits in physical address; lower-order bits (offset) same as in virtual address

The MMU in hardware:

- Used to sit between CPU/bus, nowadays integrated into CPU
    - Between chips is slower than within a chip, but integrating takes valuable real estate on chip
- Page tables
    - Initially, implemented in special fast registers
    - Nowadays: too many registers would be needed to handle modern memory size, esp. since CPU is space-limited
- Handling big page tables - store in normal memory (RAM)
    - Issue: don't want to have to perform two bus cycles (one for page table lookup, another to get actual data) for every memory access
    - Solution: use very fast set of MMU registers as cache within MMU, store page table entries
        - Hope process-issued address matches entry within cache
            - Results in worries about hit ratios, cache invalidation, other issues (no free lunch)

Demand Paging

Modern systems - typically, several processes running + many waiting to be scheduled

- Each process needs own set of pages
- Pages can go anywhere, but total page need may be more than total number of page frames
    - Can't keep all page frames in RAM
    - Extra pages - store on flash drive

        - Paged systems generally keep some pages on persistent memory device
    - Issue: code can only access pages in RAM
        - For every running process, keep some pages in RAM & some on disk
            - What if process asks for a page that is on disk?
                - Try to store less-used pages on disk, more likely to be used on RAM

Demand paging - process doesn't actually need all of its pages, only those that it references

- No need to load unused pages
    - Similarly, some initially used pages (for startup, e.g.) may not be used afterward
- May also not get rid of all process' pages when it yields
    - Otherwise, once rescheduled, would need to reload all pages
- Want to make prediction of which pages, for each process, should be kept in memory

- Ongoing MMU operations
    - If current process adds/removes pages, directly update page table in memory
        - Also need to update cache - handled by MMU via privileged instruction
    - Switching processes
        - Maintain separate page tables for each process
        - Privileged instruction loads pointer to new page table, reload instruction flushes previously cached entries
    - Sharing pages between multiple process - make each page table point to same physical page, can be read-only or read/write shared

Implementing demand paging

- Page table must store, for each process, which of its tables are in a page frame and which aren't (i.e. which might be in external storage)
    - MMU must support "not present" pages - upon referencing not present page, generate page fault/trap, ask OS to bring in requested page and retry faulted reference
- Entire process doesn't need to be in memory to start running
    - Start with only subset of pages, load additional pages as demanded/requested by program

Demand paging performance

- Demand paging performs poorly if most memory references require disk access; potentially worse than simply swapping in all pages at once, e.g.
    - Want to make sure thi sisn't the case; want to ensure that page holding next memory reference is already there almost always
- Speed of address translation
- Predicting pages needed in memory
    - Can rely on **locality of reference** - next address requested is likely to be close to last address requested
        - Is true for most programs
            - Ex: code usually executes sequences of consecutive/nearby instructions
                - Most branches tend to be relatively short distances, within same routine
            - Typically need access to things in current/previous stack frame
                - Usually only work within current stack frame (function), stack frames generally only a few pages long
                - Only work in other stack frame when adding/removing stack frame
            - Many heap references to recently allocated structures (e.g. creting/processing a message)
                - Ex: working within a big data structures show locality of reference
            - All three types of memory likely to show locality of reference

Page faults - with demand paging, page tables no longer necessarily contain pointers to RAM page frames (if page is not in RAM, is on disk)

- When program requests address from page on disk
- Page table entries initialized to " not present" unless loaded into page frame
- CPU faults if "not present" page reference
    - Fault enters kernel, like any other exception; forwarded to page fault handler

        - Handler determines which page is required + where it is, schedules I/O (to flash drive) to fetch page, then blocks process
            - Run othe rproecsses will blocked
        - Once page fetched, make page table entry point to newly read-in page, back up user-mode PC to retry failed instruction, return to user mode & run
- Page faults don’t impact correctness, only slow process down
    - Page fault largely invisible to process, except in performance sense; never crash due to page faults
    - Issue: page faults block process, lots of page faults will result in slow performance
        - Overhead (fault handling, paging in/out) associated with every page fault

Virtual memory - generalization of demand paging

- Provides processes illusion of having RAM, allow them to issue any address they want [within their address space]
- General: give each process address space of immense size
    - May be as big as hardware’s word size
    - Allow processes to request segments within space
        - May request code, data, heap, stack, etc.; can request as large as allowed
        - Any address within segment will be a legal address, no matter whether associated page frame exists
            - Use demand paging to get pages in/out
            - May use swapping to bring entire processes in and out of disk
- Protection fault - required translation in TLB/main memory, but protection bits don’t allow access
- TLB hit - translation TLB
- Page fault - not in TLB,, is valid, not present entry in page table
- Segfault - not in TLB, not listed as valid in page table

Page Replacement

- Issue: OS has no control over what pages read in by process, can only try and predict; only controls what is currently in memory
    - **Replacement algorithm** - change which page is loaded into a given page frame, move contents to disk & move other page from disk to page frame
- Page replacement
    - Basics - keep some set of all pages in memory, generally as many as fit (but not all for the same process - each process gets some amount of page frames)
        - Some will be for OS as well
    - Want to figure out which page frames aren't being used, so they can be offloaded/ejected to disk (in cases where we need to replace one)
    - Optimal replacement algorithm (Belady) - replace the page that will be next referenced furthest in the future
        - Optimal algorithm - delays next page fault as long as possible, fewest page faults per unit time -> lowest overhead
        - Issue: need to figure out how to predict which page this is (don't have an oracle - perfect predictor)
            - Incorrect prediction causes extra page fault - results in performance penalty, but not correctness penalty; generally acceptable tradeoff
                - More correct -> fewer page faults
            - In case of traces (i.e. rerunning programs), can run optimal algorithm
    - Approximating optimal algorithm
        - Rely on **locality of access** (reference?) - note which pages have recently been used
            - For each page table entry, include extra timestamp field - contains time of last access
                - Update field whenever accessing page
            - Replacing page - search for entry with oldest timestamp, choose one to replace

Other candidate replacement algorithms

- Random, FIFO - terrible algorithms

- Least Frequently Used - also not very good
- **Least Recently Used** - as above
    - Naive/”True” LRU version: requires storing timestamps somewhere, search all timestamps every time we eject -> expensive
    - Maintaining information for LRU
        - In the MMU? Issue: MMU does not hold all page table entries for all processes
            - Issue: would need extra hardware/storage to read/write time (performance cost)
            - At most: MMU will maintain read, written bit per page (reference bits)
        - In software? (Page table)
            - Requires every trips to RAM to get/update timestamp, wastes time + requires more space
- Surrogate for LRU - use **clock algorithms**
    - Organize all page frames into circular list
        - Every time page accessed by MMU, set a reference bit
    - Upon page fault: scan circular lsit of frames
        - For each frame, ask MMU if frame’s reference bit is set
            - If so, used recently -> clear reference bit, move on
            - If bit is not set, consider page as approximate LRU and choose this page
                - Update pointer to the page after just-chosen page, then eject chosen page as replacement
    - Generally similar in performance to LRU; both LRU, clock algorithms are approximations of optimal
        - Analogous to next-fit w/ guess pointer (previously seen)

Page replacement & multiprogramming

- Often, may have a lot of processes running at once (multiprogramming)
    - Typically, once a process is interrupted, would expect pages related to that process to be ejected before it gets scheduled again
        - Once it restarts, will pagefault

- Alternative: may decide not to clear out all page frames on each context switch; want to somehow share page frames across all page frames that want to run

Possible choices:

1. One single global pool of pages across all processes
    - Treat entire set of page frames as shared resource, approximate LRU for entire set, replace whichever process' page is LRU
    - Not great:
        - Doesn't work in round robin for same reason as said above, e.g.
2. Fixed allocation of page frames per process
    - Set aside some number of page frames for each running process, use LRU approx. separately for each
    - Q: how many page frames per process? (fixed number is bad)
        - Different processes exhibit different locality, and which pages + number of pages changes over time
3. Working set-based page frame allocations
    - Give each running process an allocation of page frames matched to its needs
    - **Working set** - set of pages used by process in a fixed-length sampling window in the immediate past
        - Allocate enough page frames to hold each process' working set
        - Each process runs replacement within its own set
    - Typically, observe that number of page faults initially decreases rapidly as working set size increases; but as working set size grows, number of page faults does not decrease by as much (little marginal benefit) - 1/x
        - Want to find the sweet spot

Working sets

- Optimal working set for a process - number of pages needed during next time slice
    - Fewer frames -> needed pages will keep replacing one another, process will run slowly
    - Use past process behavior to determine what working set size is
        - Which pages are contained in working set, determined by process via page faults

- Managing working set size - want to somehow manage working set size
    - Working set size may vary over time, but page frames are a relatively scarce resource
- Typically, observe paging behavior of a process (i.e. faults per unit time); want to adjust number of assigned page frames accordingly
    - Frequent page fault -> need more page frames
    - If no free page frames, **page stealing** - take page frames from another process

Page stealing

- Track last use time for each page, want to find page (approximately) least recently used by its owner
    - Similar to page replacement, use working set-clock algorithm to approximate LRU
    - Steal said page
        - Goal: processes that need more pages tend to get more, processes that don't use their pages tend to lose them

Working set size characterizes each process; but what if sum of working set sizes is greater than number of available page frames?

- Consequently: no one will have enough pages in memory, will keep stealing from another program -> many, many page faults (**thrashing**)
    - Causes all systems to run slow; continues until system takes action
- Preventing thrashing - typically, cannot add more memory or reduce working set sizes (since this also causes thrashing - all processes have less memory)
    - Can reduce number of competing processes - swap out some of the ready processes to ensure enough memory for remainder to run
        - Can kill processes and reclaim all of its page frames, but more commonly - take a process, copy all page frames to disk, then put process in special "swapped-out" state
        - Swapped-out process won't run for quite a while
            - Issue: OS generally doesn't know which processes are user-important
            - To solve, can round-robin which processes are swapped in/out; expensive, but better than thrashing

Clean vs dirty pages - when a page has been paged in from disk, have two copies of that page - one on disk, one in memory

- If in-memory copy has not been modified, then an identical valid copy exists on disk - in-memory copy is “clean”, can replace clean page without writing back to disk
    - Very easy to replace clean pages
- If in-memory copy has been modified, is “dirty” and needs to be re-written to disk if it is paged out
    - On-disk copy is no longer up-to-date
- Clean vs dirty pages
    - Clean pages can be replaced at any time; relatively cheap to do (don’t need to write to disk)
        - Issue: can’t only kick out clean pages (would limit flexibility)
- Solution: pre-emptive page laundering - can convert dirty pages to clean ones in the background to increase flexibility
    - Can run ongoing background write-out of dirty pages: find and write out all dirty, non-running pages to make them clean again
        - Assumes need to eventually page out

Threads and Synchronization

Sec. 7: Threads and IPC

Threads

Threads - form of interpreter (like processes)

- Why not processes?
    - Processes are expensive to create (they own resources) and dispatch (they have address spaces, need to context switch, etc.)
    - Different processes are very distinct - cannot share same address space, do not typically share resources
    - Not all programs require strong separation; mutually trusting elements of a system may work cooperatively for a single goal
        - Ex: web servers might handle many distinct requests independently, but the handlers for each may share code, resources, etc.
        - Multithreading - data mining, data analysis often involve lots of multithreaded capability

Thread - strictly, a unit of execution/scheduling

- Given a set of instructions, executes them
    - Each thread has its own stack, PC, registers
    - Other resources (e.g. data area, code) can be shared with other threads
        - All threads can share code, share same data area, etc.
- Typically, processes contain threads
    - Multiple threads can run in a single process
        - All share same code, data area; same resources (e.g. open files) -> cheaper to create and run
    - Can also share CPU between multiple threads in a single process
        - Modern computers - can have multiple threads running on multiple cores simultaneously

Process vs threads - given a set of tasks/submodules:

- When to have multiple processes?
    - When running multiple distinct programs
    - When creation/destruction are rare events
    - When running agents with distinct privileges
    - When there are limited interactions, resource sharing
    - To prevent interference between executing interpreters
    - To firewall one from any failures of the other
        - Segfaults on a thread will probably crash the entire process
- When to have multiple threads?
    - For parallel activities in a single program
        - Ex: data mining, web servers
    - When there is frequent creation/destruction
        - Cheaper to create/destroy threads than processes
    - When all can run with the same privileges
    - When they need to share resources
    - When they need to exchange many messages/signals
    - When you don't need to protect them from each other

Thread state - each thread has its own registers, PS, PC, stack area

- Maximum stack size specified when thread is created
    - A process can contain many threads, but cannot all grow towards same hole; consequently, thread creator must know max required stack size
        - Need defined behavior for when a thread exceeds its max stack size
    - Stack space must be reclaimed when thread exits
- Procedure linkage conventions, subroutine call behavior unchanged

Two types of threads - **user level threads** vs **kernel threads**

1. **Kernel threads**: abstraction provided by kernel
    - Are created by user syscalls and managed by the OS directly
    - Still share one address space, but scheduled by the OS
        - Can have multiple threads running simultaneously
    - Modern Linux provides kernel threads
2. **User-level threads** - abstraction by user-level libraries

        - Kernel doesn't know anything about user-level threads
            - Sees only the process, schedules at process level; doesn't see the threads at all
            - Cannot run on multiple cores
        - Provided, managed via user-level library
            - Scheduled by library, not by kernel
                - Handling threads (e.g. blocking, switching) handled by library

Generally: kernel level threads preferred

- Exception: no advantage for kernel-level if threads cannot run simultaneously for program reasons
- pthread - general thread interface library for Linux
    - On modern Linux, uses kernel threads
    - For OSes that don't support kernel threads, would resort to user-level threads

Inter-Process Communication

Sometimes, might want communication between processes (e.g. pipe - interprocess communication)

- Producer-consumer relationships between processes
- Since processes can't normally interact, mechanisms for communication provided by OS (**inter-process communications**/IPC)

Characteristics of IPC mechanisms: simplicity (allow programmers to work with mechanism), convenient, general/versatile, efficient, robust & reliable

- Some are contradictory, provide multiple different IPC mechanisms and allow programmer/user to choose appropriate one

OS support for IPC provided through system calls

- Requires activity from both communicating processes - usually can't "force" another process to perform IPC
- Mediated at each step by OS to protect both processes, ensure correct behavior

OS IPC mechanics for local processes:

- Typically: data in memory/address space of sender, want to transmit to memory space of receiver
    - Might have sender ask OS to copy data into address space of sender
        - Conceptually simple, less user/programmer confusion - both sender, receiver retain their own copy
        - Potentially high overhead
    - Alternatively, OS may use VM techniques to switch ownership of memory [i.e. pages] to the receiver
        - Much cheaper than copying bits, only requires changing entries in a page table
        - Only one of the two processes will see the data at a time
    - Mechanism chosen depending on needs

Synchronous vs asynchronous IPC

1. **Synchronous IPC** - the sender process blocks [is blocked by OS] until the data is received by the receiver process
    - IPC writes block the sender until data is sent/delivered/received
    - IPC reads block the receiver until new data is available
    - Very simple for programmers
2. **Asynchronous IPC** - once sender process has made request to OS, sender process returns to next instruction (even if data has not yet been received)
    - IPC writes return to sender's next instruction once system accepts data
        - Data sending is handled by OS afterward
        - No direct confirmation of transmission, delivery, reception; requires auxiliary mechanism to learn of errors
    - IPC reads return prompty if no data is available
        - Requires auxiliary mechanism to learn of new data
        - Often involves "wait for any of these" operation
    - Much more efficient in some circumstances
        - Good if you want to optimize sender process speed and it is not critical for receiver to have received the message first

Typical IPC operations

1. **Create/destroy IPC channel** from both sending, receiving ends
    - Via system call
2. **Write/send/put** - system calls to insert data into channel
3. **Read/receive/get** - similar, extract data from channel
4. **Channel content query** - how much data is currently in the channel?
5. **Connection establishment and query**
    - Control connection of one end to another
    - Provides information like: who are endpoints, what is the status of connections?

Fundamental IPC dichotomy: **messages** vs **streams**

1. **Streams** - a continuous stream of bytes
    - Read/write a few or many bytes at a time
        - May write/read part at one moment, then further write/read later
    - Write, read buffer sizes unrelated

        - May contain app-specific record delimiters, but not always; specified by application, not IPC mechianism
2. **Messages/datagrams** - a sequence of distinct messages
    - Each message has its own length (subject to limits), is typically read/written as a unit
        - Messages of fixed-size (100 bytes, e.g.); not considered sent until exact size has been entirely transmitted
        - Message size, limits are known by IPC mechanism
    - Delivery of a message is all-or-nothing; no partial messages

Flow control - making sure a fast sender doesn't overwhelm a slow receiver

- If rate of sender creating vs receiver consuming data is highly mismatched (e.g. much more created than consumed), can cause issues
    - Results in large backlog/queue of data not yet seen by receiver
        - Needs to be buffered by OS until requested by OS; consumes system resources
- OS can't allow buffering to proceed indefinitely
    - Can limit required buffer sizes - once buffer is filled, block sender (alternately: prevent sending of additional data, return error to additional transmit calls)
        - Alternatively: flush old data
    - Handled by network protocols or OS mechanism

IPC reliability and robustness

- Within a single machine, OS won't accidentally lose IPC data; however, across a network, messages (requests & responses) may be lost/corrupted on network
    - Since network is out of OS control
- Even on single message, sent message may not be processed
    - May be that receiver is invalid/DNE, dead, or simply not responding/ignoring
        - Q: how long must OS be responsible for IPC data?
- Reliability options (considerations for OS, need to decide which to pick)
    - When to return OK to sender's original syscall?

        - Ex: when queued locally, when added to receiver input queue (i.e. taken out of OS buffer), when read by receiver (taken out of input queue), when receiver explicitly acknowledges
    - How persistently does system attempt delivery?
        - Do we try retransmissions, and how many? Do we try alternate routes or restarts?
        - Esp. across a network
    - Do channel/contents survive receiver restarts?
        - Can new server instance pick up where old instance left off

Types of IPC

Example styles/means of IPC:

1. **Pipelines**
2. **Sockets**
3. **Shared memory**
4. ***Additional types***: mailboxes, named pipes, simple messages, IPC signals

Pipelines - data flows through a series of programs (pipe operator - output of one process is provided as input to next process)

- Data is simple byte stream, buffered in OS; no need for intermediate temporary files
- All programs under single user's control -> no security, privacy, or trust issues
- Error conditions - input: EOF, output: next program failed
- Simple mechanism, but very limiting
    - Named pipe - similar to pipelines, but are presented as files on an OS; can be accessed by separate processes - one reader, one writer

Sockets - consist of connections between addresses/ports in particular processes

- Sender sets up socket, receiver agrees to receive -> communicate across that socket
    - Three main operations: connect, listen, accept
    - Need to figure out who to connect/socket with via lookup - registry, DNS, service discovery protocols

- Very versatile, many data options - reliable or best-effort datagrams, streams, messages, remote procedure calls
- Complex flow control, error handling
    - Retransmissions, timeouts, node failures
    - Possibility of reconnection, fail-over
- Very general, but much more complex
    - Many potential issues of trust, security, privacy, integrity

Shared memory - OS arranges for processes to share read/write memory segments (actual RAM is shared)

- Memory segments mapped into multiple processes' address spaces
    - Actual "communication" is handled entirely by applications
- OS is not involved in data transfer; only facilitates memory reads/writes via limited direct execution
    - Consequently: very fast
- Simple in some ways, very complicated in others
    - Cooperating processes must themselves achieve their desired synchronization, consistency effects
        - Can't generally know when other processes have interacted with shared memory
    - Only works on local machines

Sec. 8: Synchronization

Synchronization

Synchronization - making things happen in the "right" order

- Easy if only one set of things is happening, or if simultaneously occurring things don't affect each other; very complicated otherwise
- Generally not avoidable if parallelism is required (as is usually the case)
    - Benefits of parallelism:
        - Improved throughput - blocking of one activity does not stop others
        - Improved modularity
        - Improved robustness - failure of one thread does not stop others
        - Better fit to emerging paradigms - client server computing, web-based services, etc.

Cooperating parallel programs are hard- if 2 execution streams not synchronized:

- Results may depend on order of instruction execution
- Parallelism makes execution order non-deterministic
- Results may become combinatorially intractable

Parallelism problems

- **Race conditions** - occur when execution order of processes running in parallel can go one way or another
    - Happen often; don't usually matter when non-conflicting
    - In some cases - race conditions may affect correctness
        - Ex: conflicting updates (mutual exclusion)
        - Check/act races (sleep/wakeup problem) - occur if one
        - Multi-object updates (all-or-none transactions)
        - Distributed decisions based on inconsistent views
    - Can be managed if race condition is recognized; want to do so while minimizing loss of parallelism (causes performance penalty)
- **Non-deterministic execution** - parallel execution makes process behavior less predictable

    - Ex: process blocks for I/O or resources, time-slice end preemption, interrupt service routines, unsynchronized execution on another core, queuing delays, time required to perform I/O operations, message transmission/delivery time
    - Can lead to many problems

True parallelism is very complex - not typically feasible to understand full scope of parallelism's effects on program

- More realistically - can identify key points of interaction where problems from parallelism may affect correctness of results
- Solve aforementioned problems via **synchronization** - controlling parallel behavior at those key points of interaction
    - Synchronization as two interdependent problems - **critical section serialization**, **notification of asynchronous completion**

Critical Sections

Critical section problem - a **critical section** is a resource shared by multiple interpreters (e.g. multiple concurrent threads, processes, or CPUs)

- Also critical sections: interrupted code, interrupt handlers, other OS code
- Use of resource may change its state - contents, properties, relation to other resources (e.g. directory location)
- Execution order may sometimes affect correctness of program
    - Ex: when scheduler runs/preempts which threads, relative timing of asynchronous/independent events
    - May occur when more than one thread is executing a critical section at a time (one thread may only have completed part of it when another thread starts it, e.g.)
- Examples:
    - Simultaneous updates to a file by multiple concurrent programs
    - Multithreaded updates to the same data area in memory
        - Even a single [code] instruction can contain a critical section, if it corresponds to multiple assembly instructions
            - OS guarantees no issues within a single assembly instruction, but not between instructions

Interleavings at critical sections are uncommon, but can still happen

- Can prevent bad results by ensuring only one thread can execute a critical section at a time (**mutual exclusion** of critical section)
    - One solution - temporarily block some or all interrupts (e.g. clock interrupt, rescheduling) within the critical section (**interrupt disables**)
        - Done via privileged instruction, as a side effect of loading a new Processor Status Word
        - Prevents time-slice end (i.e. timer interrupts), re-entry of device driver code
        - Issues:
            - May delay important operations - the interrupt may have happened for a reason, and delaying it may be bad
                - Received data may not be processed until the interrupt is serviced
                - Want to ensure period of interrupt disabled is short
            - May never re-enable interrupts, e.g. due to a bug
            - On multi-core machines, won't solve all sync problems
                - Can have parallelism without interrupts on multi-core machines
            - Requires privileged instructions - not an option in user mode, causes risk of malicious behavior
                - Can be done in OS kernel code
            - Can be dangerous if improperly used - might disable preemptive scheduling, disk I/O, etc.
            - May prevent safe concurrency
    - Other solutions
        - Avoid shared data whenever possible
        - Eliminate critical sections with **atomic instructions** (uninterruptible read/modify/write operations)
            - Limited to individual CPU instructions
        - Can use atomic instructions to implement software locks

Mutual Exclusion

Want to achieve mutual exclusion of critical sections to solve correctness issues

- If one thread is running critical section, want to make sure other thread definitely isn't

Critical sections in applications

- Most common for multithreaded applications sharing data structures
    - Can also occur for processes sharing OS resources (e.g. files) or multiple related data structures
- Can be avoided if absolutely no resources are ever shared, but this is not typically achievable

Recognizing critical sections

- Generally involve updates to object state - updates to a single object, or related updates to multiple objects
- Generally involve multi-step operations - cannot be done in a single [assembly language] instruction
    - Individual assembly instructions are atomic, but sets of instructions are not
    - Pre-emption can easily compromise object or operation
- Correct operation requires mutual exclusion across all threads/processes - only one thread can ever have access to object(s)
    - "Access" holds for any kind of permissions

Goal: want to achieve **atomicity** of critical sections

- Atomicity has two aspects: ***Before or After*** and ***All or None***
    - **Before or After** atomicity: ensure that the times during which two threads are within a critical section, have no overlap
        - If A enters before B starts, then B mst only enter after A completes (or vice versa)
    - **All or None**: an update that starts will either complete or be undone
        - Uncompleted update has no effect
- Correctness generally requires having both aspects of atomicity
    - Atomic - completes without interruption

- Transaction - group of distinct operations performed all-or-none

Protecting critical sections

- Turning off interrupts - often not practical
    - Not available for all applications (mainly just OSes), limits all concurrency system-wide
- Avoid sharing data whenever possible
    - Good when feasible, but not always an option; often impossible
        - In general: don't share things that don't need to be shared
        - Sharing read-only data also avoids problems, since no writes occur; however, a single write is enough to cause issues
    - May lead to inefficient resource use even when possible
- Protect critical sections using hardware mutual exclusion (e.g. atomic CPU instructions)
    - Also often impossible - would need to be able to do entire critical section within a single instruction
        - Can be done with very careful design on certain data structures, but usually not
    - Doesn't help with waiting synchronization
- **Software locking** - protect critical sections with **lock** data structure
    - Lock protects critical section - only one party can hold a lock at a time (most important property)
        - Party holding a lock can access critical section; parties not holding cannot access it
    - A party needing to use critical section must acquire lock first
        - If it succeeds, goes ahead
    - Once done with critical section, party should release the lock so someone else can acquire it

Implementing Locking

- ISAs usually do not include instructions for completely building locks
- Need to build locks in software
    - Can use multiple instructions to build a lock

- Building locks
    - Notice: operation of locking/unlocking a lock, is itself a critical section
        - If not protected, two threads may acquire same lock
    - Can solve this initial problem via hardware assistance: individual CPU instructions are atomic, so being able to implement lock with single instruction would solve this problem
        - Have unusual hardware instructions (e.g. Test and Set) that can do this
            - **Test and Set** - given a word, note the current value, set the value to be true, then return value before it was set (atomic exchange)
            - Can use Test and Set, check return value to determine if lock was successfully obtained
            - Also: **Compare and Swap** (CPU instruction, built into silicon)
                - Given a word, old value, new value:
                    - If word is old value, set to new value and return True; otherwise, return False
                - More powerful than Test and Set
        - Can use these atomic instructions to implement a lock
- Lock enforcement - locking only works if either locked resource cannot be used without lock, or if everybody respects the locking rules

What if lock was not obtained?

- Could give up and not perform critical section (not usually correct)
- Can wait until the lock is released (**spin waiting** - keep checking if the lock is available until the answer is yes)
    - Spin waiting for a lock, called **spin locking**
    - Advantages: properly enforces access to critical sections (assuming correctly implemented locks), simple to program
    - Cons:
        - Wasteful (spinning uses processor cycles), particularly in the case of a single CPU
            - Cycles burned may itself delay freeing of desired resource (could have been used by locking party to finish)

            - Less impactful for multiple CPUs; decent if num threads = num CPUs, roughly
        - Bugs by locking party may lead to infinite spin waits
        - No fairness guarantees - a contended lock might cause a thread to spin forever

Other notes

- Coarse-grained locking (one big lock across all critical sections) vs fine-grained locking (different data/data structures have their own locks)
- Additional instructions (MIPS)
    - Load-linked - acts like a load, places memory value into register
    - Store-conditional - only updates if no intervening store to the address has taken place
- Fetch and Add - atomically increments a value while returning old value at that address (single instruction)
    - Can use to build ticket lock - rather than binary flag, associate each thread with a number
        - Lock available if thread number matches value number, unlock by incrementing for the next thread
        - Guarantees all threads will be scheduled eventually

Asynchronous Completion

- Parallel activities move at different speeds; one activity may need to wait for another to complete
    - **Asynchronous completion problem** - how to perform this waiting without killing performance?
        - Ex: waiting for I/O operation to complete, waiting for response to network request, delaying execution for fixed period of real time

One option - **spin waiting** (subject to same pros & cons as above)

- May sometimes make sense:
    - When awaited operation proceeds in true parallel

        - Ex: burning CPU doesn't delay an external message from arriving, or locking party is running on a different core
    - When awaited operation guaranteed to happen soon
        - Spinning less expensive than sleep/wakeup
    - When spinning does not delay awaited operation
        - Note: burning memory bandwidth slows I/O
    - When contention [for lock, e.g.] is expected to be rare
        - Cost of waiting is more significant with multiple waiters

Another option - yield and spin

- After checking some number of times, yield; check again once rescheduled (and repeat process)
- Problems - requires extra context switches, still wastes cycles by spinning every time the process is scheduled (though not as many)
    - Might not be scheduled to check until long after event occurs
    - Works very poorly with multiple waiters - causes potential unfairness (yielding processes less likely to obtain)

Condition Variables

Fairness and mutual exclusion - what if multiple processes/threads/machines need mutually exclusive access to a resource?

- Provided by locking, but can we make guarantees about fairness?
    - Want to avoid starvation, e.g.
- Possible options:
    - FIFO treatment
    - May want to enforce some other scheduling discipline
- A final optin is **completion events**
    - Ex (lock):
        - Once process fails to get a lock, block and ask to be woken up when the lock is available
            - Works for anything else that is waited for (e.g. I/O completion, another process)

Completion events implemented via **condition variables** - synchronization object associated with a resource/request, consist of queue for waiting threads

- Initially set to “event hasn’t happened”; requester blocks, is queued awaiting event
- Once event happens, is “posted” (tell requesters it happened) - posting event to object unblocks 1+ waiters for that event
    - Will check again for event once scheduled, and go again
    - Remain blocked until event happens
    - May reset condition variable to “event hasn’t happened” afterward, e.g.
- Condition variables provided by OS, generally
    - Alternatively, by whatever library code is implementing threading
    - Block a process of thread when a condition variable is used (if it has not yet occurred), moving it out of ready queue
    - Condition variables will observe when desired event occurs, unblock blocked process/therad (putting it back in ready queue)
        - Depending on scheduling queue, may preempt another running process & instantly run
- Often use condition variables to wait for condition to be true before continuing execution within a thread
    - More efficient than spinning on a shared variable
- Library calls (Linux) pthread_cond_t c declares condition variable c
    - Associated operations: wait(), signal()

Waiting Lists

May have multiple threads waiting on multiple different things

- Will have a condition variable for one of any number of events
- Q: what if several threads are waiting for the same thing?

Waiting lists - for each completion event, have associated waiting list

- Contains all threads/processes waiting for event to occur
    - When posting event, consult list to determine who’s waiting for that event
    - Choice of what to do with waiting list depends on application

            - Ex: wake up everyone, once-at-a-time in FIFO order, once-at-a-time in priority order, etc.
                - Last option - want to avoid starvation if high priority processes keep joining queue

Q: Who to wake up?

- pthread library - can manage condition variables in various ways
    - Ex: pthread_cond_signal, wakes up one blocked thread; pthread_cond_broadcast, all blocked threads
        - Broadcast approach may be wasteful (extra overhead) if event can only be consumed once - use FIFO waiting queue to solve
            - Is a covering condition - wakes up all threads that need to be woken, but incurs potential cost from waking too many
        - pthread_cond_wait takes mutex as parameter, assumed lock when wait() is called, puts caller; wait() releases lock, puts caller to sleep atomically
            - Must re-acquire lock before returning to caller; intended to stop race conditions when going to sleep
        - pthread_join() - enter queue
        - Always hold lock when doing either waiting/signaling

Producer/consumer problem (bounded buffer problem)

- Might have one or more producer threads, generating data items and placing in buffer; consumer threads grabbing from buffer, consuming in some way
    - Ex: issuing & processing HTTP requests in multithreaded web server, piping outputs between programs
    - May have that producer wakes up consumer upon filling buffer, fills buffer until it is full; consumer sleeps until buffer is non-empty, wakes up producer when buffer is no longer full
- Bounded buffer is shared resource, must synchronize accesses
    - Otherwise: may schedule more consumers than there are resources to consume
        - Occurs if a sleeping consumer is marked as ready by producer, but another awake consumer is scheduled first

            - Signaling a thread (wakeup) only provides hint of change in system state, not a guarantee that the state seen upon scheduling will be that as is desired (**Mesa semantics**)
            - Vs Hoare semantics - more difficult, guarantees woken thread will run immediately
                - Almost all systems use Mesa
    - Fix: use if, not while, for condition variables
- Implementing producer/consumer
    - Have two condition variables - bufferFull and bufferEmpty
        - Ensures consumers will only wake producers (not other consumers) and vice versa

Locking and waiting lists

- Multiple threads waiting for a lock - don't want to spin for lock
- Alternative: can build locks with associated waiting list, wake up in FIFO order (e.g.)
    - Issue: waiting lists are shared data structure, implementation would likely have critical sections; would need to be protected by a lock
        - Rec: locks are a software structure
    - Called **sleep/wakeup race condition**
        - Ex: given process A with sleep function, process B with wakeup function
            - Expect: if event has already happened, A adds self to queue and sleeps; after it happens, wakeup is called and re-queues
                - B has locked resource, A needs lock; A calls sleep; eventually, B finishes and will call wakeup() to release lock, waking up A
            - A may see that event is not posted; context switch to B, B sees nobody in queue, doesn't call wakeup; context switch to A, A enqueues and sleeps; will never be woken up (always be blocked)
        - Solution: some OSes have setpark() syscall (or similar), indicates a thread is about to sleep
- Solving problem
    - Have critical section in sleep, need to prevent anyone from waking up or interfering with queue while in the critical section
        - Mutual exclusion problem, but can't add another software lock

- Linux futex - each futex associated with specific physical memory location + queue (in kernel)
    - Have two futex calls available
        - futex_wait(address, expected) - puts caller to sleep, assuming value at address is equal to expected value; otherwise, returns immediately
        - futex_wake(address) - wakes one thread waiting on the queue
- Two-phase locks - have an initial spin phase, followed by subsequent sleep phase
    - Enter sleep phase if lock is not acquired after spinning a set number of times and enqueue to wake up later
    - Motivation: spinning may be useful if we think a lock might soon be realized

Semaphores

Semaphores - theoretically (i.e. mathematically) correct way to implement locks

- Thoroughly studied and precisely specified; however, more limited in practice/implementation

Computational semaphores (Djikstra) - classic synchronization mechanism

- Behavior well-specified, universally accepted; foundation/standard reference for most synchronization studies
- Are more powerful than simple locks - contain a **lock counter** [rather than binary flag] + **FIFO waiting queue**
    - Semaphore has two parts - integer counter (initial value unspecified), FIFO waiting queue (initially empty)
    - Semaphores only have two legal operations (besides destroying it):
        - P (proberen/test) - wait [determine if semaphore is available, enqueue if not]
            - Decrement counter
                - If counter >= 0, return [obtained lock]
                - If counter < 0, add process to waiting queue
                    - While waiting for semaphore, always blocked; cannot run
        - V (verhogen/raise) - post/signal
            - Can only be done if in control of semaphore/"lock"
            - Increments counter
                - If queue is non-empty, wake one of the waiting processes
- Implementing semaphores
    - Mutual exclusion - treat reading/writing to counter as critical sections

Can do multiple types of synchronization with semaphores

1. Using semaphores for exclusion [**binary semaphores**]
    - Initialize counter to value 1 (counter, when negative, reflects number of waiting threads)
    - Use P/wait to take the lock, V/post to release

        - First wait will succeed; subsequent waits will block
        - V/post will unblock any waiting threads
2. Using semaphores for notifications [notifying after events occur]
    - Initialize semaphore count to 0; count reflects # of completed events
    - Use P/wait to await completion
        - If was already posted, will return immediately; otherwise, all callers will block until V/post is called
    - Use V/post to signal completion
        - Increment the count; if any threads are waiting, unblock the first process in line
    - Doesn't broadcast to entire queue, only to first (only one signal)
        - Might be done in producer/consumer situations - consumers wait until work comes in from producer
3. Using semaphores for counting [counting semaphores]
    - Might have a limited number of available resources (integer-numbered), want to control access to the resources
        - Acts as a throttling/admission control mechanism
    - Initialize semaphore count to # of available resources
    - Use P/wait to consume a resource
        - If available, return immediately; else, all callers block until V/post is called
    - Use V/post to produce a resource
        - Increment the count; if any threads waiting, unblock first in line
    - Similar to above, only one signal per wait; no broadcasts

Limitations of semaphores

- Semaphores are a very basic mechanism - simple, only a few features
    - More commonly used for proofs than synchronization
        - Are still supported by most OSes
- Lack many practical synchronization features
    - Prone to certain synchronization problems
        - One cannot check the lock without blocking
        - Do not support reader/writer shared access
        - No way to recover from wedged V operation

- No way to deal with priority inheritance

Sec. 9: Synchronization Problems

Locking to solve high-level synchronization problems (practical methods)

- Mutexes (Linux/Unix) - intended to lock sections of code
    - Used via calls to mutex operations within code
        - Expectation is that locks are only held briefly
    - Typically used by multiple threads of the same process (multithreaded programs), implemented via memory/data area of that process
        - Low overhead and very general
- Object-level locking
    - Mutexes mainly protect only code critical sections for short periods of time
        - Threads must all be operating in same address space/data
    - Persistent objects (e.g. files) - more difficult
        - Many different programs can operate on them; may not even be running on single computer (in the case of remote filesystems)
            - I/O operations involved mean critical sections likely to last much longer

Often: want to lock objects, rather than code

- Typically somewhat specific to object type, often via OS locking mechanisms
- Linux files: can perform Linux file descriptor locking
    - Use flock() syscall - can specify a file descriptor and operation
        - Operations: LOCK_SH (shared lock, multiple allowed); LOCK_EX (exclusive lock, one-at-a-time); LOCK_UN (release lock)
    - Lock applies to open instances of same file descriptor
        - Can pass file descriptor to different processes, but distinct opens are not affected; more commonly used in multithreaded programs
            - Ex: forking will cause fd to be shared amongst children
    - Locking is purely advisory, not mandatory

Enforced vs advisory locking

- **Enforced locking** - nobody can access locked object without obtaining the lock

    - Typically done by ensuring that object cannot be accessed directly, only through an intermediary that will enforce the lock [provides service of accessing the object]
        - Ex: OS objects like files require syscall, could insert lock enforcement when handling syscall
    - Are always honored, no matter whether user wants it or not
- Advisory locking - convention that “good guys” agree to, are expected to follow (though they are not required to obey it)
    - Users expected to lock object before calling methods
        - Need to write code to honor these locks
    - Gives users more flexibility in what to lock and when
        - Also gives users more freedom for error
    - Mutexes, flocks() are advisory locks

Linux ranged file locking

- lockf() syscall (Linux) - waits on a file descriptor, but works on entire file (even a different open, with different fd, will see the same lock)
    - Supported commands: check/non-blocking request lock, get/wait for lock, release a lock
    - With lockf - can lock only subsets (i.e. certain byte ranges) within a file via offset and len arguments
    - Locking may be enforced, depending on underlying file system

Reader-writer locks

- In some cases - since reads do not modify data, may say that lookups can happen in parallel as long as no inserts/writes are occurring
    - Implementing: have separate read, write locks (via semaphores)
        - For read - have reader counter denoting number of active readers
    - Allow as many readers to get read lock as desired + increment read counter; writers must wait for all readers to finish before updating
        - Issue: readers may starve writers (fairness problem)
- Issue: add more overhead, doesn’t always do better than regular locks

Dining philosophers

- Solution: make sure at least one philosopher grabs forks in a different order/direction than the others (ensures no infinite cycle of waiting)

Locking Problems

Performance and overhead

- Typically, locking performed as OS system call (especially enforced locking)
    - Incurs system call overheads for lock operations
        - If called frequently, overhead adds up
        - Even if not in OS, extra instructions (by custom locks, e.g.) still incur overhead

Locking costs

- Want to avoid locking for too long - might block other processes for a very long time
    - Want to only lock for shortest possible critical sections - typically very brief, in and out in nanoseconds
        - Issue: overhead of locking operation may consequently be much higher than time spent in critical section
- Unsuccessful attempts to obtain a lock are even more expensive
    - Blocking is expensive - requires context switches, rescheduling, etc.
        - Is much more expensive than getting the lock, e.g. 100x
    - Lots of unsuccessful attempts can build up overhead even more
        - Performance depends on probability of conflict - higher probability of conflict results in even more performance penalty
            - Cexp = Cblock * Pconflict + Cget * (1 - Pconflict)
- Costs are even worse with more threads competing for a resource
    - One process gets resource, other processes get in line behind them
        - Create a **convoy** - list of threads/processes all waiting on a single resource, will only be able to access one at a time
            - Results in parallelism disappearing (can only run one at a time), creating a bottleneck

Convoy formation

- With sufficiently high amount of fraction of time spent in critical section, probability of conflict goes to 100%
- In general: Pconflict = 1-(1- (Tcritical / Ttotal))^threads (if nobody else is in the critical section at the same time)
    - Unless a FIFO queue forms:
        - Pconflict = 1-(1-((Twait+Tcritical)/Ttotal))^threads
            - Twait grows as Pconflict grows, causing feedback loop; eventually, becomes almost certain that a conflict will occur
    - If Twait reaches mean inter-arrival time, line becomes permanent -> parallelism stops, throughput drops to that of just 1 process
- Solving convoys
    - Want to minimize fraction of time spent in critical section(s)

Priority inversion - occurs in priority scheduling systems that use locks

- Might have that low-priority locks obtains mutex uncontested, but is then preempted by a high-priority process that blocks upon asking for the lock
    - Forces high-priority process to wait for low-priority process to complete; effectively reduces priority of high-priority process, to low-priority process'
- Ex: Mars Pathfinder rover
    - If a high priority process blocks, but low-priority process is preempted by very demanding medium-priority process - high priority task doesn't run for a very long time
        - Caused system to keep rebooting, since it would notice high-priority task had not ran and assume an error occurred
- Handling priority inversion problems
    - Can temporarily increase priority of low-priority task
        - Priority inheritance - if a task is blocked by a lower-priority task, bump the lower-priority task's priority to that of the higher-priority task until lock is released
        - Ensures task is not preempted

Solving locking problems

- Reducing overhead - not much to be done

        - Locking code in OSes is usually highly optimized; most users can't do better
- Reducing contention
    - Want to try and eliminate critical section entirely, if possible
        - Can eliminate shared resource (e.g. give everyone their own copy, or find a way to do without it)
        - Can use atomic instructions
        - Good if doable, but often won't be
    - Can eliminate preemption (i.e. disable interrupts) during critical section
    - Can reduce time spent in/frequency of entering critical section
    - Can reduce exclusive use of serialized resource
        - Can share for read-only rather than needing to enforce critical section
    - Can spread requests out over more resources
        - Ex: can divide resource into multiple pieces, associate separate critical section for each
            - Reduces chance of contention on any particular piece

Lock-based concurrent data structures

- Counters
    - Just using mutexes is increasingly inefficient with more threads updating/accessing the data for more complex data structures
        - Complex data structure -> worse performance
    - Want to make data structures **thread safe** (usable across threads) by adding locks + have scalable counting
        - Scalable - threads complete as quickly on multiple processors (in terms of total time, summed up across all concurrent threads), as a single thread on one processor (perfect scaling)
    - To scale, use approximate counter:
        - Represent single (logical) counter via numerous local physical counters (one per CPU core) + single global counter
            - Threads each increment the local counter independently; local counters synchronized via local locks
                - Local updates are scalable

        - Local values periodically transfer to global counter by acquiring global lock, incrementing by value of local counter
            - Reset local counter to zero afterward
        - Depending on frequency of local-to-global transfer: lower frequency means better scaling, but larger difference between local vs global counters
            - Since updates to global counter are non-scalable
- Linked lists - want to make insert, lookup remain correct under concurrent inserts
    - Can lock critical section within insert
        - Issue: doesn't scale particularly well
    - Hand-over-hand locking/lock coupling - rather than a single lock for entire list, have a lock per node of the list
        - When traversing list - code grabs next node's lock and releases current node's lock
        - In practice: not much faster than single lock due to overheads of acquiring/releasing locks
- Queues - have two locks for head and tail
    - Use locks for enqueue and dequeue, respectively, in the case of a normal queue
    - Bounded queue - enables thread to wait if queue is empty/full
- Hash table - via concurrent lists
    - Has a lock per hash bucket, allowing for much concurrency

Making Synchronization Easier

Locks/semaphores/mutexes are hard to use correctly; want to make synchronization easier for programmers

- Can identify shared resources (objects requiring serialization), have compiler automatically generate locks/releases via appropriate mechanisms
    - Programmer can ignore serialization/critical sections, just write code
- ***Monitors***/***protected classes*** contain a built-in mutex
    - Mutex acquired on any method invocation, automatically released upon return
    - Clients can ignore locking; class is automatically protected
    - Issue: is very conservative
        - Locks entire object on any method, for entire duration of method invocations -> can cause performance problems

        - May eliminate parallelism, lead to convoy formation
- Java synchronized methods - each object has an associated mutex, only acquired for specified methods
    - Not all methods need to be synchronized
    - Nested calls (by same thread) don't reacquire; mutex released upon return
    - Static synchronized methods lock class mutex
    - Allow for finer locking, less deadlock risk; but need developer to manually identify serialized methods, may still be too conservative

Deadlock

Classes of Synchronization Problems:

- ***Atomicity Violations***: When a code region is intended/assumed to be atomic, but the atomicity isn't enforced during execution
    - Solved via locks, atomic instructions
- ***Order Violations***: When the desired order between (groups of) memory accesses is flipped, or otherwise different from intended
    - May attempt to access a memory region that has not been initialized, e.g.
    - Can solve by enforcing ordering; e.g. use condition variables to have later accesses wait for preceding ones
- ***Deadlock***: When two entities (e.g. threads) have each locked some resource, but each needs the other's resource continue/unlock its own (result: neither will ever unlock)

Deadlock often occurs in cooperating parallel interpreters

- Are relatively common in complex applications, can result in catastrophic failures
- Are difficult to find through debugging - happen intermittently, hard to diagnose
    - Modern software depends on isolated/independent services - require resources, but are complex and resource needs are always changing
    - Easier to avoid via careful design

Two resource types: ***commodity*** and ***general resources***

1. ***Commodity resources***: resources where clients need an amount (e.g. memory)
    a. Deadlock occurs when resource is over-committed; can be avoided via a resource manager
2. ***General resources***: clients need a specific instance
    a. Ex: particular file/semaphore, particular message/request completion
    b. Deadlock from dependency relationships; prevented at design time

Four conditions for general resource deadlock:

1. ***Mutual exclusion***: The resource(s) in question can only be used by one entity at a time, i.e. threads have exclusive control over owned resources.

2. ***Incremental allocation***: Processes/threads are allowed to ask for resources whenever they want during runtime, rather than having resources explicitly pre-allocated.
    - Alt. (***hold-and-wait***): Threads hold allocated resources (rather than releasing them) while waiting for additional resources
3. ***No preemption***: Once a resource is reserved, it can't be taken away (forcibly or not).
4. ***Circular waiting***: The graph of resource requests must contain some kind of cycle.
    - Otherwise, someone can complete without waiting -> no deadlock

<u>Avoiding Deadlock</u>

<u>Reservations</u>

For commodity resources, can utilize **reservations** - force clients to reserve resources in advance.

- Resource manager tracks reservations, only grants more if resources available
    - Can detect over-subscriptions before the resources are allocated, eliminating deadlock from over-commitment
    - May cause client to fail if resource is not granted; forces client to deal with reservation failures

Problem: ***Overbooking*** vs ***under-utilization***

- Processes can't perfectly predict resource needs, may ask for more than is actually needed (to ensure success) -> two approaches to allocation:
    - Grant requests only until everything is reserved -> some resources won't be used (***under-utilization***)
    - Grant requests beyond available amount (***overbooking***) -> in rare instances, may not have enough resources to fulfill all reservations
- Operating systems typically don't want to kill random processes; will often underbook instead, reserving remaining resources in case of emergency/superuser use

Rejecting requests: In some cases, with advance notice, an app may be able to gracefully adjust its services to work without an unavailable resource

- Otherwise: if app is in the middle of servicing a request, may have other resources allocated + request half-performed; need to unwind/roll-back (complex)

General Resource Deadlock

Can prevent general resource deadlock by attacking any of the four conditions:

1. *Mutual exclusion*
    - Deadlocks can't occur with shareable resources, are rarer when using reader/writer locks (rather than standard locks)
        - Can design ***lock-free***/***wait-free*** data structures that don't need locks; via replacing critical sections with single atomic instructions, e.g.
    - Processes can't deadlock on their own private resources
2. *Incremental allocation* - deadlock requires blocking held resources, requesting more
    - Can allocate all resources in a single operation, rather than incrementally
        - ***All or nothing***: Either all resources (e.g. locks) are obtained & locked, or the request fails and nothing is locked.
            - Ensures no partial resources left locked -> preventing deadlock
        - Non-blocking requests that can't be satisfied immediately will fail
        - Can perform allocation (e.g. acquiring multiple locks at once) atomically, e.g. via having a lock on making requests
            - Issue: May not be doable if problematic regions are within an external routine, e.g.
    - Can disallow blocking while holding resources
        - Must release all held locks prior to blocking, reacquire locks after returning
        - Reacquiring locks can be done in an all-or-nothing manner - no deadlock
3. *No pre-emption*
    - Can break deadlock by confiscating resources
        - Can implement resource "leases" with timeouts and lock-breaking; allow resources to be seized, reallocated by system where necessary
            - System must enforce revocation - invalidate previous owner's resource handle, or kill if not possible
        - May result in damage to resource, e.g. if previous owner was within critical section -> need mechanisms to audit/repair
            - Can design resources with revocation in mind
    - OS seizing resources
        - Can revoke by invalidating process' resource handle

            - Ex: if process needs system service to access resource, OS can make service stop honoring requests
        - Cannot revoke process' access to a resource if it has direct access (e.g. if resource is part of process' address space)
            - Revoking access requires destroying address space -> kills process
4. *Circular dependency*
    - Can use ***total (resource) ordering*** - ensure all requesters allocate resources in the same order (e.g. R1 -> R2, but never R2 -> R1)
        - Assumes resource ordering is known - by resource type/relationship, e.g.
        - Most common practical prevention technique
            - May require lock dance - release R2, allocate R1, reacquire R2
    - ***Partial ordering*** (only enforcing ordering within a subset of overall resources) is less effective, but may be easier to implement in certain systems

Preventing Deadlocks

- Can use combination of various strategies - simply need to ensure every resource has at least one solution (with leases/lock-breaking as a last resort)
- Can also avoid locks via smarter scheduling: if the scheduler knows the conditions (e.g. which threads need to be running simultaneously) needed for deadlock, can schedule processes/threads such conditions are never satisfied
- OS must prevent deadlocks in system services; applications are responsible for safeguarding their own behavior

Deadlock Detection and Recovery

In many cases, deadlocks occur very rarely; it may be more expensive to avoid/prevent them than to simply ignore them/allow them to happen -> solution: *ignore the deadlock problem*.

- Issue: What happens when a deadlock does occur?

Strategy (***Detect and Recover***): Can allow deadlocks to happen, but implement mechanism to detect them when they do occur & perform an action to break deadlock

- To detect deadlocks, need to identify all resources that can be locked
    - Difficult to do in an OS, especially if locks are application-level

    - Can maintain wait-for graph (or equivalent structure), update structure and check for deadlock on every new request

Ex: Some applications (e.g. database systems) use internal application-level locking - built into their own code, but not an OS feature

- OS isn't aware of locks, can't help in handling them; but since the system knows all relevant locks, it can implement deadlock detection
- Typically handle deadlocks by rolling back deadlocked transactions

General Synchronization Bugs

Not all synchronization problems are due to deadlock

- Many other non-deadlock problems can still cause system to hang, prevent progress
    - Deadlock detection is a poor solution to general synchronization problems - is complex to implement, only works for true deadlocks
- Can use service/application ***health monitoring*** instead
    - Can monitor application progress, submit test transactions (heartbeat messages); if the response takes too long, declare the service to be "hung"
    - Is easy to implement, can detect wide range of problems

Related synchronization problems (that aren't deadlocks)

- ***Live-lock***: running process won't free resource R1 until a message is received, but message sender is blocked while waiting for resource R1
    - Can use polling (system continuously check if process needs attention) rather than interruptst
    - Alt def: two processes are not blocked, but are unable to make progress on account of the other holding a shared resouce
- ***Sleeping Beauty***: a process is blocked, but the awaited event will never happen
    - Can occur due to sleep/wakeup race, e.g.
- ***Priority inversion*** hangs
- Aren't deadlocks (won't be found by deadlock detection), but leave the system hung -> can be handled by health monitoring

Health Monitoring

Detecting Issues

Have various methods for detecting issues:

- Can look for obvious failures (e.g. process exits/core dumps)
- Can detect hangs via passive observation - look at process CPU consumption (blocked -> low consumption), network and disk I/O requests
    - Can also watch message traffic/transaction logs to see if work is happening
- External health monitoring - via pings, null requests, standard test requests

    - Can tell clients to send failure reports to a central monitoring service when a server becomes unresponsive
    - Can tell servers to send periodic ***heartbeat messages*** to the monitoring service
- Internal instrumentation - white box audits, exercisers, monitoring

Often use a combination of multiple methods for detecting issues

- Can use an internal monitoring agent for most application failures/hangs
- Internal monitoring agent sends status reports/heartbeat messages to a central monitoring agent, which should notice any failures in the internal monitoring agent
    - External test service can generate test transactions for additional system monitoring outside of the application itself

Dealing with Unhealthy Processes

Can kill and restart all affected software, but want to minimize number of processes killed

- Hung processes may not be the troublemakers
- Kills and restarts depend on service APIs/protocols
    - Apps must be designed for cold/warm/partial restarts
- Some systems may define restart groups - groups of processes to be started or killed as a group
    - Can define inter-group dependencies - dictate order of restarts

Failure Recovery Methodology

Principle: Retry if possible, but not forever

Want to be able to reestablish communication, resume working with minimal disruption

Other Strategies:

- Rollback failed operations and return an error
- Continue with reduced capacity/functionality - only accept those requests that can be handled, reject others
- Automatic restarts - cold (restart all operations from scratch), warm (restore to last saved state & resume), partial; reset and reboot

In the case of failed recoveries, can implement escalation mechanisms

- May incrementally restart more groups/reboot more machines

False Positives

In some cases, a monitoring service may notice a failure that is actually a false positive

- Ex: a heartbeat message may be lost due to network error/process delay, without any major failure in the process
    - Central monitoring service might itself have problems
- Expensive to declare a process failed, especially if it may still actually be running
    - Best option: have failing systems detect their own problems, inform peers, and then shut down cleanly & automatically

Want to make sure a failure report is actually a failure - often implement a "*mark-out threshold*" to provide some margin of error

- May wait for multiple missed heartbeat messages
- May wait for multiple other processes/nodes to notice, report problem

Want to make sure mark-out thresholds are properly tuned

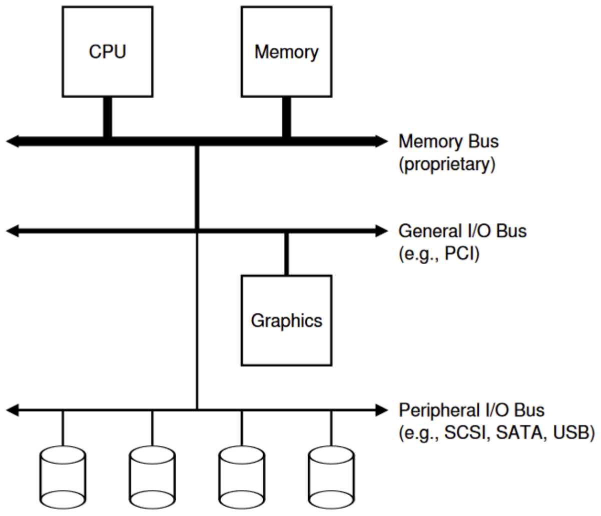
- Forcing unnecessary fail-overs from healthy servers risks unnecessary service disruptions, but waiting too long on a failed server can also prolong service outages
- Misdiagnosing problem causes and restarting the wrong components may worsen issues

Operating System Components

Sec. 10: Devices and Device Drivers

Basic computer architecture:

- CPU is attached to main memory via ***memory bus*** (or other connection)
- General ***I/O bus*** connects system to high-performance I/O devices (e.g. graphics cards)
- ***Peripheral (I/O) bus*** connects system to slower devices, e.g. disks/mice/keyboards
    - Higher-performance buses (i.e. general I/O bus) have more limited capacity
    - Lower-performance components placed further away from CPU


CPU
Memory
Memory Bus (proprietary)
General I/O Bus (e.g., PCI)
Graphics
Peripheral I/O Bus (e.g., SCSI, SATA, USB)

Modern systems may have more specialized interconnects:

- Intel chipsets connect CPU directly to I/O chip via Intel DMI for better performance
- Other devices connect to I/O chip via various connections
    - eSATA: higher-performance version of SATA for storage devices
    - PCIe for network connections, additional high-performance storage
    - USB for low-performance connections

Two important components of a device:

1. ***Hardware interface*** provides system software the ability to control hardware
2. ***Internal structure*** implements various pieces of functionality on hardware chips

Hardware interface (simplified) consists of 3 registers:

1. The ***status register*** holds current status of the device
    a. The OS (i.e. OS processes) may use ***polling*** (continuous rechecking) to await device signals, e.g. READY
2. The ***data register*** helps transfer data to/from the device
    a. Facilitates ***programmed I/O*** (**PIO**) - CPU-drive data movement
        i. OS may place the data needed for a write into data register, e.g.
3. The ***command register*** is read by the device to perform tasks
    a. OS writes commands to the command register to provide instructions

Instead of polling, may opt to use ***interrupts***:

- OS can issue request, then put calling process to sleep; device can raise an interrupt (upon completion, e.g.) to jump to interrupt handler
- More CPU-efficient than polling; can run other processes in the meantime
    - Exception: In cases where a task is expected to be performed quickly (first poll will return success), faster to poll than to handle an interrupt
        - Can use hybrid/two-phased approach - poll for a little bit, then use interrupts if device is not finished
    - Can also be dangerous for networks - if too many interrupts are generated (from a large stream, e.g.), OS may only process interrupts and never make progress on user-level processes (***livelock***)
- Can optimize via ***coalescing*** - devices may want a bit before delivering interrupts, and potentially batch multiple interrupts into a single delivery
    - Lowers overhead, but results in higher latency

10.1: Peripheral Devices

Computers typically have many (***peripheral***) ***devices*** attached; each device needs to have associated code to (i) perform various operations, and (ii) integrate with the rest of the system

- Modern OSes - most code dedicated to handling devices

Peripherals in computers

- Are typically built to perform specific tasks/commands, not arbitrary computations
- Are usually attached to a bus; allows for talking to other components
  - Signals on bus used to provide orders to the peripheral
    - Otherwise, each device generally operates asynchronously from other devices/activities
  - Can also transmit bus signals from peripheral to indicate results

Devices and performance

- Most devices are very, very slow compared to CPU/bus/RAM - results in challenges (especially performance challenges) when managing devices
  - System code must handle mismatch between CPU speeds needed for system behavior, and significantly slower device speed for device interactions
- Peripheral devices mostly handled in OS, not user-level, code
  - Peripherals may be needed for system correctness (e.g. storage devices), shared among multiple processes, or security-sensitive

<u>Device Utilization</u>

System performance is often limited by key devices, e.g.: file system I/O, swapping, network communication (all significantly slower than CPU)

- Important to keep key devices busy, such that CPU is not held up waiting for devices to respond; want to always be working on requests
  - Device idling results in lower throughput (both device and system-wide)
  - Delays can disrupt real-time data flows, worsening performance and potentially losing irreplaceable/important data

Can improve device utilization via ***parallelism*** - since devices operate independently, can have devices & CPU operating in parallel

- Issue: both devices and CPU need to access RAM
    - Modern CPUs try to avoid going to RAM - via registers/caching on CPU chip, e.g.
        - If successful, CPU avoids using memory bus -> device has sole access
        - Side benefit: RAM is much slower than the CPU, so limiting CPU trips to the memory bus boosts performance
- ***Direct memory access*** (**DMA**) allows any two devices attached to the memory bus to move data directly without needing to pass it through the CPU
    - Moves data from device to memory at bus/device/memory speed
        - OS initiates DMA via instructions to DMA engine; waits for interrupt from DMA engine (signaling completion)
    - Bus can only perform one of DMA vs. servicing CPU requests -> still want to limit CPU requests to memory bus
    - Can use DMA for data transfers -> no delay, higher speed, less CPU overhead
        - Allows OS to work on other tasks in the meantime

To improve utilization, want to keep key devices busy

- Can create request queue to store pending requests
    - Requesters block to await request completion
    - When active request completes: device controller generates completion interrupt
        - OS accepts interrupt, calls appropriate interrupt handler
        - Interrupt handler posts completion to requester, then selects and initiates next request (e.g. memory transfer)
- Memory transfers - bigger transfers are better
    - All transfers have per-operation overhead (DMA-related, device-related, and OS-related)
        - Ex: instructions to set up operation, device time to start new operation, time and CPU cycles for servicing completion interrupt
        - Larger transfers -> lower overhead/byte

<u>**Device Communication**</u>

Two methods for device communication: ***I/O instructions*** and ***memory-mapped I/O***

1. I/O instructions: Explicit instructions sent from OS to device
    a. Usually privileged instructions, done only by the OS
2. Memory-mapped I/O: Hardware pretends device registers are available to OS as memory locations; converts writes to corresponding memory locations into writes to registers
    a. Removes need for new I/O instructions; only need the load/store instructions

I/O and buffering

- Most I/O requests cause data to come into memory or be copied to a device
    - OS uses ***buffers*** in memory as locations for sending/storing data
        - Data in buffers -> ready to send to a device; empty buffer -> ready to receive data from a device
    - OS needs to ensure buffers are available when needed
- Although bigger memory transfers more efficient, may not be convenient for applications - requests tend to be relatively small
    - Operating system can consolidate I/O requests - maintain cache of recently-used disk blocks to store large amounts of writes
        - Can accumulate small writes, flush out as blocks fill
        - To service request - read whole blocks, deliver data as requested
        - Enables read-ahead: OS can read/cache blocks not yet requested

Queues

- Want to have deep request queues for better performance
    - Maintains high device utilization
    - May be possible to combine adjacent requests or avoid performing a write entirely, improving efficiency
- Various ways to achieve deep queues - many processes/threads making requests, individual processes making parallel requests, erad-ahead for expected ata requests, write-back cache flushing

Memory-mapped I/O

- DMA not always best way to do I/O
    - DMA is designed for large, contiguous transfers; some devices (e.g. display adaptors) may want to perform many smaller, sparser transfers

- Instead: treat registers/memory in device as part of regular memory space
    - Access by reading/writing to those locations
        - Ex: bit-mapped display adaptor might treat each word of memory on display controller (on CPU memory bus) as a pixel
            - Application can use ordinary stores to update the display
    - Benefits: low overhead per update, no interrupts to service, relatively easy to program
- Memory mapping vs DMA
    - DMA more efficient for large transfers
        - Ensures better utilization of devices and CPU (device doesn't have to wait for CPU to transfer), but requires considerable per-transfer overhead for operation setup and completion interrupt
    - Memory-mapped I/O has no per-operation overhead, but every byte transferred requires its own CPu instruction
        - No waiting - device accepts data at memory speed
        - Better for frequent small transfers, but more difficult to share memory-mapped devices

10.2: Device Drivers

Device drivers are used by OS to send instructions to the hardware devices

- Device driver code is usually highly device-specific; each system device has its own driver
    - Tells device to perform a certain set of operations
    - When applications send a request, the OS must invoke the correct device driver
    - Many, many drivers within any OS; make up majority of kernel code
- Typically interact with the rest of the system only in limited, well-defined ways
    - Correctness of device drivers is critical for both device behavior and overall correctness, require lots of experience with the device in question
- Is the piece/level of the OS that needs to know how a device works (in detail)
    - Rest of system can make calls using standard interfaces while ignoring actual driver implementation
    - Ex: System requests (e.g. read, write) sent to file system (or using the ***raw interface***) are then routed by OS generic block layer to the appropriate device driver

Device drivers are inherently very modular thanks to ***driver abstractions***

- The OS will often define idealized device classes (e.g. flash drive, display, printer, network, etc.) with associated expected interfaces/behavior
    - All drivers in that class will support the standard methods
        - Allows compatibility with a large set of devices + abstracts the specific characteristics of that particular device
- Driver abstractions convey knowledge of how to use device, while allowing optimization for specific devices
    - Also encapsulate fault handling - how to handle recoverable faults, while preventing device faults from becoming OS faults
- OSes will generally provide the ability to “plug in” different drivers in well-defined ways as needed, swap drivers, etc.
    - Core OS (e.g. caching, general file systems, high-level network protocols) does not contain device drivers

Layering drivers

- Device-bus and application-bus interactions very standard and typically independent of the device itself
    - In between: device drivers (device-specific)

- Devices not processes -> not scheduled; work asynchronously from the CPU
    - Primarily utilize interrupts to talk to the CPU
    - Are often much slower than the CPU

Devices and buses

- CPUs and devices not connected directly, but rather both connected to a bus (not always the same bus)
- CPUs, devices communicate across bus; can use bus both to send/receive interrupts (device -> CPU) and transfer data/commands
    - Devices signal controller when done/ready; controller puts interrupt on bus, bus transfers interrupt to CPU
        - Can transfer data afterward, e.g.
- ***Interrupts*** - similarly to traps, but source of outside (rather than inside) CPU
    - Interrupts can be enabled/disabled by special CPU instructions - can tell devices when interrupts are allowed
        - May hold interrupts as ***pending*** until software is ready, e.g.

Generalizing abstractions for device drivers

- Device abstractions - OS defines idealized device classes w/ expected standard interfaces/behavior, implemented by individual device drivers
- ***Device-driver interface*** (**DDI**) - standard device driver entry points from OS to the driver ("top-end")
    - Are the basis for device-independent applications - allows system to exploit new devices, provide interface contract for 3rd party developers
    - Some entry points correspond directly to system calls (e.g. open, close, read, write)
        - Others associated with OS frameworks - e.g. flash drivers called by block I/O, network drivers by protocols
    - Common DDIs contain certain entry points that are shared across all device types (e.g. initialize, cleanup, basic I/O)

            - Specific device types may implement their own additional entry points via sub-DDIs, e.g. network, serial, disk entry points

Driver-kernel interface (**DKI**) specifies bottom-end services provided by OS to drivers

- Lays out permitted requests from the drivers to the OS kernel
    - Analogous to an ABI for device driver writers
        - Similar to an ABI, must be well-defined and stable
- DKIs are OS-specific, but generally implement similar sets of operations
    - Ex: memory allocation/data transfer, I/O management and DMA, synchronization and error reporting, dynamic module support

Linux device driver abstractions done via *class-based* system

- Have several overlapping superclasses (e.g. ***block devices***, ***character devices***; in some cases, network devices) with additional subclasses within each superclass
    - ***Character devices***: Devices that read/write one byte at a time
        - May be either stream or record structured, sequential or random access
        - Support direct, synchronous reads/writes via read(), write(), seek()
        - Ex: keyboards, monitors, most other devices; used by applications
    - ***Block devices***: Devices that deal with a block of data (usually fixed-size) at a time
        - Reads or writes a single-sized block (e.g. 4K bytes) of data at a time
            - Are random-access, support queued asynchronous reads/writes
        - Ex: disk drives for file system I/O; via request() call
    - Network devices - devices that send/receive data in packets
        - Originally treated as character devices; later became its own class (in some definitions)
        - Only used in the context of network protocols; have their own traits
        - Ex: Ethernet cards, 802.11 cards, Bluetooth devices
- Block vs character devices
    - Block devices span all block-addressable forms of random-access storage, e.g. hard disks, CD, flash, etc.
        - Require very elaborate services (compared to character device) - buffer allocation, buffer cache management, scheduled I/O, asynchronous completion, etc.

        - Block I/O services designed to provide high performance
            - Relied on by important system functionality (e.g. file systems, swapping/paging)
    - Character devices primarily for application use; can use DMA for large, efficient transfers between device and user-space buffers, e.g.

Device drivers identified via major device number

- Major device number specifies which device driver to user for it
    - Several distinct devices may have same drivers (e.g. multiple of the same disk drive, or logically distinct partitions of a disk drive) -> use minor device numbers to distinguish between
- Linux - device drivers accessed via file system
    - Have special files associated with a device instance - contain device major, minor numbers in file
    - Opening a file opens associated device; open/close/read/write/etc. calls mapped by OS to appropriate entry points of selected driver

Dynamically Loadable Kernel Modules

Some systems have the ability to dynamically select and load implementations to interfaces at runtime (rather than needing to specify at build time)

- Often done for device drivers - allows OS to identify, load device drivers for new devices as needed without having to pre-install drivers
    - Modern I/O buses support ***self-identifying devices*** - contain query-able type, model, and/or serial number for lookup in device driver registries
    - Hot-plug buses (e.g. USB) will generate events whenever devices are added/removed, triggering OS device loading/unloading
- OS runtime loader helps OS dynamically load driver dependencies

Using dynamically-loaded modules

- Module will contain initialization function to register devices + unload/removal function for device cleanup and deallocation
    - OS maintains table of registered devices + associated entrypoints for every operation, will forward requests to appropriate entrypoint

- Relies on existence of a stable standard interface

Sec. 11: File Systems

File systems provide a method for systems to store data persistently

- Is a core piece of functionality for system - even the OS is itself stored via file system
- "Persistently" - even after reboot/power down
    - Options: raw storage blocks (difficult to work with), database (overly rigid, more overhead), file systems (more intuitive, less complexity)
- Two primary parts of a file system: ***data structures*** and ***access methods***

Principle: Organize data into natural, coherent, self-contained units - ***files***

- Typically store each file in a way allowing efficient access
    - Files contain low-level "name" (***inode number***/***i-number***) for identification
    - File structure isn't relevant to OS; treated as simple sequence of bits
- Usually provide some way to organize collections of files for easy access

File systems store two types of (persistent) information for each file - ***data*** and ***metadata***

- ***Data***: The information stored by the user/applications into the file
- ***Metadata/attributes***: Information about the data, e.g. size/type
    - Data, metadata are typically stored together

File systems also provide ***directories***, allowing access to other files/directories

- Consist of an **(i)** an inode number + **(ii)** a list of pairs mapping user-readable file names/directory names to corresponding inode numbers
    - Can place ***subdirectories*** inside of other directories to construct a hierarchy

File systems and hardware

- File systems typically stored on hardware providing persistent memory (e.g. flash drive); implementation usually matched to hardware for performance reasons
    - File systems are an abstraction - a given file system might still work with multiple kinds of hardware
- Flash drive - solid-state (i.e. no moving parts) persistent storage devices
    - Fast to read/write, but a given block can only be written once - need to erase to write again (slower)

- Want file system to be storage medium-agnostic, i.e. have the same interface for the file system regardless of medium/hardware
    - File system priorities: persistence, ease of use, flexibility, portability across different hardware types, performance, reliability, suitable security

File system considerations

- Performance: Want to make file system fast, but many storage devices are significantly slower than CPU speeds
    - Need to somehow deal with mismatch in operating speeds
- Reliability: To ensure persistence, need to ensure file systems are absolutely error-free
    - Must not have any errors arising from concurrency, e.g.
- Security: File systems usually incorporate some form of access control model/mechanism
    - Need to provide guarantees that system will check access, enforce control - only the data owner can control who can access
        - Want to balance with performance

11.1: File Systems and the OS

File systems live on the kernel level, above block I/O interface

- File system API, virtual file system integration layer provide interface between system calls (higher) and file system (lower)

Have several layers of abstraction between application file accesses, and physical reading/writing of blocks by block devices

- ***File system API*** provides a single API for all files to users/programmers
    - Ensures portability - works regardless of actual file system implementation
    - Operations are mapped to file system implementation

Three categories of file system API system calls

1. ***File container operations*** are standard file management system calls; functionality does not depend on the contents of the file
    a. Manipulate files/directories as objects - get/set attributes, create/destroy, etc.
2. ***Directory operations*** provide organization to the file system

    a. Typically structured in a hierarchical or semi-hierarchical manner
    b. At a baseline, ***directories*** translate a name to a lower-level file pointer
        i. Operations: find a file by name, create new name/file mapping, list set of names, etc.

3. ***File I/O operations***: open files, read from/write to file, etc.
    a. Open - use name to set up an open instance
    b. Read/write - via logical block fetches, utilizing a ***file buffer*** to transfer between user space and file system
    c. Other operations: map file into address space (i.e. load from file system, page into memory), seek (change logical offset associated with open instance)

The ***virtual file system*** (**VFS**) layer provides an interface between the file system API and the actual file system implementation

- Acts as a federation layer - allows OS to work regardless of which file system implementation is being used
    - Can dynamically add new file systems as needed
    - File system implementations implement various basic methods, e.g. create/delete
- Higher-level clients see only standard methods/properties, not implementation
    - Applications can only use system calls to interface

File system layer implemented on top of block I/O, under VFS

- Generally want to be independent of underlying devices used in block I/O
- Typically all perform the same basic functions, e.g. mapping names to files, create, etc.
    - Map (file and offset) into (device and block)

May have multiple file systems within a single larger system

- Different types of storage devices may benefit from using different kinds of file system
- Different file systems may provide different services, offer different guarantees (despite sharing the same interface)
    - May have different performance characteristics, read-only vs read/write permissions, etc.
- May have different file systems for different purposes (e.g. a temporary file system)

Block I/O layer lives between file system and device drivers (via DDIs)

- Block I/O layer is usually a general one - abstracts away the specifies of the hardware
    - Implements standard operations on each block device, e.g. asynchronous reads and writes (taking physical block #, buffer, bytecount)
        - Also maps logical block numbers to device addresses
    - Encapsulates all particulars of device support - I/O scheduling, initiation and completion, error handling, hardware limitations, etc.
- Block I/O layer is typically designed to be device-independent, provides various capabilities to the system
    - Provides unified buffer cache for drive data, supporting pre-fetch read-ahead
        - Limits unnecessary (and slow) disk accesses
        - Only a single block I/O cache (rather than one per device, e.g.) - more efficient if multiple users are accessing the same file, provides better hit ratio than several caches
    - Provides buffers for data re-blocking
        - Handles buffer management - allocation/deallocation, writeback to changed buffers
        - Also helps adapt file system block size to device block, user request sizes

Mounting a file system

- After creating a file system and building (via mkfs() call), need to use mount() program to make it accessible within the broader file system tree
    - Takes existing directory as ***mount point*** and (effectively) pastes the new file system into directory tree at that point

File System Control

Primary role of file system is to facilitate storing/retrieving data, managing data storage

- File - at its core, simply a named collection of information
- Common operations: find first/next/specific block of file, allocate new block to the end, free associated blocks, etc.

Finding data on devices

- Space management is complex - must correctly manage space assigned to each file
    - Files continuously created/destroyed/modified
        - Poor placement, management may result in poor performance
    - Done on-device, via a master data structure <u>for each file</u>

On-device file control structures

- On-device, contain description of important attributes of a file (most importantly, where the data is located)
    - Implementations differ between different file systems, but almost all file systems have some kind of control structure
    - Also usually have an in-memory representation of the same information
- File typically consists of multiple data blocks (often changed, added/removed); control structure must be able to find them, preferably quickly
    - Want to avoid having to read entire file to find blocks near the end

In-memory representation

- On-disk, have structure pointing to device blocks (+ containing other information)
    - When file is opened, in-memory structure is created
- In-memory representation is not the same as device version
    - Device version points to device blocks; in-memory version points to RAM pages (or indicates block isn't in memory)
        - In-memory version also keeps track of which blocks have been written and which blocks haven't
    - In-memory structure usually also contains cursor pointer - indicates how far into the file data has been read/written
        - Issue: if multiple processes (that may or may not be cooperating) are accessing the same file, what should and shouldn't be shared?

Unix approach to in-memory representations

- Four levels:
    - Open (user) file references in process descriptor, e.g. stdin, stdout, stderr, etc.
    - Open file instance descriptors - contain offset, options, pointer to inode
    - ***I-nodes*** - in-memory file descriptors (Unix *inode*)

    - On-disk file descriptors (Unix *dinode*)
- Each level contains a pointer to one or more of the next
    - Two processes can share a file (instance) descriptor
    - Two descriptors can share an inode

Types of Files

Files are just binary byte streams; are understood by imposing structure, semantics (e.g. via a program that understands underlying format)

- Can be done via requiring user to invoke the correct command (e.g. calling gcc for C++ files), or having a registry associating programs with file types
    - Registry may be system-wide, program-specific, or an attribute of the file

Classing files

- Common approach: use file name suffix/extension (e.g. ".txt")
    - Convention in Unix; potentially required in Windows
- May also include "magic number" at start of file, indicating type (Unix)
- Can have file type as an attribute in systems supporting extended attributes

Other types of files

- Directories represent namespaces
    - Directory operations implemented via OS
- IPC ports are channels through which data is passed
- Some I/O devices may be placed in file namespace (e.g. disks), but not others

File metadata contains various attributes describing data

- Standard Unix attributes: type (e.g. regular vs directory), ownership, protection, creation/update/access dates, size/number of bytes (for regular files)
- Extended attributes may be implemented
    - May be a limited number/size of name=value attributes per file, or a pairing of each file with a "shadow file"/resource fork containing the attributes
    - Ex: encryption algorithm, signed certificate, checksum, supported languages

11.2: File System Structure

- Most file systems live on *block-oriented devices* - devices divided into fixed-size blocks (e.g. 512, 1024, 2048, etc.)
    - Most blocks used to store user data
    - Some blocks used for storing organizing “meta-data” - description of file system layout & state, file control blocks to describe individual files, lists of free/unallocated blocks
        - All file systems have these structures, but goals & implementations differ
- 0th block of a device usually reserved for ***boot block*** - code allowing the machine to boot an OS
    - For devices that are not bootable, 0th block is usually reserved anyway
        - True for all OSes, not just DOS
    - Not under the control of file system; file system typically ignores boot block entirely & starts at block 1

Managing allocated space

- Similar to memory management - want to avoid fragmentation
    - Allocate space in “pages” - internal, but no external fragmentation
- File control data structure determines how many chunks/pages a file can contain
    - Issue: file control data structure has limited capacity for pointers
    - Solution (***linked extents***): file control block contains exactly one pointer to the first “chunk” of the file; each chunk contains a pointer to the next chunk
        - Allows for adding arbitrarily many chunks to each file
        - Might have auxiliary “chunk linkage” table for faster searches

<u>The DOS File System</u>

DOS file system: first two blocks are boot block/Master Boot Record (MBR), then BIOS parameter block/BPB (specifies cluster size, FAT length); afterward is ***file allocation table***/**FAT**

- End of bootstrap block contains FDISK table - partitions disk into logical sub-disks in case of multiple file systems
    - Four etnries, each describes a disk partition

        - Each entry consists of: type (e.g. DOS/Unix), ACTIVE true/false, start and end disk addresses, number of sectors contained
        - First sector of partition called Partition Boot Record/PBR, OS/file system-specific; may contain a BIOS block
- Boot block and BIOS block may be combined into a single block in some systems
- After FAT, data clusters begin immediately (starting with cluster 1 - root directory)
    - Store files/directories; first file is root directory

DOS file systems divide space into clusters (i.e. chunks) - cluster sizes multiples of 512, fixed for each file system (numbered 1 to N)

- File control structure points to first cluster of a file
- Files are allocated in logical multi-block clusters, rather than physical blocks
    - Clusters/block is determined on file system creation; larger allocation chunks improves I/O performance, but results in more internal fragmentation
- FAT is a chunk linkage table - contains one entry per cluster
    - Entry (if cluster is allocated) contains number of next cluster in file; 0 entry means cluster not allocated, -1 entry signifies EOF
        - Used as free list and allocation tracker
    - File system sometimes also referred to as "FAT"
    - Is relatively small -> can keep in memory while file system is in use
- DOS FAT clusters - directory entries (i.e. FAT file descriptors) contain name, length, index of/pointer to first cluster
    - Combine file description, naming into a single entity
    - DOS directories contain a series of (fixed-size) entries
    - File allocation table entries each correspond to a cluster

DOS file system

- To find particular block of file - get number of first cluster from directory entry, follow pointer chain through FAT table
    - Entire FAT table kept in memory - no disk I/O required, but can still be slow to search for very large files
    - Max file system size determined by width of FAT (number of bits describing a cluster address); originally 8 bits, later expanded to 32 (FAT32)

The Unix File System

File index blocks - alternate way to keep track of where file's data blocks are on device

- With file control blocks - file capacity (or number of extents) is limited by number of pointers control block can hold
- Index blocks - basic file index block points to blocks
  - Some blocks contain pointers, which in turn point to blocks
  - Can point to many extents, but still a limit (though larger)

Unix system V file system

- Block 0 - boot block
- Block 1 - super block (specifies block size, number of I-nodes)
- Block 2+ - I-nodes; I-node #1 traditionally describes root directory
- Afterward: data blocks begin immediately after end of I-nodes

Unix file system is a ***multi-level index***: block pointers within an I-node may either link to a data block directly, or may require following pointers within blocks to reach a block

- Call intermediate blocks indirect, double-indirect, triple-indirect blocks depending on distance between I-node, data block (triple-indirect -> first of three intermediate blocks, e.g.)
- Advantage: compared to a simpler scheme (e.g. all block pointers are triple-indirect), provides greater flexibility depending on file size
  - Smaller file size -> can directly access, removing extra I/O of fetching indirect blocks off disk
  - More layers of indirection -> larger maximum file size

On-disk inode contains 13 block pointers

- First 10 point to first 10 blocks of file; 11 through 13 point to indirect, double indirect, and triple indirect blocks (respectively)
  - Each of 11-13 point to 1024 blocks, one indirectness tier removed (e.g. triple indirect block pointers to 1024 double indirect blocks)

Performance considerations

- Inode is in memory whenever file is open -> can find first 10 blocks without extra I/O
- Afterwards: reading indirect blocks requires I/O operations
    - Only 1-3 extra I/O operations per thousand blocks; any block can be found with 3 (or fewer) reads
    - Indirect blocks, once loaded, will be kept in buffer cache by block I/O
        - Sequential file processing will keep referencing

File system usage

- Calls to open() return a file descriptor (integer, private, managed per-process)
    - Rec: file (instance) descriptor contains offset, options, pointer to inode
        - Also specifies readable/writable, integer used as reference
        - Unix fiel table stores all open file descriptors
            - Holds reference count for each descripptor; removes entry uwhen reference count reaches 0
    - Repeated calls create multiple descriptors for te same file
    - Can use fork(), dup() to share entries in open file table (i.e. descriptors) with child processes
- read(), write() take file descriptor (+ data buffers) as arguments
    - lseek to change position (offset) within file
- close() indicates a program is done with a file (descriptor as argument)
- stat(), fstat() to view metadata
- mkdir() for directories + opendir/readdir/closedir/rmdir

11.4: Allocation Strategies

Allocation/Deallocation Problem - file systems usually aren't static, involve lots of creating, destroying, and modifying (e.g. extending/shortening) files

- Operations may convert unused blocks to used blocks, or vice versa; need a correct and efficient way to do so
    - Can maintain free list of unused blocks, similar to memory allocation
        - Alt: free elements are fixed-size blocks -> bit maps or similar
- Other considerations: locality for hard disks, erasure and load leveling for flash drives

1. Creating a new file
    a. Allocate a free file control block
        i. Unix: Search super-block free I-node list/bitmap, take first free I-node
        ii. DOS: search parent directory for an unused directory entry
    b. Initialize a new file control block with type, protection, ownership, etc.
    c. Give new file a name, i.e. write it into a directory
2. Extending a file
    a. Application first requests new data be assigned to a file
        i. May be via explicit allocation/extension request, or implicit (e.g. by writing to a currently nonexistent block)
    b. Find a free chunk of space via traversing free list/bitmap, and remove from free list/bitmap
    c. Go to appropriate place in file/extent descriptor and update to point to newly-allocated chunk
        i. Generally don't need to update directory entry
3. Deleting a file
    a. Release all space allocated to the file
        i. Unix: return each block to the free block list
        ii. DOS: rather than freeing space, uses garbage collection
            1. Will search out deallocated blocks, add to free list later
    b. Deallocate file control lock
        i. Unix: zero inode and return to free list
            1. Afterward: delete directory entry
        ii. DOS: zero first byte of name in parent directory (indicating that the directory entry is no longer in use)

Free space maintenance

- File system manager manages free space
    - Want getting/releasing blocks to be fast (since they are extremely frequent), avoid doing I/O as much as possible
    - Unlike memory - which block we choose matters
        - Can't write fully-written flash blocks

        - May want to do wear-lelvelling, keep data contiguous
        - Hard disk drives - other issues
    - Want to address both speed, preference concerns
- Free list vs bitmap
    - Free lists - lists of pointers to free memory elements; searched via following pointers
    - Bitmaps - structures with a single bit indicating if a piece of memory is free
        - Search via logical operations, e.g. shifting and ORing (much quicker than following pointers)
        - Used in most, but not all, file systems

Large files may require multiple blocks

- Some file systems: can find sequences of free blocks to make portions of the file contiguous (improving performance)

<u>File System Performance</u>

Performance improvement strategies:

1. ***Increasing transfer size***
2. ***Read caching***
3. ***Write buffering***

Transfer size

- Per operation overheads are high (DMA startup, interrupts, device-specific costs); larger transfer units more efficient
    - Q: What unit (e.g. chunk size) for allocating storage space?
        - Similar to before - need to consider fixed- vs variable-sized chunks, fragmentation vs efficiency
- Flash drive issues - very fast write-once/read-many access to any location, but can only erase very large areas of memory

Read caching

- Persistent storage I/O takes a long time; want to minimize

    - Can maintain in-memory cache, check cache for whether blocks can be reused before scheduling I/O
    - Other improvements: deep queues, large transfers
- Read-ahead - can request blocks from device before explicitly asked for (by processes), reducing wait time
    - May be done when client specifically requests (or is otherwise performing) sequential access
        - Risks wasting device time, buffer space on unneeded blocks
- Static vs dynamic partitioning
    - Static caching: Can create a fixed-size cache for read blocks, taking up a certain portion of memory
        - Used early, but may take more space than needed
    - Dynamic partitioning: Can create ***unified page cache*** between virtual memory pages and file system pages
        - Can allocate memory more flexibly between virtual memory and file system based on need

Write buffering: Rather than immediately writing to device, can first place (most) device writes into a ***write-back cache***, then flush cache to device at a later point in time

- Since writes are very inefficient - may require several separate I/O operations (read data bitmap, write bitmap, read/write inode, write block, potentially modify directory)
- Benefits:
    - Aggregates small writes (i.e. less than full-block) into large writes
    - Eliminates writes that don't matter (e.g. if application subsequently rewrites same data or deletes file)
    - Can use to accumulate large batches of writes, acting as deeper queue for better disk scheduling
- Issue: Introduces risk of losing updates if system crashes before flush
- Can use fsync() call on a specific file descriptor to immediately force all dirty (i.e. not yet written) data for that file to disk
    - May also need to call fsync() on directory to ensure newly-created files appear in directory listing

    - Can also bypass cache using direct I/O interfaces, or bypass file system via raw disk interface

Common types of disk caching

- ***General block caching*** - for popular files that are read frequently
    - Alternatively: for files that are written, then promptly re-read
    - Provides buffers for read-ahead, deferred write
- ***Special purpose caches***
    - Ex: Directory caches speed up searches of same directories, inode caches speed up reuses of the same file
    - More complex than general caches, but work better

Caching typically done via LRU (or similar strategies)

- In some cases - some file data is ***pinned*** in memory, temporarily not subject to cache replacement
    - Ex: inodes of files that have been opened by processes (ensures quick access in case the file is used again), contents of current working directories

Fast File System (FFS)

Improved upon old Unix file system

- Changed on-disk structures to video disk into ***cylinder groups*** - precursor to modern-day block groups
    - Can place two files into the same group to make seeks shorter, boosting performance
    - Within each group, FFS places needed info (structures tracking inodes, blocks, etc.) into the same group
        - Keeps copy of superblock for reliability reasons - can back up from replicas
        - Per-group inode bitmap, data bitmap tracks allocation of inodes, data blocks; inode, data block regions hold respective items
- Allocation policies via heuristics
    - Directory placement: balance directories across groups by placing directories in group with least number of allocated directories, enough free inodes

        - File placement: put data blocks of file in same group as inode
            - Try to place all files in same directory within the same group
- Has large-file exception for files that would fill their entire block group (under above placement criteria)
    - Don't want to prevent later related files from being placed into block group
    - FFS: first allocate some number of blocks into first block group, then place next "large" chunk of file (e.g. chunk pointed to by first indirect block) into a different or emptier block group; keep doing so for all chunks
        - Want to choose chunk size carefully - with large enough chunks, file system will spend more time transferring data and relatively less time seeking
- Combating internal fragmentation - FFS had sub-blocks, smaller "blocks" for allocating files (separate from larger blocks)
    - Could write small files to sub-blocks rather than full blocks to save space; allocated a full block later if file grows
    - Avoiding performance: rather than writing to sub-blocks individually, use buffered writes to only issue full-sized block writes

<u>**The ext2 File System**</u>

11.5: File System Naming

For each file, need some kind of handle/name for easy referral

- OS prefers simple numbers, but IDs aren't especially user-friendly
- Need a better way to name files - user-friendly, good for organization, easy to implement

File names and binding

- File systems identify files via descriptors; want more useful names for users
    - File systems must handle name-to-file mapping, ***binding*** a name to a file
        - Must be able to change names, find file corresponding to a name, etc.
- ***Name spaces*** - the total collection of all names known by some naming mechanism
    - In some cases - all names that could be created by the mechanism
    - Many possible structures, e.g.:

        - Flat name spaces - all names exist on a single level
        - Graph-based name spaces (usually directed)
            - Hierarchical name spaces, i.e. tree-based
    - Q: Do all files on the machine have to fall under the same name space?
        - Q: How many files can have the same name?
            - Flat name spaces - one per file system
            - Hierarchical - one per directory
        - Q: How many different names can a file have?
            - A single "true" name, but arbitrarily many aliases allowed
        - Q: Do different names have different characteristics?
            - Does deleting one name make others disappear?
            - Do all names see the same access permissions?

Hierarchical name spaces - a form of graph-based name space

- Typically organized via ***directories*** - files that contain references to other files
    - Directories represent a non-leaf node in the graph
    - Directories also used as naming context:
        - Each process has a current directory
        - File names are interpreted relative to that directory
- Nested directories can form a tree, expanding from a "root" node
    - File name describes a path through that tree
        - Name called "fully qualified"/absolute if it begins from root
        - Can recursively traverse tree to find file
    - May form a digraph instead, if files are allowed to have multiple names
- Directories are a <u>special type of file</u>
    - Used by OS to map file names into associated files
        - "Name.extension" naming commonly used for files, but specifying type is just convention (is never enforced by the OS)
    - A directory contains multiple directory entries
        - Each directory entry describes one file + its name
    - User applications allowed to read directories to find out what files exist + get information about each file
        - Typically, only OS allowed to write them (users need special system calls)

            - Important for file system integrity
    - Renaming via rename(): implemented as atomic call, updates readable name
- Traversing the directory tree
    - Some entries in directories point to child directories lower in the hierarchy
        - To name a file there - name the parent directory, then child directory, then file (with some kind of delimiter, e.g. '/', in between)
    - Directories usually have a special entry denoting parent directory, e.g. "../"
        - Unix: "./" is a directory entry that points to the directory itself
- File names vs path names
    - In some name space systems, files have "true names"
        - Only one possible name for a file, kept in a record somewhere
        - Ex. (DOS): each file described by a directory entry
            - Local name is specified in that directory entry; fully qualified name is the path to that directory entry
    - In other systems, files have no "intrinsic" name
        - Ex. (Unix): no single "true" file name, all names come from directory paths

Unix Naming

File descriptors are inodes - directory entries only point to inodes

- Multiple directory entries can point that the same inode -> a single file (descriptor) may have multiple names
- An association of a name with an inode is called a ***hard link***
    - Links only serve as names; all other metadata, protections, etc. stored in file inode
    - All links are equal, provide same access to the file; anyone with read access can create new link
        - Caveat: directories themselves can be protected
- Unix directory entry contents: name relative to current directory + pointer to inode of associated file
- A file exists as long as at least one name exists
    - File inode keep reference count of links
    - Removing a name is just removing a link; other names/links still work
        - unlink() syscall to remove a link

Symbolic/soft links - alternative way of giving files multiple names

- Implemented as <u>special type of file</u> - contains path name (indirect reference) to the linked file
    - File system recognizes symlinks, automatically opens associated file instead (or fails if not found)
    - Does not provide a reference to the inode
        - Don't prevent deletion, update link count; may have dangling reference if file is deleted

11.5 File System Reliability

Many possible issues: bad file system state from software crashes, data loss from hardware failure/software error, file system corruption, etc.

Core problem: System writes typically involve multiple operations (multiple hardware operations) that must all go through to succeed

- Need to write data blocks, metadata blocks, inode, free list, directory blocks, etc.
- Worst case, when using deferred writes- a process may allocate a new block to a file, but crash before deferred write to free list occurred; afterward, process make allocate that same block to a different file (since free list outdated), two files contain same block -> writes to one corrupt the other; leaves system state incomplete (***crash-consistency problem***)
    - Applications expect writes (done via system call) to be "safe" once the system call returns
        - Can block writing application to guarantee, but this may take a long time to do
            - Crashes (and resulting failures) are quite rare

Crash consistency problem may result in inconsistency in file system data structures, space leaks, garbage data, etc.

- Solutions 1: Let inconsistencies happen, fix them later
    - Fsck: Unix tool for finding, repairing file system structures

        - Can't fix garbage data in data blocks, e.g. , only ensures metadata (e.g. superblock, bitmaps, inodes, directories) are internally consistent
        - Runs before file system mounting
        - Issue: poor performance for large disks

Buffered writes

- Rather than waiting for a write to be persisted, keep track of it in RAM and return success to the application
    - Write to persistent memory at a later point
- Results in less application blocking and deeper queues, but may result in errors if a crash occurs between returning success & writing to persistent memory

Ordered writes - can use to reduce potential damage

- Write out data before writing pointers to it
    - Unreferenced objects can be garbage collected (in the event of a crash), but pointers to incorrect info are more serious
- Write out deallocations before allocations
    - Want to disassociate resources from old files quickly; can update free list later, via garbage collection
        - Improperly shared data more serious than missing data
- Issues:
    - Greatly reduces I/O performance - can't accumulate nearby operations, consolidate updates to the same block
    - May not be possible - some modern devices may reorder queued requests
    - Only reduces severity of incomplete writes, but doesn't fix them

Use copy-on-write

- Rather than overwriting data: write to unused locations, then later update pointers to point to new structures

Use backpointer-based consistency: to every block, add a back pointer (e.g. every data block has reference to associated inode)

- Can check that inode forward pointer goes to a block with that inode as its back pointer
- Use to verify consistency, not to fix it

Optimistic crash consistency - use checksums + other techniques to detect inconsistencies

- Can use to make writes faster, improving performance

Can use auditing/repairing

- Can design file system structures to be able to be audit/repair
    - Have redundant information in multiple distinct places
        - Audit for correctness, use redundant info for automatic repair
- Once standard practice, but no longer practical - extremely slow to audit very large file systems

Journaling

Can create a circular buffer journaling/write-ahead logging device:

- Journal writes always sequential
- Journal writes can be batched
- Journal relatively small; may use NVRAM

Can journal all intended file system updates: inode updates, block write/alloc/free

- Ex: If replacing two existing pages, can put a record in the journal with ***transaction identifier/TID*** + metadata about where to put blocks
    - Also write the two actual blocks: either via ***physical logging*** (write copies of the blocks) or ***logical logging*** (via references to blocks, e.g.)
    - Have delimiters to distinguish start/end of request
- After creating entry: First return true to the process, then copy data pages to true locations at a later point (***checkpoint***) and delete entry
- Issue: may have crashes while the update is being written to the journal
    - To fix: only write transaction end/commit block once all previous writes complete
        - Disks guarantee atomicity of writes for small-enough blocks (e.g. 512 KB); want to fix transaction end/TxE section into size
- Actual file system updates can be efficiently scheduled (via writeback cache, batching, motion-scheduling, etc.)
    - Journal completions when real writes happen
- Batched journal entries

    - Operation is safe after journal entry persists (caller must wait for this to occur)
    - May accumulate batch until full/max wait time for efficiency

Journal recovery

- Journal small circular buffer -> can be recycled after old operations completed
    - Use timestamps to distinguish new vs old (checkpointed vs non-checkpointed)
    - Relatively efficient to review
    - For each journal: have superblock recording which transactions have and have not been checkpointed
        - Mark completed transactions as "free" in journal superblock
- ***Redo logging***: After system restart, review entire journal and note which ops have been completed
    - Replay (in order) all writes not known to have completed - use data, destination stored in journal
        - Skip transactions not fully written to log (i.e. without end block)
    - Remove completed operations from journal, resume normal operation

Journaling

- Journal writes much faster than data writes
    - Journal writes are all sequential
- Normal operation - journal is write-only, file system never reads/processes journal
    - On restart, can scan the journal quickly
        - Journal is small + can be efficiently read sequentially
        - All recovery processing done in memory
- Journal pages may contain info for multiple files, by multiple processes/users
- Can buffer journal writes, send in batch (e.g. if two writes are to the same directory, bitmaps, etc.) for efficiency

In some cases, may only choose to journal metadata (***ordered/metadata journaling***) instead of journaling everything (i.e. ***data journaling***)

- Metadata is small and random (I/O inefficient), but critical for integrity
    - By contrast: data is large and sequential (I/O efficient), less order-sensitive; would take up most of journal capacity

- Safe metadata journaling
    - First allocate space for & write data; then journal metadata updates
        - Avoids case where metadata is written, but actual data values are garbage
- More natural for flash

Potential issue: may journal multiple updates to a block (if block is reused) -> if metadata journaling, block may end up with outdated contents

- Can either never reuse blocks until delete is checkpointed, or add revoke record
    - System scans for revoke records when replaying, doesn't replay any revoked data

Log-structured File Systems

Log-structured file systems (LFS): The journal acts as the file system, i.e.: the entire file system is a circular buffer and writes are performed sequentially on top of previous write

- All inodes, data updates written to the log
- Updates are ***redirect-on-write/copy-on-write*** - rather than overwriting old data, write the new data elsewhere and change the metadata (inode) pointer to point to it
    - In-memory index/map caches inode locations
    - Buffer writes for performance - make writes to an in-memory ***segment*** (buffer), then write the segment to disk when it fills
    - Also called ***shadow paging***
- Important architecture for flash file systems, key/value stores
    - Issues: long recovery time to reconstruct index/cache, need to implement log defragmentation and garbage collection

Navigating a logging file system

- Inodes point at data segments in the log
    - Sequential writes may be contiguous; random updates can be spread all over the log
    - Updated inodes added to end of log
- ***Index/inode map*** (***imap***) points to latest version of each inode
    - Given an inode number, produces disk address

        - Index (i.e. chunks of inode map) periodically appended to log - data blocks, inodes, and inode maps are all together in the buffer on disk
- ***Checkpoint region*** - fixed place on disk, containers pointers to the most recent pieces of the inode map
- Directories act similarly to Unix; map names to inode numbers
    - Reference inode map to translate inode numbers to addresses
- Recovery: find and recover latest index, then replay all log updates since (***roll forward***)
    - Ensuring checkpoint region updated atomically: LFS keeps two CRs at opposite ends of disk, writes to them alternately
        - Order of update: first write out header w/ timestamp, then CR body, then last timestamped block
            - If system crashes during CR update, can detect inconsistent timestamps and use most recent CR with consistent timestamps
- Since nothing is ever overwritten, need garbage collection to free space
    - After finding several partially-used segments: can write out new set of segments containing only live/active blocks, then free the old segments for writing
        - Determining block liveness - to each segment, for each block D, include inode number and offset in ***segment summary block*** at segment head
        - Determing which blocks are clean - nontrivial

Redirect-on-write

- Done by many modern file systems - ensures that blocks, inodes are immutable once written
    - Immutable property beneficial for flash devices
- Simply need to add new info to the log, update index
    - Old inodes, data remain in log; can still be accessed if an old inex is available
        - Clones and snapshots are almost free
- Cost: requires management, garbage collection
    - Must inventory, manage old versions; eventually, need to recycle old log entries as well

11.6 Flash Memory

Solid-state storage - storage devices with no mechanical/moving parts, only transistors (e.g. memory, processors)

- Unlike typical RAM, solid-state storage devices (SSDs) retain information even if power is lost; alternative to hard disks for persistent storage

(***NAND-based***) ***flash*** - special form of memory

- To write a chunk (i.e. ***flash page***), first need to erase a bigger chunk (***flash block***); potentially expensive
    - Writing too often to a page will cause it to ***wear out***
- Flash chips store 1/2/3 bits in a single transistor; maps level of charge to binary value(s)
    - Single-level cell (SLC) vs multi-level (MLC); SLC typically higher-performing, pricier
- Flash chips accessed in two units: larger (***erase***) ***blocks*** (e.g. 128, 256 KB) and smaller ***pages*** (e.g. 4KB)

Flash operations

- Read: Random-access, very fast
- Erase: to write a page to flash, need to first erase entire page's block
    - Erasing sets all bits to 1; computationally expensive, destroys all contents
    - Block is reset and each page can be programmed after erase complete
- Program: Program command can change some 1s to 0s within page
    - Less expensive than erasing, more expensive than reading

Flash Performance and Reliability

- Lower risk of mechanical/physical damage
- Primary concern is ***wear out*** - as flash blocks are continuously erased and programmed, accumulate small amounts over time; make differentiating 0, 1s more difficult (and eventually impossible)
    - Want ***wear leveling*** - spreading writes across flash blocks more evenly
- Also have ***disturbance*** - when accessing a page in flash, possible to accidentally have bits flipped in neighboring pages (***read/program disturbs***)

- Flash SSDs composed of a number of flash chips for persistent storage
    - May also have some amount of volatile/non-persistent memory; for caching and buffering, e.g.
    - Also contains control logic for handling device operations
        - Flash translation layer/FTL: converts read/write requests on logical blocks (device interface) into low-level read/erase/program commands on underlying physical blocks and physical pages
            - Can utilize multiple flash chips in parallel for performance
- Want to reduce ***write amplification*** - ratio of total write traffic issued to flash chips by FTL compared to total write traffic issued by client to SSD

Bad approach - ***direct mapping*** maps each logical page N to the same physical page N

- Bad for performance - any write to logical page N must always result in an erase and rewrite of the flash block -> write amplification equal to num pages/block
- If workload not spread evenly across logical blocks, popular logical blocks will have corresponding physical blocks worn out quickly

Log-structured FTLs - most common modern approach

- Writes to logical blocks N are appended to next free spot in currently-being-written-to block (***logging***)
    - Device stores (***in-memory***) ***mapping table*** - maps logical blocks to physical addresses, update with new writes to the logical block
        - Stored in persistent memory somewhere
    - Blocks typically written in-order (low to high) to reduce disturbance issues
- Can pick and choose which block to write to -> can perform ***wear leveling***
- Issue: since log-structured, need to periodically perform garbage collection
    - Might result in write amplification if done poorly
    - Garbage collection: find ***dead blocks*** (containing 1 or more garbage pages), write non-garbage pages out to block, reclaim entire original block
        - Requires reading and rewriting -> want to select blocks that only contain dead pages; can erase immediately without needing to copy contents

    - Some SSDs overprovision device - increase capacity beyond listed size, allows SSD to delay cleaning and increase internal bandwidth for cleaning
- Issue: in-memory mapping tables may become very large with larger devices
    - Inefficient to have page-level mapping (mapping logical pages to physical 1:1)
        - Increasing page size decreases memory table size, but larger pages -> larger erases needed (more write amplification)
    - Can due ***block-level mapping*** - translate logical block addresses (containing chunk number + offset)
        - Chunk number maps to physical block (i.e. starting page # of physical block)
        - FTL then computes address of page by adding offset from logical address to physical address of block
        - Issue: may still have issues with performance when writes are smaller than physical block size with needing to erase
            - When a logical block is updated, need to copy over all physical blocks surrounding the modified physical block

Hybrid mapping - FTL keeps a few blocks (***log blocks***) erased, directs all writes to them

- FTL keeps two types of mapping table:
    - Keeps per-page mappings for log blocks in ***log table***
    - Keeps per-block mappings in ***data table***
- Finding a logical block: FTL first checks log table, if address not found, then checks data table
- Want to keep number of log blocks small:
    - FTL periodically looks at log blocks, switches into block-mapped blocks (mapping per-block instead of per-table); method based on block contents
        - Assume four physical pages in a log block correspond to four adjacent logical pages
            - Best-case: If all four physical pages overwritten, can copy contents to new block with just a singular block pointer & erase old block, use as new log block (***switch merge***)
            - Worse-case: if only some physical pages overwritten, need to copy over non-modified pages to new block as well (***partial merge***)
                - Same state as switch merge, but one more operation

        - Worst-case: FTL may need to pull together pages from many other blocks (***full merge***)
            - If log block contains pages from multiple different logical pages: to switch to block-mapped page, need to first create new data blocks for each
                - Ex: contains 0, 4, 8, 12 -> need to find 1/2/3 and create new block for 0/1/2/3, block-mapped; similar for 4, 8, 12
- Performance optimizations
    - Can cache only active parts of FTL in memory
        - Issue: if ***working set*** size exceeds cache, need to evict old mapping (potentially needing to write it to flash, if it hasn't been persisted) and perform extra read to bring in new mappings
- Wear leveling
    - If blocks are filled with long-lived never-overwritten data, will never be reclaimed by garbage collection -> doesn't receive any write load
        - To fix: FTL must periodically read all live data out of such blocks, rewrite elsewhere; increases write amplification, decreases performance
- SSD performance and cost
    - Can perform I/O much faster than traditional hard disk drives, but more expensive

<u>Disk Failures</u>

Early disks fail-stop (either the entire disk works, or the entire disk fails); modern disks fail-partial (disks can fail while still appearing to work)

Two main types of single-block failures in modern disks:

1. ***Latent sector errors*** (***LSEs***)
2. ***Block corruption***

LSEs occur when a disk sector(s) has been damaged

- May be due to disk head touching surface (head crash), cosmic rays flipping bits, etc.
- Can use in-disk ***error correcting codes*** (**ECC**) to determine whether on-disk bits are good, or fix (in certain cases; otherwise, return error)

        - Storage systems may have redundancy mechanisms if a block access fails (e.g. data mirroring)
            - RAID systems may try to reconstruct a disk after failure by reading through disks in parity group(s), recomputing missing values
                - May fail if a different disk also encounters an LSE

Block corruption: Disk blocks may become corrupt in non-detectable manners (silent faults)

- Ex: buggy firmware may write a block to the wrong location, or a faulty bus may corrupt transmitted data
- Modern systems primarily use ***checksums*** to detect corruption
    - Functions take a chunk of data, compute smaller checksum as summary
        - Common functions: XOR, Fletcher, cyclic reundancy check
    - If a recomputed checksum on data is different than previous, data has been altered or corrupted
        - Can store a checksum for each disk sector/block (either within its own block, or all stored in a separate sector/block)
        - When reading blocks, compute checksum over retrieved block and compare to retrieved checksum

Another problem: ***misdirected writes*** occur when disk/RAID controllers write the correct data to the wrong location in disk

- Solve by adding a **physical identifier** (**physical ID**) to each checksum - disk and sector numbers of the block
    - When reading, compare retrieved ID with expected values to detect corruption

Final problem: ***lost writes*** occur when a write request is signaled as completed, but the write is never persisted (saved to disk)

- Checksums/physical IDs don't work
- Can perform write verify/read-after-write - upon writing data, immediately read back to double check that it was saved (inefficient)
    - Alternatively: can add a checksum in each file system inode, indirect block for every block included in a file
        - Can compare checksums of inode, data block

When to check checksums - can periodically read through every system block to verify checksums are still valid (***disk scrubbing***)

Checksum requires space and time overhead

- Space overheads small:
    - Small checksum on each disk sector + room in memory for retrieved checksums while accessing data
- Time overhead is significant
    - Can combine data copying and checksumming into a single activity, since copy operation is done by both
    - Can tune background scrubbing

Sec. 12: Virtual Machines

A ***virtual machine*** (**VM**) is software meant to appear (to apps and users) to be a real machine

- Uses real hardware to implement virtual hardware:
    - Instructions for virtual CPU run on real CPU
    - Real RAM stores data for virtual RAM
    - Real disk stores data for virtual disk
    - Etc.
- Virtual server runs on virtual hardware components
- Benefits: fault isolation, better security, better control over hardware sharing
    - Fault isolation: crashing a VM's OS (or similar faults) only takes down the VM, not the hosting machine
    - Security: virtual machines don't need to see real shared resources (e.g. file system, IPC channels) -> harder to interfere with other processes/VMs using them
    - Can use to run a different OS (e.g. Linux on Windows)
    - Sharing resources: generally difficult for an OS to guarantee a particular allocation of resources between processes; important for cloud computing, e.g.
        - Instead, can give a set allocation to an entire VM -> processes in the VM won't be able to steal from other processes outside the VM

12.1 Running VMs

Easiest if virtual, real machine use same ISA (common case); harder and slower otherwise

Rely on limited direct execution - run as much VM activity directly on CPU as possible

- When necessary, trap from VM to VMM
    - Necessary when VM needs to perform a privileged operation
    - Initial syscall will trap to VMM; will usually first forward it to VM's OS, but subsequent calls trap back to VMM
- VMM runs in privileged mode; VM runs in non-privileged mode
    - Historic architecture: VM runs in privileged mode
    - Each VM has its own OS kernel, drivers (running in non-privileged mode)

        - Within VM: distinguish between VM privileged mode (VM OS) and VM non-privileged mode (VM apps)

Hypervisor*/*virtual machine monitor (**VMM**) - controller that handles all virtual machines running on a real machine

- VM traps go to VMM; VMM handles trap, returns to LDE (similar to process for a system call to an OS)
    - More overhead than regular LDE
- Handling traps
    - System calls/privileged instructions from any applications inside VM (that are not performable by VM OS) are first sent to VMM handler
    - VMM can't typically perform syscall correctly (no knowledge of VM OS's internal state; may not offer that syscall at all), but can refer to VM OS's trap table and invoke VM OS's code to handle
        - VM OS tries to install handlers during startup (privileged) -> traps to VMM, VMM records locations
    - VM OS's code runs; when it tries to use a privileged instruction, VM OS traps; VMM catches trap, runs instruction, returns to VM OS
- VMM running all privileged instructions means VMM has control
    - Can refuse to perform certain instructions, e.g.
- Running multiple VMs
    - VMM needs to perform machine switch between VMs - needs to save entire machine state of one OS (including registers, PC; unlike context switches, privileged hardware state), restore machine state of the other, then jump to PC of to-be-run VM

Virtualizing memory in VMs

- VM OS needs a way to provide segregated virtual memories to its applications, but VMM controls CPU registers that point to the needed page tables
    - Issue: VMM doesn't know anything about the page tables the VM OS is associating with each application
- Solution: virtual OS thinks it has physical memory addresses; provides virtual addresses to its processes, handles translation from virtual addresses to its "physical" addresses

    - VMM has machine addresses - translates to physical addresses within a single VM
        - Still uses the same paging hardware
    - Three address types - machine addresses (real RAM addresses), “physical” and “virtual” addresses (not RAM addresses; VM-managed)
    - Ex:
        - VM app issues virtual address
        - Virtual address causes TLB miss, trapping to VMM
        - VMM catches trap, invokes OS A
        - OS A looks up virtual address in app’s page table, tries to install physical page number for app in TLB; traps to VMM
        - VMM installs correct machine address in TLB, returns to VM app
- Implications:
    - TLB misses much more expensive in VMs - incurs overhead from swapping between unprivileged/privileged mode, running extra system code
        - Extra paging data structures in VMM -> even more overhead
        - Consequence: VMs likely to suffer performance penalties

Improving VM performance

- Via special hardware: some CPUs have features to make virtualizing CPU, memory cheaper
- Some VMMs can use inference
    - Ex: can detect VM idle loop by noticing low-power mode
- Via ***paravirtualization*** - modifying guest OSes in VMs to make them more suitable to virtualization
    - Basic VM approach - guest OSes don’t know anything about virtualization; can make changes to make virtualization cheaper

VMs and cloud computing

- Cloud computing involves sharing hardware among multiple customers
    - Cloud provider typically sells/rents computing services, handles difficult issues of allocating resources; customer needs only run applications
- Cloud providers often operate on large scales -> need lots of hardware, shared across many customers at a time (potentially multiple computers/customer)

    - Also need to guarantee isolation between customers
- VMs - cloud providers want to make efficient use of hardware
    - Often, a customer may not need full power of a machine; but customer will always need isolation -> can utilize virtual machines
- Can run everyone in a VM (or multiple VMs for big jobs)
    - VMs may share physical machines, but will still be isolated
        - Customer work loads may fluctuate
- Want most efficient packing of VMs onto physical machines as possible
    - Many physical nodes, many more VMs - typically reduces to a bin-packing algorithm (NP-hard), use estimation techniques

Other benefits of VMs

- Allow easy experimentation that may not be easy to do on real hardware
- Allow easy division of resources for servers
- Allow greater flexibility in types of software a computer can run

Sec. 13: Operating System Security

OSes are the lowest user-visible layer of software, often have partial or complete access to the hardware -> to protect the machine, need to protect the OS

- OS controls access to application memory, processor scheduling, responding to application requests
  - Flaws in OS will generally compromise security at all higher levels - nearly all security systems have to assume the OS itself is secure

Important definitions:

1. ***Security***: Policies regarding aspects of the OS (e.g. "file is read-only")
2. ***Protection***: The mechanism(s) implementing security policies
3. ***Vulnerabilities***: Weaknesses that can allow attackers to cause problems
   a. Not all vulnerabilities cause all problems
4. ***Exploits***: Incidents where a person takes advantage of a vulnerability
   a. Also refers to code/methodology used to take advantage
5. ***Trust***: Determines how different parties are treated
   a. Almost always have to trust the OS
6. ***Authentication***: Determining which party is asking to perform a given action
   a. ***Credentials***: Any kind of OS-managed data for access control/authentication
7. ***Authorization***: Checking whether that party should be allowed to do it

Principles of system design (for systems with security requirements):

1. Economy of mechanism: Keep the system as small and simple as possible; ensures understandability of code, reduces bugs
2. Fail-safe defaults: Default to security, not insecurity; any policies that can change system behavior should have defaults set to be more secure
3. Complete mediation: Every time an action to be performed is taken, the system should always check to ensure the action meets security policies
4. Open design: Assume any adversary knows every detail of the design when attempting to meet security goals
5. Separation of privilege: Require separate parties or credentials (e.g. 2-factor) to perform important/critical actions

6. Least privilege: Give a user/process the minimum privileges needed to perform the intended actions
7. Least common mechanism: For different users/processes, use separate data structures or mechanisms to handle them; ex: separate page tables for each process in memory ensures neither can interfere with the other
8. Acceptability: A system is only good if users will use it

13.1 Access Control Mechanisms

Authentication

- Primarily done on computers via user ID; all processes run by user inherit their ID
- May determine the identity of a user via password, e.g.
    - Issues: Susceptible to password sniffers, vulnerable to brute force attacks, generally outdated (but widely used)
    - To prevent system-stored passwords from leaking, can store password as a hash (so system doesn't have to store plaintext password)
        - Can associate a random number (***salt***) with each user; concatenate password, salt before hashing
            - Since cryptographic hash algorithms known, using a salt can prevent an attacker with alg + password hash from guessing
- Other approaches:
    - Challenge/response systems (e.g. text challenges to an external device, secret functions for smart cards)
        - Less complex forms still susceptible to network sniffing, brute force
    - Can have physical/external security tokens (i.e. a dongle)
    - Biometric authentication - based on some physical attribute of the user (e.g. fingerprints), compared against a stored value
        - Issues: requires very special hardware, characteristics may vary day-to-day, complex to do properly across a network
            - May incur false positives/false negatives
            - Remote authentication - biometric reading just a bit pattern, can be easily copied if found by an attacker

            - Need high confidence in security of path between biometric reader, checking device; no Internet, e.g.
        - Sensitivity (how close a match must be to admit) affects error rate
    - Multi-factor authentication - using multiple separate authentication methods (current preferred approach)

Access control - OS mechanisms for enforcing security policies on who can access what

- ***Access control lists*** (**ACLs**) - for each protected object, maintain a single OS-managed list specifying who can access the object (and in what mode of access)
    - Can check access control list whenever something requests access
    - Unix: Per-file ACLs embedded into inode
        - Three subjects on list for each file - owner, group, other
            - Group ID stored in file inode
        - Three modes - read, write, execute
    - Advantages: easy to determine who can access, change permissions, performant
    - Disadvantages - difficult to determine all objects a subject can access, changing access rights requires getting to the object
        - In distributed systems, need all identities to be consistent to work
- ***Capabilities*** - Each entity keeps a set of data items ("tickets") specifying allowable accesses
    - To access an object, entity needs to present the proper capability
        - Essentially a data structure/collection of bits; since having one grants access, need to make them unforgeable
        - May store, manage in OS rather than giving users/processes access
        - For each process: store a capability list in kernel memory, place pointer to list in process' PCB
    - Advantages: Easy to determine what objects a subject can access, potentially faster than ACLs, easy model for transferring privileges
        - ACLs: child processes inherit identity of parent process (incl. privileges), but can differ capabilities between child and parent; parent can simply not transfer certain privileges
    - Disadvantages: Hard to determine who can access an object, needs extra mechanism for revocation
        - May need cryptographic methods to prevent forgery over a network

- OSes often use both ACLs, capabilities
    - Unix/Linux: ACLs for initial file opens; file open creates a file descriptor with certain access rights, acting as a capability

Enforcing access in an OS

- Processes have associated user IDs for access decisions
- Protected resources must be inaccessible - use hardware protection to ensure only the OS can make them accessible to a process
    - Make processes issue request/system call to the OS, allowing OS to consult access control policy data before authorizing
    - OS may grant access directly (resource manager maps resource into process) or indirectly (resource manager returns a capability to process)

Discretionary (owning user can control all access permissions) vs mandatory (owner cannot control certain elements of access control; are set by an authority) access control
Role-based access control (RBAC), type enforcement

Enforcing access to files

- Unix: for each file, permission bits lay out permitted operations (out of read/write/execute) for three classes of user (owner, group, and other/anyone)
    - Group specified via some group name
    - chmod() to change file mode/permissions
- Some file systems (e.g. AFS) have ACLs per directory, can set access permissions

13.2 Cryptography

Cryptography - transforming bit patterns in controlled ways to obtain security advantages

- Intended to make only authorized parties able to read encoded secrets
- Terminology:
    - Cryptography usually described in terms of sending a message between sender S, receiver R
    - ***Encryption*** - the process of making the message unreadable/unalterable by anyone but the receiver

    - ***Decryption*** - the process of making the encrypted message readable by R
    - ***Cryptosystem*** - the system that performs the encryption/decryption transformations
        - May follow set of rules for transformation (called a ***cipher***)
    - ***Plaintext*** (original form of a message, labeled P) vs ***ciphertext*** (encrypted form of the message, labeled C)
        - Encryption algorithm referred to as E: C = E(K, P)
        - Decryption algorithm referred to as D
- Most cryptographic algorithms use a ***key*** K to perform encryption/decryption
    - Key acts as a secret - decryption easy if and only if you have the key
    - Reduces secrecy problem from needing to protect message, to only needing to protect the shorter key
    - Both the encryption, decryption algorithms utilize the key
        - Combination of encryption, decryption algorithms called a cryptosystem

Cryptosystems:

- ***Symmetric cryptosystems***: P = D(K, C) = D(K, E(K, P))
    - E, D not necessarily the same algorithm; but only one key K
    - Advantages: encryption/authentication done via single operation, can be faster than asymmetric systems if cryptosystem is well-known/trusted, no centralized authority (e.g. key server) required
    - Disadvantages: hard to separate encryption from authentication, non-repudiation difficult without servers, key distribution can be a problem
        - Non-repudiation - the ability of the author to prove that a message was sent by them
    - Popular symmetric ciphers:
        - ***Data Encryption Standard*** (**DES**) - old US standard
        - ***Advanced Encryption Standard*** (**AES**) - current US standard, most widely used cipher
        - Others (e.g. Blowfish)
    - If symmetric cipher is flawless, can only be cracked by brute force (trying every key until one works)

        - Cost of brute force depends on key length - for N possible keys, attacks need to try an average of N/2 keys
        - DES uses 56-bit keys (brute-forceable by modern systems); AES uses 128/256-bit keys (long enough)
- ***Asymmetric cryptosystems/public key cryptography*** (**PK**): Encryption and decryption use different keys
    - Have separate keys K_E, K_D: C = E(K_E, P), P = D(K_D, C) = D(K_D, E(K_E, P))
        - Sometimes works the other way around as well: P=D(K_D, E(K_E, P))
    - Keys created in pairs; one kept secret by the owner, the other is made public to the world
        - To send encrypted message, use public key to encrypt; only owner has private key to decrypt
        - Authentication - to sign a message, can encrypt with private key
            - Only owner knows private key; receivers can verify identity using public key
            - Much better than in an symmetric cryptosystem - receiver could not have created message, only sender could have
    - Issues with key distribution:
        - Security depends on using the right public key
            - If a message is sent using the wrong public key, then the owner of the incorrect key can now read the message
        - Consequently: need high assurance a given key belongs to a particular person
            - Via either ***key distribution infrastructure*** or ***certificates***
            - Both problematic at large scale/in real world
    - PK algorithms typically based on a mathematics problem (e.g. factoring extremely large numbers)
        - Security based less on brute force, more on complexity of underlying problem
            - If problem is solved, then keys are no longer secure
        - Choosing key pairs may be complex and expensive
    - Example ciphers:
        - ***RSA*** - most popular public key algorithm; based on factoring large numbers

            - Issue: size of what constitutes a “secure” RSA key keeps increasing with better integer factoring capabilities
                - Longer keys are more expensive to encrypt/decrypt
        - ***Elliptic curve cryptography*** - better-performing, but less-studied alternative to RSA

Common approach - combine symmetric, asymmetric cryptography

- Use RSA/other PK algorithms to authenticate, establish session key; use DES/AES with session key for rest of transmission
    - Utilizes asymmetric cryptography to “bootstrap” symmetric crypto

Cryptography and OSes

- ***At-rest data encryption***: Can encrypt stored data to prevent external access to hardware (without going through OS)
    - ***Full disk encryption*** encrypts all of (or nearly all of) the storage device
        - System requests decryption key, or credentials for decryption key, at boot time
        - Data decrypted as it moves out of disk; remains decrypted in memory (while in buffers, e.g.); encrypted when stored
    - Comes with a performance penalty
    - Encrypting select data at rest - via password vault/key ring
        - Encrypt all different passwords, store on machine (indexed by site); decrypt and provide where needed

Distributed Systems

Sec. 14: Distributed Systems

Distributed Systems Overview

Distributed systems (coordinating multiple different computers to run a single task/tasks) - how most modern computing is done

- Advantages:
    - Better scalability and performance; can be used for apps with large resource requirements (>1 computer)
    - Improved reliability and availability - a single computer failure isn't catastrophic, e.g.
    - Ease of use - easy to have centralized management of all services and systems with distributed systems
- Potential issues:
    - Different machines don't share memory/devices -> machines can't easily determine other machines' states -> risk of synchronization issues
    - Only way to act remotely is via network -> asynchronous, slow, error-prone, and not under the control of any single machine
    - Failures of one machine aren't immediately visible to other machines

Communication basics

- Modern networking - communication is fundamentally unreliable
    - Packets may be lost or corrupted, e.g.; or due to a lack of buffering within a network switch, router, endpoint, etc. (packet drops)
- Some applications know how to deal with packet loss; can let them communicate with a basic unreliable messaging layer (an ***end-to-end argument***)
    - Ex: UDP/IP networking stack (unreliable); a process can use sockets API to create communication endpoint, receive UDP datagrams
        - UDP relatively unreliable, but has some features (e.g. checksums for corruption)
- Can try and build a more reliable communication layer

    - Can use ***acknowledgments*** - receivers send short messages back to acknowledge message receipts
    - ***Timeout***: Sender can set a timer to go off after some period of time; assume message was lost if no acknowledgment received in that time
        - Can retry a send afterward
    - Issue: what if the acknowledgement was the message lost?
        - In some cases: retry send may be interpreted as a separate message; want to ensure messages sent exactly once
        - Can use ***sequence counter*** - both sender, receiver agree upon start value for a counter (e.g. 1); both sides maintain a counter
            - Messages sent by sender are numbered N, N+1, etc.; current value of counter sent along with message
                - Receiver always acknowledges, but may or may not pass message up to application if it has already been received
    - Common protocol: TCP/IP

Often want distributed systems to be ***transparent*** - resembling a single machine system as much as possible

- Issue: there is no true transparency (Deutsch's Seven Fallacies)
    - Network may be unreliable
    - Latency - response times may not be instant
    - Available bandwidth is finite
    - Network may not be secure
    - Topology of network may change
    - Network (or portions of the network) may have multiple administrators
    - Transporting additional data often incurs additional costs

Distributed system paradigms

1. ***Parallel processing***, relying on tightly-coupled hardware
    a. Not widely used; requires special hardware
2. ***Single-system images*** make all nodes look like one big computer
    a. Difficult to implement, also not widely used
3. ***Loosely-coupled systems*** - the typical modern approach to distributed systems

4. ***Cloud computing*** - a more recent variant

Loosely-coupled systems consist of a parallel group of independent computers connected via high-speed LAN

- Typically service similar, but independent, requests; little coordination or cooperation required between computers
- Ex: web servers, app servers
    - Scalable and cost-efficient; easy to manage and reconfigure
    - High availability if protocol permits stateless servers

Scalability

- ***Horizontal scalability***: since each node is largely independent; can simply add a node in order to add capacity
    - Can be limited by network/load balancer instead of hardware or algorithms
    - Advantages
        - Individual servers relatively inexpensive -> good scalability
        - Very reliable - a node failing reduces capacity, but doesn't severely impact the broader system; high availability
            - Frontend switch can automatically bypass failed servers
            - Stateless servers and client retries can fail-over (switch to a redundant server in event of a failure) easily
    - Challenges - need to manage many, many servers
        - Automated installation, global configuration servers
        - Self-monitoring, self-healing systems
        - Scaling often limited by management, not hardware/algorithms

Elements of loosely-coupled architecture

1. Have a farm of independent servers handling computational tasks
    a. Servers run the same software, but serve different requests
    b. May share a common backend database
2. A *front-end switch* distributes incoming requests among available servers
    a. Can do both load balancing and fail-over
3. Service protocol

    a. Stateless servers and idempotent operations
        i. Idempotent - will always return the same result no matter how many times it is run
    b. Successive requests may be sent to different servers

Managed restarts

- ***Non-disruptive rolling upgrades***: For systems capable of operating without some of its nodes, can achieve non-disruptive rolling upgrades by upgrading nodes one-at-a-time
    - Requires upward compatibility between old, new versions of software for integration of updated and non-updated nodes
    - Often have an automatic fallback option to return to previous (working) release if rolling upgrade fails
- ***Prophylactic reboots***: Performance often degrades in software systems the longer they run; to fix, can automatically restart at regular intervals (e.g. a few hours/days)
    - Performance degradation may be due to memory leaks, e.g.; easier to restart than to fix all bugs
    - Via similar method to rolling upgrades

Cloud Computing

Cloud computing - recent twist on distributed computing

- Runs tasks in a cloud, rather than on local computers
- Often involves using special tools that support parallel/distributed processing
    - Goal: Want to abstract away most distributed systems technicalities, management away from the user
- Excellent for horizontal scaling - as loads change, can add/release nodes on the cloud dynamically to match
    - No need to buy additional machines, administer physical machines

MapReduce - a method for dividing large problems into compartmentalized pieces

- Principle: if there is a single function that a user wants to perform on a lot of data, can divide the data into disjoint pieces, perform the function on each piece (*map*), then combine the results to obtain output (*reduce*)

- Can perform each piece on a separate node for efficiency, combine results at the end on a final reduce node
    - Can even perform MapReduce on the task of combining results
- Most common cloud computing software tool/technique
- Synchronization in MapReduce
    - Each map node produces an output file atomically for each reduce node, but reduce node can't work until whole file is written
        - Creates synchronization point between map, reduce phases

Distributed Programs and RPC

Remote procedure calls (**RPC**) - one way to build a distributed program

- Built on remote procedure calls as a fundamental paradigm - clients, servers interact predominantly via procedure calls
    - Procedure calls acts as message sends/receives
    - "Remote" - procedure calls are as usual, but the subroutine may ultimately be executed on a different computer (i.e. address space) than the caller
- Advantage: procedure calls are already a primary unit of computation in most languages
- Limitations
    - No implicit parameters/returns (e.g. global variables)
    - No call-by-reference parameters
    - Remote procedure calls are much slower than standard procedure calls
- Nowadays: have numerous higher-level abstractions for RPC

RPC system has two pieces: stub generator/protocol compiler, runtime library

- Stub generator automates packing of arguments, results into messages
    - Caller: Creates buffer, packs information (***marshaling/serialization***), sends to destination server, waits for reply, unpacks return code and other arguments (***unmarshaling/deserialization***), returns to caller
    - Server: Unpacks message (un/de), call into remote function, package results, send reply to caller)
- Runtime library handles a lot of heavy lifting

    - Has to handle how to locate a remote service (naming) - might be using hostnames/port numbers from Interneet protocols, e.g.
    - Has to decide transport-level protocol to use (e.g. UDP vs TCP)
        - Less reliable layers may be more performant, but require additional work from runtime library to provide reliability
    - Procedure calls with large arguments may not fit into a single packet
        - May need to be ***fragmented*** by server into smaller packets; reassembled by ***receiver*** into a larger whole; done either by network protocol or RPC runtime ittself

RPC concepts

1. *Interface specification* lays out methods, parameter types, return types
2. *External data representation* (*XDR*) describes (machine-independent) representations of data types (e.g. byte ordering)
    - May be optimized for clients/servers that are similar
3. *Client stub* provides client-side proxy for a method in the API
4. *Server stub/skeleton* details the server-side recipient for API calls

RPC features

- Client application only ever calls, receives results from local procedures
    - All RPC implementation done inside local procedures - client application does not see the RPC occurring
        - Client does not know message formats, worry about sends/timeouts/reads, know the XDR, etc.
    - All RPC functionality done automatically
        - Message sends/reads performed via RPC tools, using the interface specification as the guideline
- Most RPCs done synchronously, but some allow asynchronous invocation

RPC limitations

- Requires client/server binding model, live connection
- A single thread services requests one-at-a-time -> multiple requests will spin off numerous threads (one per request)

    - Can instead use thread pool - finite set of threads created at server start, incoming messages are dispatched to worker threads (by main thread) for concurrent execution
- Limited failure handling - client must arrange for timeout, recovery
    - May need acknowledgments, retries
- Limited consistency support - only between calling client and called server, but complications arise with multiple clients and servers

Distributed shared memory (DSM) - allows processes on different machines to share a large virtual address space; early way to build distributed systems

- Not used today - failures on one machine will spread to other machines, accesses to memory were potentially very expensive

Network Security

Security even harder in distributed systems than in single machines:

- Since network activities happen outside of the OS, the OS cannot guarantee privacy/integrity; can only choose to trust (or not trust) external server
- Harder to authenticate remotely than locally
- Physical network connections (e.g. wires) may be insecure or vulnerable
- Need to coordinate security across multiple machines
- Internet is an open (and potentially dangerous) network

Goals of network security:

1. Want to secure conversations, guarantee *privacy* and *integrity*:
    a. *Privacy*: Only the sender and receiver know what is said
    b. *Integrity*: No outside party can tamper with sent messages
2. Want to ensure *positive identification of both parties*
    a. Want to be able to authenticate identity of message sender, get assurance that messages are not replays/forgeries
    b. *Non-repudiation*: Want to guarantee that an ID-ed message can only have been sent by the intended sender
3. Want to ensure *availability*: network and other nodes must be reachable when needed

Elements of network security

- Cryptography provides protection in communication
  - Symmetric cryptography for protecting bulk transport of data
  - Public key cryptography primarily for authentication
  - *Cryptographic hashes* to detect message alterations
- *Digital signatures* and *public key certificates* help authenticate message senders
- Filtering technologies (e.g. firewalls) defend against bad traffic

Cryptographic hashes used for tamper detection

- Can use checksums to detect data corruption
  - Ex: add up all bytes in a block, send sum along with data
  - Recipient adds up bytes; if the two checksums agree, data is probably OK
  - Checksum algorithms relatively weak
- Cryptographic hashes are very strong checksums
  - Unique - two messages are very unlikely to produce the same hash; in particular, difficult to find two messages with the same hash
  - One way - cannot (feasibly) infer original input from output
  - Well-distributed - any change to the input changes the output
    - In particular: change to the output is unpredictable
- Using cryptographic hashes
  - Start with a message to protect
  - Compute cryptographic hash for that message (via SHA-3, e.g.)
  - Transmit hash securely
  - Recipient does same computation on received text
    - If both results agree, message is intact
    - Otherwise, message has been corrupted/altered
- Secure hash transport
  - Cryptographic hashes aren't keyed -> can be produced by anybody
  - Can encrypt the hash instead, or transmit via secure channel
    - Encrypting a hash is cheaper than encrypting the entire message, if the latter isn't needed for perfect secrecy

Secure socket layer (**SSL**) - a general solution for securing network communication

- Establishes secure link between two parties; built on top of existing socket IPC
    - Typically uses certificate-based authentication of servers to verify identity of a server to a client
    - Optionally uses certificate-based authentication of client to server (for non-repudiation, e.g.)
- Uses PK to distribute symmetric session key (new key for each new socket)
    - Rest of data transport via symmetric crypto
    - Offers safety of public-key, efficiency of symmetric
- ***Transport Layer Security*** (**TLS**, or **SSL/TLS**) - improved version of SSL using encrypted sockets; used over SSL in almost all modern-day systems

Digital signatures - can encrypt a message with a private key as a means of signature

- Principle: Only the owner of the private key could have encrypted the message + could not have been tampered with
    - Acts as a means of tamper detection
- Generally, don't want to encrypt everything with private key (slow for large messages)
    - Instead, may encrypt just the cryptographic hash with private key

Signed load modules provide a way to know when to "trust" a program via digital signature

- First designate a certification authority (e.g. OS manufacturer - Apple, Microsoft, etc.)
- Authority verifies reliability of software (via code review/testing), signs certified module with their private key
- Can verify signature with their public key to prove module has been certified + has not been tampered with

Important issue for public keys: no way to determine who owns the private key corresponding to any given public key

- Want to ensure the private key owner is the intended one
    - OS manufacturers typically include public key with OS to verify load modules/updates
- General case - certify authenticity of a public key via ***PK certificates***

    - ***PK certificate***: data structure containing an identity + matching public key (plus potentially other information), along with a digital signature of those items
        - Signature usually signed by a trusted authority (***Certificate Authorities/CAs***) with a known public key
    - Can use PK of signing authority to validate that the signature is correct + certificate is unaltered
    - Issue: relies on having a known, trusted signing authority
        - At some point, someone's key must be delivered *out-of-band* (i.e. via some other means), e.g. in a trusted program (web browser) or otherwise
            - Web browsers often contain public keys for many authorities

Replay attacks - even if a message has a certificate from a correct source, it may have been copied by an adversary at some point and retransmitted later (i.e. is not from the source itself)

Web cookies are certificates verifying client identities

HTTPS - cryptographically-protected version of HTTP
- Connects to SSL, then lets SSL handle all client-server interactions

Secure Shell (SSH)

Distributed Synchronization

Distributed synchronization is difficult - hard to ensure globally coherent views of the system, unlike single machines
- Spatial separation: Different processes run on different systems -> no shared memory for atomic instruction locks
    - May even be controlled by different operating systems
- Temporal separation: Difficult (or impossible) to achieve proper chronological ordering of spatially-separated events
- Independent modes of failure: Even if one machine dies, other machines may still continue working

Difficult to implement locking of remote resources in distributed systems

- Potential problems:
    - The node that holds the lock might fail -> lock never gets (manually) released
    - The node may release the lock, but the release message might get lost
    - The node that holds the resource may fail -> upon recovery, the system may or may not recognize it as having held the lock

Leases offer a more robust alternative to locks

- Are similar to locks, but only valid for a <u>limited period of time</u> - afterwards, lease cookie expires and new leases can be granted
    - Updates with stale cookies not permitted
- Leases obtained from resource manager, gives client exclusive update rights to a file
    - Lease cookie must be passed to server on update; can be released at the end of the critical section
- Handle a wide range of failures: losses/failures in process, client node, server node, network, etc.

Lock breaking and recovery

- Easy to revoke an expired lease
    - Lease cookie includes "good until" time (based on server's clock); considered stale anytime afterward
- Safe to issue a new lease, since old lease-holder can no longer access object
    - If object was left in a "bad" state, may need to restore to "good" state - roll back to state prior to aborted lease
        - Can implement all-or-none transactions
- Issue: even if the resource itself is rolled-back, other changes (potentially based on that resource) may have been made to other resources that can't so easily be rolled back
    - The leasing machine may have even communicated with other machines, e.g.
    - Consequence: no (practical) way to roll-back everything without very high performance costs; most systems only restore the leased resource

Distributed ***consensus algorithms*** provide a possible solution in the face of node, network failures

- Goal: Want to achieve simultaneous, unanimous agreement on resource states

    - Requires agreement, termination, validity, integrity
    - Impossible to achieve in bounded time in general case, but can be done in certain special cases/with relaxed requirements
- Consensus algorithms used sparingly
    - Tend to be complex, take a long time to converge
    - May use for the first, relatively simple step in a longer process (for electing a leader, e.g.)
        - Use consensus to elect a leader, then the leader decides all future decisions going forward (without consensus)
- Typical consensus algorithm (electing a leader):
    - Each interested member broadcasts a nomination
    - All parties evaluate proposals according to a <u>fixed and well known rule</u>
    - Each voter votes for the best proposal (after a certain period of time)
    - If a proposal has a majority, proposing member broadcasts claim that the question has been resolved
    - Each party that agrees with winner's claim acknowledges resolution; election ends when a quorum acknowledges the result
- Assumptions
    - Static set of possible participants, agreed-upon order
    - All messages delivered within T_m seconds, responses sent within T_p seconds of delivery
    - Note: last condition implies synchronous behavior; provides an upper bound for time spent
- Consensus algorithms in practice
    - Possible leaders try to take over; if they detect a better leader, they agree to its leadership
        - In the event of a timeout, go to next-best leader
    - Keep track of state information about whether you are electing a leader; only do real work once a leader has been agreed upon
- Within distributed systems
    - One node is the coordinator; expects a certain set of nodes to be active and participating
    - Coordinator asks all other nodes

    - If expected node doesn't answer (or answers negatively), start an election
        - If an unexpected node answers, start an election
- Results
    - Basic consensus algorithm works well if timeouts are effective (i.e. site really is down), no partitions

Issue (***Partitions***): Depending on the layout of the network (plus any potential network failures), it may be the case that certain groups of nodes can communicate with other nodes in the group, but not with nodes outside of it

- Causes various members of the system to have different views of state -> may lead to different behaviors in each part of the system
    - Can have multiple partitions, changing partitions, etc.
- Want to design networks to avoid partitions
    - Less likely to occur on LANs, or if there are redundant network paths between all participants

Accessing Remote Data

In many cases, want to access data on a different machine; how?

- Goals:
    - Transparency: remote files should be indistinguishable from local files as much as possible; all clients should see all files from anywhere
    - Performance: Want to maximize performance relative to local disk per-client, be unaffected by number of clients
    - Cost: less than local (per client) disk storage, zero administration needed
    - Want to maximize capacity, availability
- Key characteristics of remote data access solutions
    - APIs lay out how users, processes access remote data
        - Transparency - how closely does remote data mimic local data?
    - Performance, robustness, and synchronization
        - Is remote data as fast and reliable as local data?
        - Is access to remote data synchronized, like local data?
    - Architecture: How is the solution integrated into clients and servers?

    - Protocol and work partitioning: How do clients and servers cooperate?

For ***remote file access***, want to maximize transparency

- Transparency:
    - Want normal file system calls to work on remote files
    - Want to support file sharing by multiple clients
    - Also want to maximize performance, availability, etc.
- Typical remote file access - <u>exploits plug-in file system architecture</u>
    - Client-side file system acts as local proxy for operations; translates file operations into network requests
        - Remote file system makes calls to socket I/O, NIC network controller; NIC driver makes request to server
    - Server-side daemon receives and processes requests; translates into real file system operations
        - Server NIC driver forwards request to socket I/O, then remote file system server; server references its disk for operations
- Predominant model for client/server storage
- Advantages:
    - Very good application-level transparency
    - Very good functional encapsulation
    - Can support multi-client file sharing
    - Can potentially achieve good performance, robustness
- Disadvantages:
    - At least part of implementation must be in the OS itself
    - Client, server sides tend to be fairly complex

Security for remote file systems

- Major issues:
    - Privacy and integrity for data on the network
        - Solution: Encrypt all data sent over network
    - Authentication of remote users (various approaches)
    - Trustworthiness of remote sites (various approaches)
- Authentication approaches

- Anonymous access (self-explanatory)
- ***Peer-to-peer authentication***: all participating nodes are trusted peers
    - User authentication/authorization done client-side
        - All users are made known to all systems
        - All systems are trusted to enforce their own access control
    - Advantage: Simple to implement (ex: NFS)
        - Disadvantages: can't always trust all remote machines; universal user registry is not scalable
            - May not work in heterogeneous OS environment
- ***Server-authenticated***: client agent authenticates to each server
    - Client authentication used for entire session
        - Authorization is based on credentials produced by server
        - Ex: Login-based FTP, SCP, CIFS
    - Advantage: Simple to implement
        - Disadvantages: Universal user registry not scalable; may not work in heterogeneous OS environment
            - No automatic fail-over if server dies
- ***Domain authentication***: client, server both authenticate independently via a single independent authentication service
    - Authentication service may issue signed "tickets", guaranteeing user's identity and rights
        - Both client, service knows/trusts only the authentication service
        - Tickets may be revocable or timed-lease
    - Alternatively, may establish secure two-way session
    - Ex: Kerberos

- Two approaches to distributed authorization
    - Authentication service returns credentials; server checks against ACL
        - Advantage: auth service doesn't know about ACLs
    - Authentication service returns capabilities; server verifies via signature
        - Advantage: servers do not know about clients
    - Both approaches commonly used
        - Credentials - if subsequent authorization required
        - Capabilities - if access can be granted all-at-once

Reliability and availability in accessing remote files

- Reliability - assurance that service works properly, will not lose data
    - Challenging in distributed systems - need to contend with full/partial failures
    - Want to reduce probability of data loss
        - Typically via some form of redundancy: spreading data across multiple disks, copies on multiple servers, or backups (worst-case)
            - Want to ensure disk/server failures don't result in data loss
        - ***Data mirroring***: Replicating data in primary storage onto a separate form of secondary storage (or multiple)
            - Provides additional copy for redundancy + can result in performance gains (more choices of storage to index)
    - Want to be able to automatically recover after failure
        - Make sure remote copies of data will become available again; any redundancy losses due to failure will be made-up
- Availability - assurance that service will always be available when needed
    - Distributed systems must make sure that some system elements failing doesn't prevent data access
    - ***Fail-over*** - when a server fails, want to transfer work/requests to another server
        - Must detect failure of primary server, fail-over client to secondary
            - Data must be mirrored to secondary server
        - Session state must be reestablished: client authentication/credentials, session parameters (e.g. working directory, offset)
            - Easier if operations are idempotent
        - In-progress operations must be retransmitted
            - Clients must expect timeouts, retransmit requests
                - Are responsible for writes until server acknowledges
    - Failure detect/rebind - if a server fails, clients need to detect it and rebind to a different server
        - Client-driven recovery: client detects server failure (connection error), reconnects to successor server, reestablishes session

            - Transparent failure recovery: system detects server failure (via health monitoring), redirects primary's work to successor (via replacing IP address, e.g.), reestablishes state
                - Re-establishing state: successor recovers to last primary state checkpoint (if using a stateful protocol)
- Stateless vs stateful protocols
    - ***Stateful protocols*** (e.g. TCP): operations occur within a context
        - Server must save state to maintain context
            - Any replacement server must obtain the session state to operate properly
        - Each operation depends on previous operations
    - ***Stateless protocols*** (e.g. HTTP): client supplies necessary context within each request
        - Operations are self-contained and unambiguous
        - Make fail-over easy
            - Successor server needs no memory of past events

Remote file system performance

- Any storage device has bandwidth limitations -> a single server has limited throughput
    - Can stripe/mirror files across multiple servers -> more scalable throughput; utilizes bandwidth of multiple storage devices
- Network impacts on performance
    - Bandwidth limitations
    - Delay implications
        - Especially important if acknowledgements are required
    - Packet loss implications
        - If loss rate is high, will require acknowledgements
- Performance of reads
    - Several steps in a network read:
        - Client application requests read
        - Read request sent via network
        - Server receives read request, fetches requested data
        - Server sends data across network

        - Client receives data, gives to application
    - Most file system operations are reads -> want to optimize every step
    - Common optimization - via ***read caching***
        - Client-side caching: Remote data on the server is cached by the client
            - Eliminates waits for remote read requests, reduces traffic + per-client load for the server
            - Creates risk of consistency issues
        - Server-side caching: Similar to single-machine caching
            - Reduces disk delays, but not network costs
        - Read-ahead caching may be done
            - Benefits are higher than local disk, but so are costs
            - Clients can speculatively request reads, or the server can speculatively perform them
                - Speculative reads may be sent or cached locally
        - Whole-file vs block caching
            - Many distributed file systems (e.g. AFS) use whole-file caching
                - Higher network latency justifies whole file pulls; can then store in local cache-only file system
                    - Can use to satisfy early reads before the entire rest of the file arrives
            - Block caching also common (e.g. NFS)
                - Typically integrated into shared block cache
- Performance of writes
    - Clients issue writes to stored data on servers
        - Issue: What about other clients caching that data?
    - Not caching writes is very expensive; would require traversing network, sending an acknowledgement
    - Caching introduces consistency issues, but improves performance
        - Write-back cache - create a cache for accumulating small writes
            - Combines small writes into larger writes, creating illusion of fast writes
            - Results in fewer, larger network and disk writes; enables local read-after-write consistency, but potential poor remote consistency

                - Whole-file updates - no writes sent to server until file is closed
                    - Reduces many successive updates to a single final result
                        - File might be deleted before it is written, e.g.
                    - Enables atomic updates, close-to-open consistency
                        - May introduce other consistency problems
        - Costs of consistency
            - Caching is essential in distributed for performance and scalability
                - Is easy in a single-writer system - can simply force all writes to go through the cache
            - Issue: multi-writer distributed caching is difficult
                - For each local cache copy, need to determine when it needs to be updated
                    - May not know about other cache copies
                    - Can even have conflicting updates to different local cache copies
        - Distributed cache consistency approaches
            - Time-To-Live: Items are deleted from the cache after some time
                - Generally hope no writes were missed in the meantime
            - Check validity on use: Requires remote access (somewhat self-defeating)
            - Only allow one writer at a time, per file (very restrictive)
            - Change notifications: Notify cachers when the main copy gets an update
                - Most common solution

Reliability and availability with performance

- Distributed systems expected to offer good service even in the face of failures
    - Want to work towards fewer failures + faster recovery in the event of failure
- Improving availability
    - Reduce Mean Time to Failure (MTTF): use more reliable components, fix bugs
    - Alternatively, can reduce Mean Time To Recovery (MTTR): use architectures that take less time to restore service upon recovery
- Scalability and performance
    - Network traffic

- Network messages are expensive - require NIC and network capacity to carry, server CPU cycles to process, client delays to await responses
- Want to minimize messages/client/second
    - Can cache results to eliminate some requests entirely
    - Enable performing complex operations with a single request
    - Can buffer up large writes in write-back cache
    - Can pre-fetch large reads into local cache