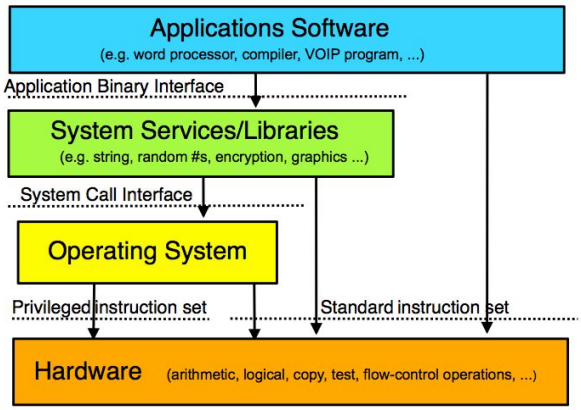
What OS does:

- manage hardware for programs
  - allocate hardware and manage its use
  - enforce controlled sharing and privacy
  - oversees execution and handles problems
- abstracts hardware
  - improve software portability
- provide new abstractions for applications



Instruction Set Architectures:

- set of instructions supported by a computer
  - what bit patterns correspond to what operations

Binary Distribution Model:

- binary distribution is ready to run
- source distribution must be compiled
- OSes usually distributed in binary
- binary model for platform support (device drivers can be added, after-market, same driver works with many versions of OS

Binary Configuration Model:

- good to eliminate manual/static configuration
  - enable one distribution to serve all users, improve performance and ease of use
- automatic resource allocation
  - dynamically (re)allocate resources on demand

Flexibility:

- mechanism/policy separation
  - allow customers to override default policies and change policies w/o changing OS
- dynamically loadable features
  - allow new features to be added, after market
  - file systems, protocols, load module formats etc

Things about OS:

- continues running while apps come & go
- has complete access to hardware (privileged instruction set, all of memory & I/O)
- mediates applications' access to hardware
- trusted (store and manage critical data)

Functionality in the OS:

- OS code expensive
- functionality in OS only if it requires use of privileged instructions, manipulates OS data structures
- functions should be in libraries if the are a service commonly needed by applications

OS speed:

- faster to offer service in the OS than outside it
- there's a push to move services with strong performance requirements down to the OS
- faster because OS has direct access to many pieces of state and system services
  - if sth involves processes communicating, working at app level requires scheduling and swapping them
  - OS can make direct use of privileged instructions

OS not necessarily faster:

- entering OS involves elaborate state saving and mode changing
- if you don't need special OS services, it'll be cheaper to manipulate at the app level

OS and Abstraction:

- OS implements abstract resources using physical resources
    - ex: processes implemented using CPU and RAM
    - ex: files implemented using disks

Why Abstract Resources:

- no need to worry about keeping track of disk interrupts
- compartmentalize complexity (no need to be concerned about what other executing code is doing and how to stay out of its way)
- make it look like you have the network interface entirely for your own use

Common Types of OS Resources:

1) serially reusable resources
    - used by multiple clients, but only one at a time
    - require access control to ensure exclusive use
    - graceful transitions from one user to the next
        - graceful: don't allow 2nd user to access until 1st user finished
        - graceful: no traces of data or state left over from previous user
    - ex: printers
2) partitionable resources (still need graceful transitions)
    - divided into disjoint pieces for multiple clients
        - containment: cannot access resources outside your partition
        - privacy: nobody else can access resources in your partition
        - ex: RAM
3) sharable resources
    - usable by multiple concurrent clients
    - clients do not have to wait for access and do not own a subset of resource
    - ex: copy of the OS, shared by processes
    - doesn't need graceful transitions
        - shareable resource usually doesn't change state
        - isn't reused, doesn't get dirty (execute only copy of the OS)

General OS Trends:

- grown larger and more sophisticated
- role change
    - from shepherding the use of hardware to shielding apps from hardware
    - sophisticated traffic cop
- sit between apps and hardware
- apps become more complex (more complex internal behavior, interfaces, more interactions)
    - OS needs to help with the complexity

Lec 1 Conclusion

- OS interact directly with OS
- OS provide services via abstractions
- OS constrained by many non-technical factors

Key OS Service Abstractions:

- CPU/Memory abstractions

- processes, threads, virtual machines
- virtual address spaces, shared segments

- Persistent storage abstractions
  - files and file systems
- Other I/O abstractions
  - virtual terminal sessions, windows
  - sockets, pipes, VPNs, signals (as interrupts)

Higher Level Abstractions:

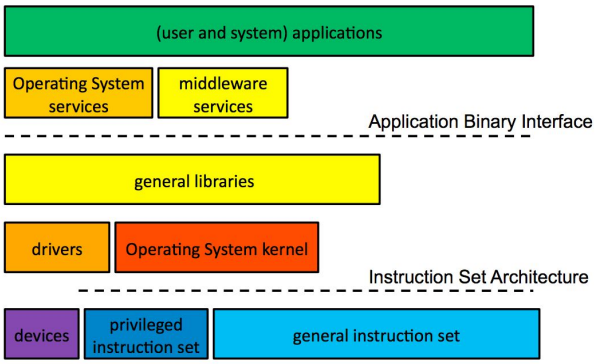
- cooperating parallel processes
  - locks, condition variables, distributed transactions, leases
- security
  - user authentication, secure sessions, at-rest encryption
- user interface
  - GUI widgetry, desktop and window management, multi-media

Under the cover services:

- enclosure management: hot-plug, fans, power, fault handling
- software updates and configuration registry
- dynamic resource allocation & scheduling: CPU, memory, bus resources, disk, network
- networks, protocols, & domain services
  - USB, Bluetooth
  - TCP/IP, DHCP, LDAP

Service Delivery via Subroutines:

- access via direct subroutine calls:
  - push parameters
  - jump to subroutine
  - return values in registers on stack
- advantages:
  - fast
  - DLLs enable run-time implementation binding
- disadvantages:
  - all services implemented in same address space
  - limited ability to combine different languages
  - can't usually use privileged instructions



Layers- Libraries:

- standard utility functions can be found in libraries
- library: collection of object modules (single file that contains many files), modules can be used directly without recompilation
- most systems come with many standard libraries (system services, encryption, stats)

Library Characteristics:

- advantages: reusable code, single well-written/maintained copy, better building blocks
- 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 run-time
- no special privileges

Shared Libraries:

- library modules are usually added to a program's load module
  - each load module has own copy of each library → increase size of each process
  - program must be re-linked to incorporate new library
- instead, make each library a sharable code segment
  - one in memory copy, shared by all processes
  - keep library separate from load module
  - OS load library along with program
- advantages:
  - reduced memory consumption
  - faster program start-ups (need not be loaded again if already in memory)
  - simplified updates: library can be updated, programs automatically get new version when restarted
- limitations:
  - cannot define/include global data storage
  - read into program memory whether they are actually needed or not
  - called routines must be known at compile time
    - only code fetching is delayed until run-time
  - dynamically loadable libraries are more general

Service Delivery via System Calls:

- force entry into OS
  - parameters/returns similar to subroutine, but implement in shared/trusted kernel
- advantages:
  - able to allocate/use new/privileged resources
  - able to share/communicate with other processes
- disadvantages:
  - all implemented on local node, 100 ~ 1000 times slower than subroutine calls

Layers- The Kernel:

- privileged instructions (interrupts, I/O)
- allocation of physical resources (memory)
- ensure process privacy & containment, and integrity of critical resources

Service Delivery via Messages:

- exchange messages with server (via syscalls)
- advantages: server can be anywhere, highly scalable and available, can be implemented in user mode code
- disadvantages: 1000 to 100,000 times slower than subroutine, limited ability to operate on process resources

Layers- Middleware:

- not part of OS
- database, pub/sub messaging system, Apache
- kernel code expensive & dangerous
- user-mode code easier to build, test, debug, more portable, can crash and restart

Interfaces: APIs:

- source level interface specifying:
    - include files, data types, constants, macros, routines, and their parameters
- basis for software portability
    - recompile program for the desired architecture
    - linkage edit with OS-specific libraries
    - resulting binary runs on that architecture and OS
- API compliant program will compile and run on any compliant system
- API requirements frozen at compile time

Interfaces: ABIs:

- binary interface specifying:
    - dynamically loadable libraries, data formats, calling sequences, linkage conventions
- binding of an API to a hardware architecture
- API-only compatibility requires users to obtain & compile their application's sources
- ABI compatibility allows merely loading and running the application (binary)
- ABI defined and implemented by software (OS specific)

Interoperability Requires Compliance:

- cannot test all applications on all platforms
- focus on the interfaces, which are completely and rigorously specified

Critical OS Abstractions:

1) memory abstractions
    - write(name, value)
    - value <- read(name)
    - complications: OS doesn't have abstract devices w/ arbitrary properties
    - hard disk drive implementing the file
        - accesses performed in chunks of fixed size
    - so, file system of OS reorder disk operations to improve performance, optimize based on caching and read-ahead
2) processor abstractions
    - complications:
        - OS has certain amount of physical memory to hold environment info
        - usually only one set of registers
        - process doesn't have exclusive access to CPU (due to other processes)
    - so, there are memory management hardware & software
        - to multiplex memory use among processes
        - giving each the illusion of full exclusive use of memory
        - access control mechanisms, so other processes can't fiddle with my files
3) communications abstractions
    - communication link allows one interpreter to talk to another
    - physical: memory and cables; abstract: networks & interprocess communication mechanisms
    - send(link_name, outgoingmsgbuf): send info contained in buffer on link
    - receive(link_name, incomingmsgbuf): read info off link and put in buffer
    - often asynchronous, received can perform only because send occurred
    - ex: unix-style socket
    - complications: great asynchrony even on same machine, security problems

- so, don't block entire system while waiting for msg delivery, don't completely trust what comes in from the network

Interpreters:

- performs commands, element of computer that get things done
- components:
  - instruction reference: tells interpreter what instructions to do next
  - repertoire: set of things interpreter can do
  - environment reference: describes current state on which the next instruction should be performed
  - interrupts: situations in which the instruction reference pointer is overridden
- ex: a process, OS maintains program counter for process (instruction reference), and its source code specifies its repertoire, its stack heap register contents are its environment , no other interpreters should be able to mess up the process' resources

Federation Frameworks:

- structure that allows similar but somewhat different things to be treated uniformly
- by creating one interface that all must meet
  - ex: make all hard disk drives accept same commands

Abstractions & Layering:

- common to create complex services by layer abstractions
  - ex: file system layers on top of abstract disk, which layers on top of real disk
- layering → good modularity
  - easy to build multiple services on lower layer
  - easy to use multiple underlying services to support a higher layer

Downside of Layering:

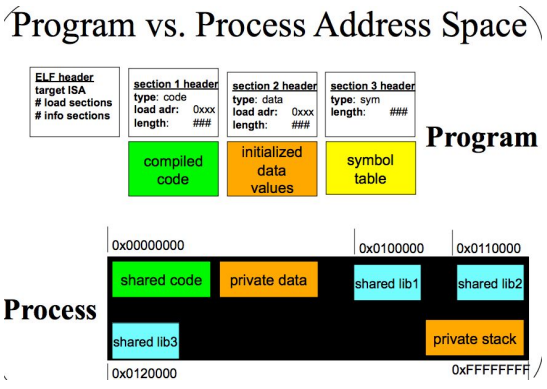
- add performance penalties
- expensive to go from one layer to next
- lower layer limit what upper layer can do
  - abstract disk prevent disk operation reorderings to maximize performance

What is a Process:

- executing instance of a program, object characterized by state

Process Address Space:

- each process has some memory address reserved for private use (its address space)
- OS provide illusion that process has all of memory in its address space



Unix Process Memory Layout:

- code, data, stack
- data grow up, stack grow down, don't meet

Address Space: Code Segments:

- load module (linkage editor output)
  - external references resolved, modules combined into few segments
- code must be loaded into memory

- virtual code segment created
- code must be read in from the load module
- map segment into virtual address space

- code segments are read/only and sharable

Address Space: Data Segments:

- data must be initialized in address space
  - process data segment must be created
  - initial content copied from load module
  - BSS: segments initialized to all zeros
  - map segment into virtual address space
- data segments are read/write and process private
- program can grow or shrink it

Address Space: Stack Segment:

- grows larger as calls nest more deeply
- after calls return, stack frames can be recycled
- OS manages the process's stack segment
  - stack segment created same time as data segment
  - some allocate fixed size stack at program load time
  - some dynamically extend stack as program needs it
- stack segments are read/write and process private

Address Space: Shared Libraries:

- static libraries are added to load module
  - each load module has own copy of each library, program must be relinked to get new Version
- make each library a sharable code segment
  - one in memory copy shared by all processes
  - keep library separate from load modules
  - OS load library along with program (separate from load modules)
  - reduced memory use, faster program loads, easier library upgrades

OS State For a Process:

- state of process's virtual computer
- registers (program counter, stack pointer, general registers)
- address space (text, data, and stack segments, sizes, locations, and contents)
- OS needs data structure to keep track of process state

Process Descriptors:

- stores all info relevant to process
  - state to restore when process is dispatched
  - references to allocated resources

- kept in OS data structure
- used for scheduling, security decisions, allocation issues
- ex: keeps track of
    - unique process ID
    - state of the process (ex: running)
    - parent process ID
    - address space info

Other Process State:

- not all process state is stored in process descriptor
- application execution state is on the stack and in registers

Create New Process:

- OS puts process descriptor in a process table (data structure to organize all currently active processes)
- OS allocate memory for code, data, and stack
- OS then loads program code and data into new segments
- initializes stack segment
- sets up initial registers (PC, PS, SP)

Choices for Process Creation:

- Windows: start with blank process
    - CreateProcess() system call
- Unix/Linux: use calling process as a template, give new process same stuff as the old one (including code, PC)
    - process forking, essentially clones the existing process
    - avoid cost of copying a lot of code
    - easy to manage shared resources (like stdin, stdout, stderr)
    - easy to set up process pipe-lines
    - eases design of command shells

Forking and Data Segments:

- forked child shares parent's code but not its stack, like a 2nd process running same program has reached the same point in its run
- have own data segment

Exec Call:

- replace a process' existing program with a different one
    - new code, different set of resources, different PC and stack
- called after you do a fork
- OS must get rid of child's old code (stack and data areas)
- must load brand new set of code for that process
- must initialize child's stack, PC, and other relevant control structure

Loading Programs into Processes:

- get code to the running version by performing the loading step (initialize code, stack, data)

- have a load module (output of linkage editor)
- modules combined into few segments (code, data, etc)
- compiler cannot execute load module
    - have to execute instructions in memory
    - code must be copied from load module to memory

Destroying Processes:

- reclaim any resources process may be holding (memory, locks, access to hardware devices)
- inform other processes that needs to know
- remove process descriptor from process table

Running processes:

- processes must execute code → OS must give them access to processor core
- but usually there are more processes than cores
- processes need to share cores, can't all execute instructions at once

Load Process:

- initialize hardware, load core's registers, initialize stack, set stack pointer, set up memory control structures, set program counter

How Process Runs:

- uses limited direct execution
- most instructions executed directly by process on the core
- some instructions cause trap to the OS
    - privileged instructions that can only execute in supervisor mode
    - OS takes care of things from here

Limited Direct Execution:

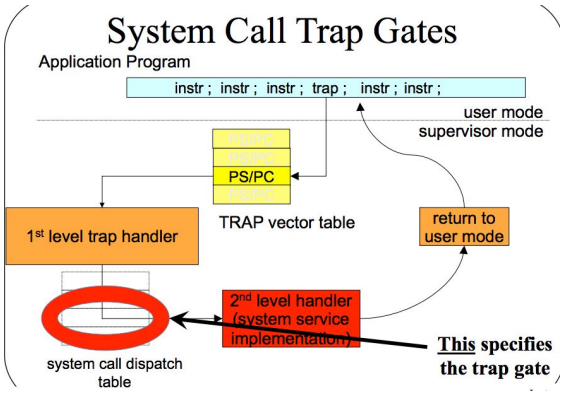
- CPU directly executes all application code punctuated by occasional traps and w/ occasional timer interrupts
- maximizing direct execution is goal (user mode processes)
- enter OS as seldom as possible and get back to application as quickly as possible

Exceptions:

- routine: EOF, arithmetic overflow, conversion error, ← check these after each operation
- unpredictable: segfault, user abort (^C), hang-up, power failure ← asynchronous exceptions
- for asynchronous exceptions: programs can't check, hardware and OS support traps that catch these exceptions and transfer control to the OS

Trap Handling:

- trap cause as index into trap vector table for PC/PS
- load new processor status word, switch to supervisor mode
- push PC/PS of program that caused trap onto stack
- load PC(with address of 1st level handler)



- 1st level handler pushes all other registers
- 1st level handler gathers info, selects 2nd level handler
- 2nd level handler actually deals with the problem (handle event, kill process, return…)

Returning to User-Mode:

- 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
    - resume-user mode execution at next instructions
- saved registers can be changed before return
    - change stacked user r0 to reflect return code
    - change stacked user PS to reflect success/failure

Asynchronous Events:

- want to do other stuff while waiting
- need event completion call-backs

Blocking and Unblocking Processes:

- any part of OS can set blocks, any part can change them, process can ask to be blocked
- when process needs an unavailable resource, change process's scheduling state to blocked
- call scheduler and yield CPU

Swapping Processes:

- processes can run out of main memory
- process can execute only if it's in memory
- sometimes move processes out of main memory to secondary storage (disk drive)
- we don't swap out all blocked processes (swapping is expensive)
- swap out:
    - copy them out to secondary storage, alter process descriptor to indicate what you did
- swap back:
    - reallocate required memory and copy state back from secondary storage
        - both from stack and heap
- unblock process' descriptor to make it eligible for scheduling

Process Queue:

- OS keeps a queue of processes that are ready to run, ordered by which one should run next
- graph first one on the process queue when time comes to schedule new process

Potential Scheduling Goals:

- maximize throughout (get as much work done as possible)
- minimize average waiting time
- ensure some degree of fairness (minimize worst case waiting time)
- meet explicit priority goals

- real time scheduling (scheduled items tagged with a deadline to meet)

Different Kinds of Systems, Different Scheduling Goals:

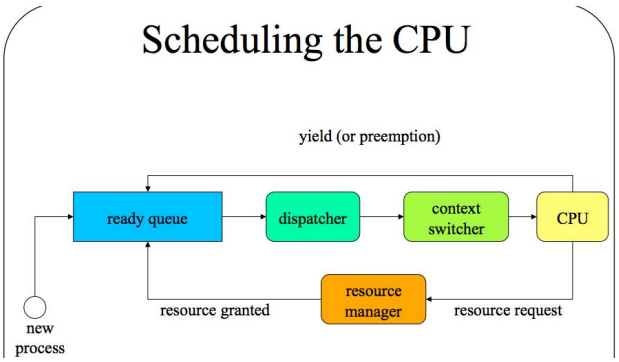
1) Time sharing
   - faster response to interactive programs
   - each user gets equal share of CPU
2) Batch
   - maximize total system throughput
   - delays of individual processes are unimportant
3) Real-time
   - critical operations must happen on time
   - non-critical operations might not happen at all

Preemptive VS Non-Preemptive Scheduling:

1) non-preemptive: scheduled work always runs to completion
   - pros: low scheduling overhead, tends to produce high throughput, simple
   - cons: bugs can freeze machine (infinite loop), not good fairness, make real time and priority scheduling difficult
2) preemptive: scheduler temporarily halts running jobs to run something else
   - pros: good response time, fair usage
   - cons: more complex, requires ability to cleanly halt process and save state, may not get good throughput
   - piggy process can starve others
   - buggy process can lock up entire system
   - dispatching: scheduler move jobs into and out of a processor

Quantify Scheduler Performance:

1) throughput (processes/second)
2) delay (milliseconds)
   - turnaround time? (time to finish job)
   - time to get response?
3) fairness
4) mean time to completion (seconds)



Can't Achieve Ideal Throughput:

- take time to dispatch process (overhead)
- less time is available to run processes per dispatch
- minimize gap: reduce overhead, minimize number of dispatches

Why response time (delay) explode:

- when limits exceed, request dropped → infinite delay for them
- unless careful, overhead during heavy load explode

Graceful Degradation:

- overloaded: meaning a system is no longer able to meet service goals

- we can continue service w/ degraded performance
- or maintain performance by rejecting work
- resume normal service when load drops to normal
- not do: allow throughput to drop to zero & let response time grow without limit

Non-Preemptive Scheduling Algorithms:

1) First come first served
   - runs first process on ready queue until it completes or yields, then run next
   - highly variable delays
   - all processes will eventually be served
   - work well in batch system where response time doesn't matter
   - work well in embedded systems where computations are brief

First Come First Served Example

Dispatch Order		0, 1, 2, 3, 4	
Process	Duration	Start Time	End Time
0	350	0	350
1	125	350	475
2	475	475	950
3	250	950	1200
4	75	1200	1275
Total	1275		
Average wait		595	

Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.

2) Shorted job next
3) Real time schedulers
   - industrial control systems
   - hard: must meet deadline, will fail otherwise (nuclear power plant)
     - must have deep understanding of code (know how long it takes)
     - turn off interrupts (avoid non-deterministic timings, screwed if slowed down
     - set up predefined schedule (no run time decisions)
   - soft: some deadlines may occasionally be missed
     - some have harder classes of deadlines
     - not as vital that tasks run to completion to meet deadline
     - if miss deadline, drop that job
     - algorithms: Earliest Deadline First
       - keep job queue sorted by those deadlines

Preemptive Scheduling:

- thread or process chosen to run, runs until either it yields or OS interrupts
- process can be forced to yield anytime (if higher priority process becomes ready or its own priority lowered)
- interrupted process complicate saving and restoring its state if its not in a clean state
- enables enforced fair share scheduling
- introduces gratuitous context switches
- potential resource sharing problems

Implement preemption:

- process makes sys call or interrupt
- consult scheduler before returning to process
- scheduler finds highest priority ready process, if current process, return as usual, if not, yield and switch to higher priority process

Clock Interrupts:

- can generate interrupt at a fixed time interval, halting any running process
- ensure that runaway process doesn't keep control forever

Round Robin Scheduling Algorithm:

- goal: fair share scheduling (all processes have equal shares of CPU, experience similar delays)
- all processes assigned a nominal time slice (usually same sized slice for all)
- each process runs until it blocks or its time slice expires
- then put at end of the process queue
- then the next process is run, eventually each process reaches front of queue
- all processes get quick chance to do some computation at the cost of not finishing as quickly
- far more context switches which are expensive
- scheduler doesn't halt processes if they block for I/O → kind of like FIFO

Cost of Context Switch:

- enter OS
  - take interrupt, save registers, call scheduler
- cycles to choose who to run
  - scheduler/dispatcher does work to choose
  - move OS context to new process
    - switch stack
  - switch process address space
    - map out old process, map in new process
  - losing instruction and data caches
    - greatly flow down the next hundred instructions

Round Robin Example

Assume a 50 msec time slice (or *quantum*)

Dispatch Order: 0, 1, 2, 3, 4, 0, 1, 2, . . .

Process	Length	1st	2nd	3d	4th	5th	6th	7th	8th	Finish	Switches
0	350	0	250	475	650	800	950	1050		1100	7
1	125	50	300	525						525	3
2	475	100	350	550	700	850	1000	1100	1250	1275	10
3	250	150	400	600	750	900				900	5
4	75	200	450							475	2
Average waiting time: 100 msec										**1275**	**27**

First process completed: 475 msec

Multi-Level Feedback Queue:

- create multiple ready queues:
  - short quantum (foreground) tasks that finish quickly
    - short but frequent time slices, optimize response time
  - long quantum (background) tasks that run longer
    - longer but infrequent time slices, minimize overhead
  - finds balance between good response time and good turnaround time
  - start all processes in short quantum queue
    - move downwards if too many time-slice ends and vice versa
    - processes dynamically find the right queue

Priority Scheduling Algorithm:

- assign each job a priority number
- run according to priority number
- if non preemptive, priority scheduling is about ordering processes

- like shortest job first but with priority instead
- if preemptive:
  - when new process created, it might preempt running process if its priority is higher
- problems: possible starvation, low priority may not be run
- fix by:
  - processes that have run for a long time have priority temporarily lowered
  - processes that have not been able to run have priority temporarily raised

Hard Priorities v.s. Soft Priorities:

- hard: higher priority has absolute precedence over the lower
- soft: higher priority should get larger share of the resource than the lower

Priority in Linux:

- each process in Linux has priority called a nice value
- commands can be run to change process priorities
- only privileged user can request higher priority for his process

Priority in Windows:

- 32 different priority levels
  - use multi queue approach
  - real time scheduling requires special privileges
- users can choose from 5 of these priority levels
- kernel adjust priorities based on process behavior

Memory Management Goals:

1) Transparency
   - process only see its own address space, unaware memory is shared
2) Efficiency
   - high effective memory utilization, low run-time cost for allocation/relocation
3) Protection and isolation
   - private data will not be corrupted, cannot be seen by other processes

Memory Management Problem:

- entire amount of data required by all processes may exceed available physical memory
- switching between processes must be fast
- most processes can't perfectly predict how much memory they will use

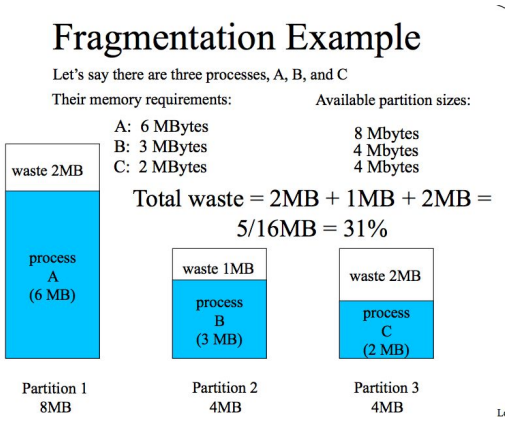
Memory Management Strategies:

1) Fixed partition allocations
   - preallocate partitions for n processes
     - one or more per process, reserve space for largest possible process
   - easy to implement, allocation/deallocation cheap and easy
   - need to enforce partition boundaries (prevent one from accessing another's mem)
   - could use hardware (only accept addresses within the register values)
   - problems:
     - presumes you know how much memory will be used beforehand

- fragmentation causes inefficient memory use, waste space in fixed size blocks
- possible option for embedded systems

2) Dynamic partitions

- variable sized, any size requested
- each partition contiguous in mem addr
- partitions have access permissions for each process
- potentially shared between processes
- each process could have multiple partitions
- problems:
  - not relocatable, not easily expandable
  - subject to fragmentation
  - can't support several applications whose total needs are greater than memory
  - relocation: can't just move because pointers in contents will be wrong
  - expansion: partitions allocated on request, partitions that have been given can't be moved somewhere else in memory
- keep track of variable sized partitions by maintaining free list
  - linkedlist of descriptors (one per chunk) that lists the size of the chunk and whether it's free or not
- subject to external fragmentation (gradually build up small unusable memory chunks
  - solutions: try to recombine fragments into big chunks, avoid creating small fragments
- eliminate fragmentation, more expensive, external fragmentation



3) Relocation

Coalescing Partitions

- all variable sized partition allocation algorithms have external fragmentation
- reassemble fragments: recombine free neighbors whenever possible

Buffer Pools

- popular size buffers → reserve special pools of fixed size buffers
- benefit efficiency (eliminate searching and coalescing)
- process requests a piece of memory for special purpose

Memory Segment Relocation:

- process address space is made up of multiple segments, use segment as unit of relocation
- computer has segment base registers (point to start of each segment in physical memory)
- OS uses these to perform virtual address translation
- virtual address of stack doesn't change
- enables us to move process segments in physical memory
- also need protection (prevent process from reaching outside its allocated memory)

Relocation Solves:

- we can use variable size partitions
  - cut down internal fragmentation
- can move partitions around
  - help coalescing be more effective

Swapping to Disk:

- when process yields, copy its memory to disk, when it is scheduled, copy it back
- but costs of context switch are high

Segmentation Revisited:

- segment relocation used base registers to compute a physical address from virtual address
- allow us to move data around in physical memory by updating base register
- but did nothing about external fragmentation because segments are required to be continuous

Paging:

- divide physical memory into units of single fixed size
  - small one, like 1-4K bytes or words, called page frame
- for each virtual address space page, store its data in one physical address page frame
- use some mechanism to convert virtual to physical pages

Paging and Fragmentation:

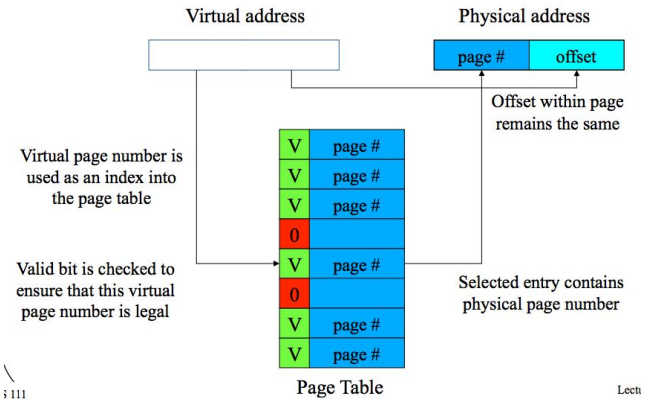
- segment implemented as set of virtual pages
- internal fragmentation: averages only ½ page (half of last one)
- completely non-existent fragmentation

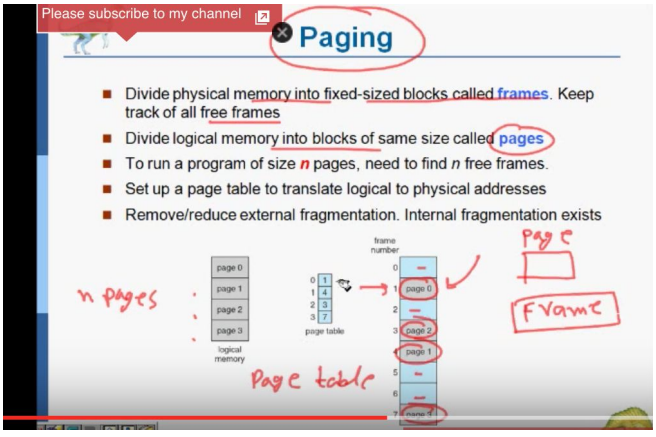
Translation Mechanism:

- need to change a virtual address to a physical address on a per page basis
- use hardware to be fast (Memory Management Unit MMU)

MMU Hardware:

- integrated into CPU
- page tables





Locality of Reference:

- code usually executes sequences of nearby instructions
- typically need access to things in current or previous stack frame

Page Faults:

- when program requests an address from a page that is not in RAM
- generate page fault to tell system to go get it
- only slow a process down, after fault handled, desired page is in RAM
- process can rerun and can use it, program doesn't crash because of page faults

Pages and Secondary Storage:

- when not in memory, pages in typically in a disk in an area called swap space
- page faults may block processes and cause overhead, delay execution
- key is to have the right pages in memory

Virtual Memory:

- a large quantity of memory for each process, at a speed approaching that of actual RAM
- directly accessible via normal addressing
- allow processes to request segments within that space, use dynamic paging and swapping to support the abstraction

Page Replacement:

- rely on locality of reference
- note which pages have recently been used
- use this data to predict future behavior

Clock Algorithms:

- surrogate for LRU
- organizes all pages in a circular list
- MMU sets a reference bit for the page on access
- if so, reset reference bit in MMU & skip this page
- if not, consider this page to be the least recently used
- next search start from this position, not head of list

Implement Working Sets:

- assign page frames to each in-memory process

- observe paging behavior (faults per unit time)
- adjust number of assigned page frames accordingly

Thrashing:

- when we don't have enough memory and whenever anything runs, it grabs a page from someone else, and they get page fault soon after they start running
- prevention:
    - reduce number of competing processes, swap some of the ready processes out

Clean VS Dirty Pages:

- dirty: when in-memory copy of page has been modified, it is dirty
- must be written to disk if swapped out of memory
- clean pages can be replaced any time
- dirty pages must be written to disk before the frame can be reused

Pre-Emptive Page Laundering:

- convert dirty pages to clean ones
- find and write out all dirty, non-running pages

Why Not Just Processes?

- processes are expensive
    - to create: they own resources
    - to dispatch: have address spaces
- different processes are very distinct
    - can't share same address space nor resources (usually)
- not all programs require strong separation

What is a thread:

- a unit of execution/scheduling
- each thread has own stack, PC, registers
- but other resources are shared with other threads
- multiple threads can run in a process
    - all share same code and data space
    - all have access to the same resources

When to use Processes:

- to run multiple distinct programs
- when creation/destruction are rare events
- when there are limited interactions and shared resources
- multiple processes → application run slower, difficult to share resources

When to use Threads:

- for parallel activities in a single program
- when there is frequent creation/destruction
- when they need to share resources
- when they exchange many messages/signals
- when you don't need to protect them from each other

- multiple threads -->have to create, manage, serialize resource use, complicated program

IPC: Synchronous and Asynchronous:

- Synchronous:
    - writes block until msg sent/delivered/received
    - reads block until new msg is available
- Asynchronous:
    - writes return when system accepts msg
        - no confirmation of transmission/delivery/reception
    - reads return promptly if no msg available

IPC and Flow Control

- flow control: make sure fast sender doesn't overwhelm a slow receiver
- queued messages consume system resources
- many things can increase required buffer space (fast sender, non-responsive receiver)

Some styles of IPC:

1) Pipelines
    - data is a simple byte stream
    - no security/privacy/trust issues (under control of a single user)
    - error conditions: EOF, next program failed
    - simple but very limiting
2) Sockets
    - connections between addresses/ports
    - many data options (streams, msges, remote procedure calls)
    - complex flow control & error handling
        - possibility of reconnection
    - trust/security/privacy/integrity
3) Mailboxes and named pipes
    - compromise between sockets and pipes
    - client/server must be on same system
    - server awaits client connections, once open, may be as simple as a pipe
4) shared memory
    - OS arranges for processes to share read/write memory segments
    - only works on local machine

Parallelism Benefits:

- improve throughput
- improved modularity
- improved robustness

Synchronization Problems:

- race conditions: what happens depends on execution order of processes/threads in parallel
    - parallel execution reduces predictability of process behavior
- non-deterministic execution

Critical Section Problem:

- critical section: resource that is shared by multiple threads
- use of resource changes its state
- correctness depends on execution order
- can cause problems when more than one thread executes them at a time
- preventable if we ensure only 1 thread can execute a critical section at a time
- need to achieve mutual exclusion of critical section
- solution1: interrupt disables
    - temporarily block some or all interrupts (done with privileged instructions)
    - prevent time-slice end (timer interrupts)
    - dangers: may delay important operations, could disable preemptive scheduling
- other solutions:
    - avoid shared data
    - eliminate critical sections with atomic instructions (uninterruptable)
    - simple: increment/decrement, and/or/xor
    - complex: test-and-set, exchange, compare-and-swap
    - use atomic instructions to implement locks

When to disable interrupts:

- in situations that involve shared resources
- that involve non-atomic updates
- must disable interrupts in these critical sections
- interrupt service time is costly, disable as few interrupts as possible
- interrupt routines cannot block or yield the CPU

Mutual Exclusion:

- ensure only one thread can execute a critical section at a time
- critical sections: most common for multithreaded applications that share data structures

Recognizing Critical Sections:

- involves updates to object state
- involves multi-step operations
- correct operation requires mutual exclusion

Critical Sections and Atomicity:

- requires:
    1) Before or After atomicity (no overlap)
    2) All or None atomicity (uncompleted update has no effect)

Options for Protecting Critical Sections:

- turn off interrupts to prevent concurrency
- avoid shared data whenever possible
    - may lead to inefficient resource use
- protect using hardware mutual exclusion (atomic CPU instructions)

Atomic Instructions:

- CPU instructions are uninterruptable
- can read/modify/write operations
- either do entire critical section in one atomic instruction
- or use atomic instructions to implement locks

Lock Free Operations:

- alternative to locking or disabling interrupts
- program data structure to perform critical operations with one instruction
- unusable for complex critical sections
- expensive instructions, but cheaper than syscalls

Locking

- protect critical sections with a data structure
- locks: party holding a lock can access the critical section
- when finished with critical section, release lock
- interrupt + lock = dangerous combo
- spin waiting: check if event occurred, if not, check again and again

Spin Locks:

- properly enforces access to critical sections
- wasteful: spinning uses processor cycles
    - bug may lead to infinite spin-waits, delay freeing of desired resource

Building locks from single instructions:

- requires a complex atomic instruction
    - test and set
    - compare and swap

Spinning sometimes makes sense:

- when awaited operation proceeds in parallel
- when awaited operation is guaranteed to be soon
- when spinning does not delay awaited operation

Yield and Spin

- check if occurred, then yield, when rescheduled, check again, repeat until event ready
- problems: extra context switches, might not get scheduled to check until long after

Locking and Waiting Lists

- locks should have waiting lists
- waiting list: shared data structure

Lecture 9

Basic Locking Operation

- when possible concurrency problem shows up
    - obtain lock related to shared resource (block or spin if you don't get it)
    - use the shared resource once you have the lock
    - release the lock

- when implementing lock
    - use atomic instructions or disabling interrupts to ensure no concurrency problems

Semaphores

- more powerful than simple locks, incorporate a FIFO waiting queue
- have a counter rather than a binary flag

Semaphores - Operations

- has two parts: 1) integer counter (initial value unspecified) 2) FIFO waiting queue
- P: decrement counter, if count >=0 return, if counter <0, add process to waiting queue
- V: increment counter, if counter >=0 & queue non-empty, wake 1st process

Semaphores for Exclusion

- initialize semaphore count to one (count reflects #threads allowed to hold lock)
- use P operation to take the lock, first will succeed while subsequent attempts will block
- use V operation to release lock, restore semaphore count to non-negative, if any threads are waiting, unblock the first in line

Using Semaphores for Notifications

- initialize semaphore count to 0 (count reflects #completed events)
- use P operation to await completion, return immediately if already posted, otherwise all callers will block until V is called
- use v to signal completion, increment count, unblock first in line if any threads are waiting
- one signal per wait, no broadcasts

Counting semaphores

- initialize semaphore count to reflect # of available resources
- use P operation to consume a resource, return immediately if available, else all callers will block until V is called
- use V operation to produce a resource, increment the count,, unblock the first in line if any threads are waiting
- one signal per wait: no broadcasts

Limitations of Semaphores

- very simple, few features, more designed for proofs than synchronization
- lack many practical synchronization features
    - easy to deadlock with semaphores
    - cannot check the lock without blocking
    - do not support reader/writer shared access
    - no way to recover from a wedged V operation
    - no way to deal with priority inheritance
- most OSs support them

Mutexes

- a linux/unix locking mechanism
- intended to lock sections of code
- typically for multiple threads of the same process

- low overhead and very general

Object Level Locking

- mutexes protect code critical sections
    - brief durations
    - other threads operating on the same data
    - all operating in a single address space
- persistent objects are more difficult
    - longer critical sections
    - many different programs can operate on them
    - may not run on a single computer
- solution: lock objects rather than code

Advisory vs Enforced Locking

- Enforced:
    - done within the implementation of object methods
    - guaranteed to happen, whether or not user wants it
    - may be too conservative sometimes
- Advisory
    - users expected to lock object before calling methods
    - give users flexibility in what to lock and when
    - mutexes are advisory

Locking Performance

- locking typically performed as an OS system call
- typical system call overheads for lock operations
- if called frequently, high overheads
- even if not in OS, extra instructions run to lock and unlock

Locking Costs

- locking called when you need to protect critical sections to ensure correctness
- overhead of the locking operation may be much higher than time spent in critical section

What if you don’t get your lock

- block, which is much more expensive than getting a lock
    - context switch, milliseconds if swapped out or a queue forms
- performance depends on conflict probability

The Pathfinder Priority Inversion

- use preemptive priority scheduling so high priority task should get the processor
- multiple components share an information bus
    - used to communicate between components
    - essentially a shared memory region
    - protected by a mutex

Reducing Contention

- eliminate critical section entirely
    - eliminate shared resource, use atomic instructions, often you can't
- eliminate preemption during critical section
    - may require disabling interrupts (not always an option)
- reduce time spent in critical section
    - eliminate potentially blocking operations
    - allocate required memory before taking lock
    - do I/O before taking or after releasing lock
    - especially move calls to other routines out of the critical section
    - cost: may complicate the code
- reduce frequency of entering critical section
- reduce exclusive use of the serialized resource
    - only writers require exclusive access
    - allow many readers to share a resource
    - only enforce exclusivity when a writer is active
- spread requests out over more resources
    - change lock granularity
    - spreading activity over many locks reduces contention
    - time/space overhead, more locks, more gets/releases
    - error=prone: harder to decide what to lock and when

Handling Priority Inversion Problems

- in a priority inversion, lower priority task runs because of a lock held elsewhere
    - prevents higher priority task from running
- temporarily increase priority of meteorological task
    - while high priority bus management task was blocked by it
    - so the communications task wouldn't preempt it
    - when lock is released, drop meteorological task's priority back to normal
- priority inheritance: general solution

Lecture 10

What is a Deadlock

- situation where two entities have each locked some resource
- each needs the other's locked resource to continue
- common in complex applications
- result in catastrophic system failures
- hard to find through debugging, easier to prevent at design time
- mostly result from careless design

Deadlocks may not be obvious

- modern software depends on many services, most are ignorant of one-another
- each require numerous resources

Deadlocks and different resource types

- commodity resources
  - clients need an amount of it (memory)
  - deadlocks result from over-commitment
  - avoidance can be done in resource manager
- general resources
  - clients need a specific instance of something
    - a particular file or semaphore
    - a particular message or request completion
  - deadlocks result from specific dependency relationships
  - usually done at design time

Four basic conditions for deadlocks

1) mutual exclusion
   - resources can each only be used by one entity at a time
   - P1 having the resource preludes P2 from getting it
2) incremental allocation
   - no deadlock if processes/threads are allowed to ask for resources whenever
   - no deadlock if they get all they need to run to completion or don't get all they need and abort
3) no pre-emption
   - when an entity has reserved a resource, you can't take it away
4) circular waiting
   - A waits on B which waits on A

Deadlock avoidance

- advance reservations
  - only grant reservations if resources are available
  - over-subscriptions are detected early (before process get resource)
  - client must be prepared to deal with failures
  - over-booking vs. under-utilization
  - ex: memory reservations, disk quotas
  - failures will not happen at request time, only at reservation time
- dealing with reservation failures
  - apps must handle failures, like continue running and refuse to perform request
  - must have a way of reporting failure
  - all critical resources must be reserved at start-up time
- release locks before blocking
  - reacquire: does not involve hold-and-block, so cannot result in deadlock

How to monitor process health

- look for obvious failures (process exits, core dumps)
- passive observation to detect hangs (is process consuming CPU time)

- external health monitoring (pings, null requests, standard test requests)
- internal instrumentation (white box audits, exercisers)

What to do with unhealthy processes

- kill and restart all affected software
- kill as many as necessary but as few as possible

Monitors: simplicity vs. performance

- monitor locking is very conservative
    - lock entire class
- can create performance problems, eliminates parallelism, a convoy can form

Lecture 11

How to design for performance

- establish performance requirements early
- anticipate bottlenecks
    - frequent operations (interrupts, copies, updates)
    - limiting resources (network/disk bandwidth)
    - traffic concentration points (resource locks)
- design to minimize problems (eliminate, reduce use, add resources)
- include performance measurement in design

Common types of system metrics

- a measurable quantity
- duration/response time
- processing rate (how many web requests handled per second)
- resource consumption (how much disk is currently used)
- reliability (how many messages were delivered without error?)

Simulated work loads

- controllable operation rates, parameters mixes
- scalable to produce arbitrarily large loads
- can collect excellent performance data
- random traffic not a usage scenario
- may not create all realistic situations
- wrong parameter choices yield unrealistic loads

Replayed workloads

- captured operations from real systems
- represent real usage scenarios
- can be analyzed and replayed over and over
- hard to obtain
- not necessarily scalable
- represent a limited set of possible behaviors
- limited ability to exercise little-used features

- kept around forever, become stale

Testing under live loads

- real combinations of real scenarios
- measured against realistic background loads
- enables collection of data on real usage
- demands good performance and reliability
- potentially limited testing opportunities
- load cannot be repeated or scaled on demand

Standard benchmarks

- heavily reviewed by developers and customers
- believed to be representative of real usage
- standardized and widely available
- inertia, used where they are not applicable

Types of Performance problems

- non-scalable solutions
- bottlenecks
- accumulated costs

Execution profiling

- automated measurement tools
- compiler options for routine call counting
- statistical execution sampling
- tools to extract data and prepare reports
- useful in identifying bottlenecks

End to end testing

- client side throughput/latency measurements
- easy tests to run, easy data to analyze, reflect client experienced performance
- no information about why it took that long, no info about resources consumed

Lecture 12

Typical properties of device drivers

- highly specific to the particular device
- inherently modular
- usually interacts with the rest of the system in limited, well defined ways
- correctness is critical

Abstractions and device drivers

- OS defines idealized device classes (disk, display, printer, tape, network, ports)
- classes define expected interfaces/behavior (all drivers in class support standard methods)
- device drivers implement standard behavior
- abstractions regularize and simplify the chaos of the world of devices

What can driver abstractions help with

- encapsulate knowledge of how to use the device

- encapsulate knowledge of optimization
- encapsulate fault handling

Device drivers cs. Core OS code

- device driver code is in the OS
- common functionality belongs in the OS
    - caching
    - file systems code not tied to a specific device
    - network protocols above physical/link layers
- specialized functionality belongs in the drivers
    - things that differ in different pieces of hardware
    - things that only pertain to the particular piece of hardware

Devices and interrupts

- devices are primarily interrupt-driven
- work at different speed than the CPU (typically slower)
- can do their own work while CPU does something else
- use interrupts to get the CPU's attention

Devices and Busses

- devices are not connected directly to the CPU
- CPU and devices connected to a bus (may be same may be different)
- devices communicate with CPU across the bus
- bus used both to send/receive interrupts and to transfer data and commands

CPUs and Interrupts

- interrupts look like traps, traps come from CPU, interrupts caused externally to CPU
- unlike traps, can be enabled/disabled by special CPU instructions
    - device can be told when they may generate interrupts

Good device utilization

- if device sits idle, throughput drops
- delays can disrupt real-time data flows
- important to keep key devices busy
- parallelize activities by letting CPU and device operate tgt
    - let device use bus instead of CPU
    - modern CPUs try to avoid going to RAM (registers, caching on chip)

Direct Memory Access (DMA)

- allows any two devices attached to the memory bus to move data directly
    - without passing through CPU first
- bus can only be sued for one thing
- often CPU doesn't need it, with DMA, data moves from device to memory at bus speed

I/O and Buffering

- most I/O requests cause data to come into the memory or to be copied to a device

- data requires a place in memory called a buffer
- OS needs to make sure buffers are available when devices are ready to use them

OS Buffering Issues

- fewer/larger transfers are more efficient
    - may not be convenient for applications
- OS can consolidate I/O requests
- enables read-ahead (OS reads/caches blocks not yet requested)

Deep Request Queues

- having many I/O operations queued is good
    - maintains high device utilization (little idle time)
    - reduces mean seek distance/rotational delay
    - possible to combine adjacent requests
    - can avoid performing a write it all
- ways to achieve deep queues:
    - many processes making requests
    - individual processes making parallel requests
    - read-ahead for expected data requests
    - write-back cache flushing

Performing Double Buffered Input

- having multiple reads queued up, ready to go
- filled buffers wait until appreciation asks for them
    - application doesn't have to wait for data to be read

Memory Mapped I/O

- DMA may not be the best way to do I/O
    - designed for large contiguous transfers
    - some devices have many small sparse transfers
- instead, treat registers/memory in device as part of the regular memory space
- ex: bit-mapped display adaptor
- low overhead per update, no interrupts to service
- relatively easy to program

Memory Mapping vs. DMA

- DMA performs large transfers efficiently
    - better utilization but considerable per transfer overhead
- memory-mapped I/O has no per-op overhead
    - but every byte is transferred by a CPU instruction
- DMA better for occasional large transfers
- memory-mapped better for frequent small transfers, more difficult to share

Device Driver Interface (DDI)

- standard (top-end) device driver entry points

- some entry points correspond directly to sys calls (open, close, read, write)
- some are associated with OS frameworks
    - disk drivers are meant to be called by block I/O
    - network drivers are meant to be called by protocols

Driver/Kernel interface

- specifies bottom-end services OS provides to drivers
    - things drivers can ask the kernel to do
- must be very well-defined and stable
- each OS has its own DKI, but they are all similar

Criticality of Stable Interfaces

- drivers are largely independent from the OS
- OS and drivers have interface dependencies, must be carefully managed

Why classes of drivers

- classes provide a good organization for abstraction
- provide a common framework to reduce amount of code required
- ensure all devices in class provide certain minimal functionality

Character device superclass

- devices that read/write one byte at a time
- keyboards, monitors

Block device superclass

- devices that deal with a block of data at a time
- usually a fixed size block
- disk drive

Network Device Superclass

- devices that send/receive data in packets
- originally treated as character devices
- only used in the context of network protocols
- ethernet cards, bluetooth devices

Lecture 13

File Systems and Hardware

- file systems typically stored on hardware providing persistent memory
- with the expectation that a file put in one place will be there when we look again

What hardware do we use

- until recently, file systems were designed for disks
- require many optimizations based on particular disk characteristics
- disk provides cheap persistent storage at the cost of high latency

Data and Metadata

- data: information that the file is supposed to store (instructions of program)
- metadata: information about the information the file stores (how many bytes)
- both must be stored persistently

Desirable file system properties

- persistence
- easy use model
- flexibility
- portability
- reliability (want files to be there when we check, free of errors)
- performance
- suitable security (whoever owns it can control who accesses it)

Three file system calls

1) File container operations
   - manipulates files as objects
   - ignore contents of file
   - get/set attributes, ownership, create/destroy files and directories
2) Directory operations
   - provide organization of file system
   - directories translate a name to a lower level file pointer
3) File I/O operations
   - open: use name to set up an open instance
   - write and write data to and from file
   - map file into address space

The file systems layer

- all implemented on top of block I/O
- should be independent of underlying devices
- all file systems perform the same basic functions
  - map name to files
  - manage free space and allocate to files
  - create and destroy files

Why Device Independent Block I/O

- better abstraction than generic disks
- allows unified LRU buffer cache for disk data
- provides buffers for data reblocking
- handles automatic buffer management

Cache

- eliminate many disk accesses which are slow
- users often open files from same directory
- users often read and write a single block in small operations

On-disk file control structures

- description of important attributes of a file
- virtually all file systems have such data structures
- core design element of a file system

In-Memory Representation

- on-disk structure pointing to disk blocks
- when file opened, in-memory structure is created
- disk version points to disk blocks
- in-memory version points to RAM pages
- also keeps track of which blocks are dirty and which aren't

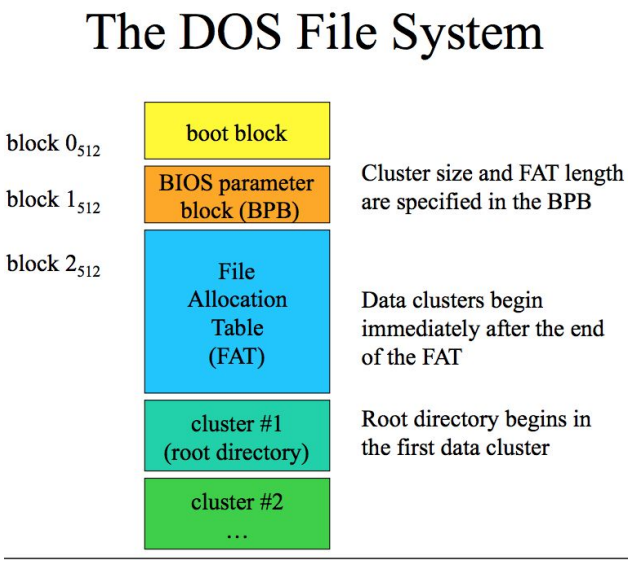
Basics of file system structure

- most file system live on disks
- most blocks will store user data
- some will store metadata
- all OS have such data structures

The boot block

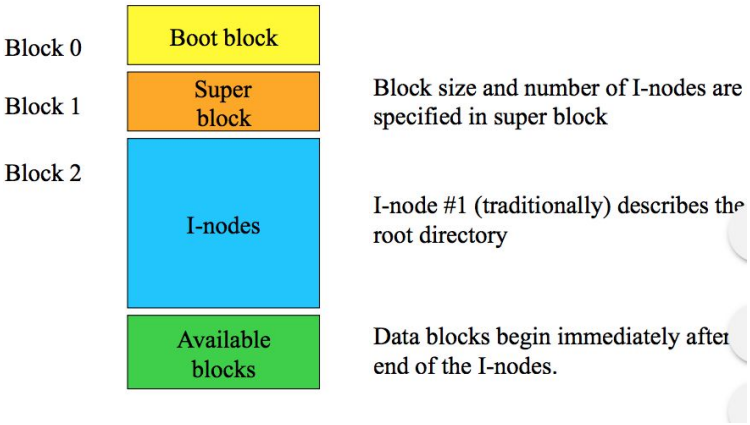
- 0th block of a disk
- not usually under control of file system
- file sys start work at block 1

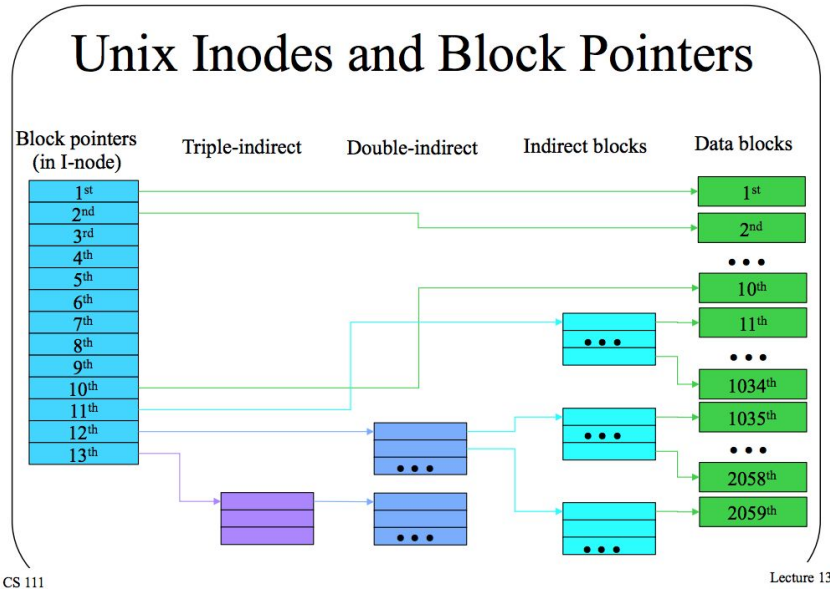
DOS File system



- FAT table: contain number of the next cluster in file
  - 0 entry means that the cluster is not allocated
  - -1 means end of file
  - entire table is kept in memory (no disk I/O required to find cluster)

Unix System V File System





- all block pointers triple indirect
- allows us to access up to 40K bytes without extra I/Os
    - double and triple must themselves be fetched off disk

How big a file can unix handle

- on-disk inode: 13 block pointers
    - first 10 point to first 10 blocks of file
    - 11th: indirect block
    - 12th: double indirect block
    - 13th: points to triple indirect block

Lecture 14

Outline

- allocating and managing file system free space
- other performance improvement strategies

- file naming and directories
- file reliability issues

Creating a new file

- unix
    - search super block free i-node list
    - take the first free inode
- DOS
    - search the parent directory for an unused directory entry
- initialize new file control block
- give new file a name

Extend a file

- find free chunk of space
    - traverse free list to find one
    - remove the chosen chunk from free list
- associate it with the appropriate address in the file

Deleting a file

- release all space allocated to file
- deallocate file control block
    - unix: zero inode and return to free list
    - DOS: zero first byte of the name in parent directory
        - indicate that directory entry is no longer in use
- unix, return each block to the free block list
- DOS does not free space
    - uses garbage collection
    - search out deallocated blocks and add them to the free list later

BSD File system

- use bit map approach to manage free space
- divide each file system into cylinder groups
- enables significant reductions in head motion

Extend BSD File system

- determine cylinder group for the file's inode
- find cylinder for the previous block in the file
- find a free block in the desired cylinder
- update the inode to point to the new block

Read ahead

- requests blocks from the disk before any process asked for them
- risks: may waste disk access time reading unwanted blocks

Write caching

- most disk writes go to a write back cache

- aggregates small writes into large writes
- eliminates moot writes
- accumulates large batches of writes

Types of disk caching

- general block caching
- special purpose caches
- special purpose caches are more complex

Hierarchical name spaces

- graphical organization
- nested directories can form a tree

Directories are files

- directories are a type of file
- contain multiple directory entries
- user applications allowed to read directories
- usually only the OS is allowed to write them

Multiple file names in unix

- metadata stored in the file inode
- all links provide the same access to the file
- all links equal

Symbolic links

- different way of giving files multiple names
- indirect reference to some file
- contents is a path name to another file
- not a reference to the inode, will not prevent deletion

Robustness - ordered writes

- write out data before writing pointers to it
- write out deallocations before allocations
- greatly reduced I/O performance
- may not be possible, doesn't solve problem

Robustness - audit and repair

- audit file system for correctness prior to mount
- use redundant info to enable automatic repair
- no longer practical

Journaling

- create circular buffer journaling device
    - writes are sequential, can be batched, relatively small
- journal all intended file updates (inode updates, block write/alloc/free)
- efficiently schedule actual file system updates (write-back cache, batching)
- journal completions when real writes happen
- small writes still inefficient

- much faster than data writes (no competing head motion)
- scanning on restart is fast

Navigating a logging file system

- inodes point at data segments in the log
- updated inodes are added at the end of lg
- index points to latest version of each inode
- find and recover the latest index

Redirect on write

- many modern file systems do this
  - once written, blocks and inodes are immutable
  - add new info to log, update index
- old inodes and data remain in the log
- price is management and garbage collection

Lecture 15

Outline

- intro
- authentication
- access control
- cryptography

Access control lists

- for each protected object maintain a single list
- each list entry specifies who can access the object

Capabilities
Cryptography
Cryptographic Keys
Advantages/Disadvantages of Symmetric cryptosystems
Some popular symmetric ciphers
Asymmetric cryptosystems (public key)
PB Key distribution
Example public key ciphers
Combined use of symmetric and asymmetric cryptography

Lecture 16

Outline

- distributed system paradigms
- remote procedure calls
- distributed synchronization and consensus
- distributed system security

Transparency
Changing paradigms
Distributed system paradigms
Loosely coupled systems (web servers, app servers, cloud computing)

Horizontal scalability
MapReduce
Remote procedure calls
Distributed synchronization
Leases - More Robust Locks
Distributed consensus
Typical consensus algorithm
Distributed security is harder
Goals of network security
Elements of network security
Symmetric encryption
Cryptographic hashes
Public key cryptography
Secure socket layer (SSL)
Digital signatures
Signed load modules
PK certificate

Conclusion

- distributed systems offer us much greater power than one machine
- do so at the costs of complexity and security risk
- handle complexity using distributed systems in carefully defined ways
- handle security by proper use of cryptography and other tools

Lecture 17

Outline

- data on other machines
- remote file access architectures