CS 111(Operating Systems) Notes

3 Easy Pieces Textbook

Chapter 1 - Dialogue

- The 3 key pieces of operating systems are **virtualization**, **concurrency**, and **persistence**.

Chapter 2 - Introduction

- A running computer program executes instructions.
- The processor fetches an instruction from memory, decodes it, and executes it.
  - Von Neumann model of computing
- The body of software responsible for making easy to run programs is the **operating system**.
- OS makes sure that everything operates correctly and efficiently through virtualization. It takes a physical resource and transforms it into a more powerful and easy to use virtual form of itself.
- OS also has the job of being the resource manager.
- The **policy** of the OS determines what programs should run if two are run at the same time.
- A **thread** is a function running within the same memory space as other functions.
- **Abstractions** are built so that the system is convenient and easy to use.
- Difference between a system call and a procedure call is that system calls transfer control to the OS.
- User applications run in user mode which means that the hardware restricts what the software can do.
- When a system call is initiated through a hardware instruction (called a **trap**), hardware transfers control to a trap handler, and raises privilege level to kernel mode.

Chapter 3 - Virtualization

- **Virtualization** is taking a physical thing (like a CPU) and making it look like there are multiple virtual ones. Each application thinks it has it's own CPU to use, but it actually doesn’t.
- One of the abstractions the OS provides to users is a process, which is a running program.
- Users run multiple processes at a time, so how do we provide the illusion of lots of CPUs?
  - The solution is **time sharing**, where processes are run, then stopped, then run again.

- Time sharing is one of the most basic techniques to share a resource. It's used for a little by each entity.
- Context switching is stopping one process and then starting another one on that CPU.
- Policies are algorithms for making some decision in an OS. The **scheduling policy** is the way that the CPU decides the order in which to run the processes.
- Process API
    - Create: For creating processes
    - Destroy: For killing processes
    - Wait: Waiting for a process to finish
    - Misc: Suspending process and then starting it again
    - Status: Finding out how long the process has run for and the state it is in
- Before running anything, the OS must do some work to get the important program bits from disk and into memory.
- A process can be running, ready, or blocked

Chapter 5 - Process API

- **Fork** system call is used to create a new process.
    - The process that's created is almost an exact copy of the calling process.
    - This new process has it's own private memory, it's own registers, and it's own PC.
- Each process has an identifier, a PID
- The CPU scheduler determines which process runs at a given time. Because the scheduler is complex, we don't really know which process will run first and therefore it is non-deterministic.
- The **wait** system call is normally something the parent calls to delay execution until the child finishes executing.
- The **exec** system call is when you want to run a program that's completely different from the calling program.
    - It doesn't create a new process or anything. It just loads the code from the executable that you want to run, and then runs that program.
    - A successful call to exec never returns.

Chapter 6 - Limited Direct Execution

- In order to virtualize the CPU, the OS needs to share the CPU so that one process runs for a little bit, then run another, etc. This is time sharing.
    - In doing this, we need to make sure we still have performance (low overhead) as well as control (running processes efficiently)
- When a program wants to run, OS creates an entry for it on the process list, allocates memory for that program, loads program into memory, sets up argc/argv, clears registers, and then calls the main function. The program runs main, then returns, and then the OS frees memory of process and removes it from the process list.

- We want the program to sometimes stop running so that we can switch to another process.
- There is also a problem of when a certain process wants to be able to perform I/O and other restricted operations. You want to allow the process to do this, but don't want to give the process complete control over the system.
- Code that runs in **user mode** is restricted in what it can do (can't issue IO requests).
- **Kernel mode** is the mode in which the OS runs.
- If a user process wants to perform some privileged operation, it needs to use a system call.
- To execute a system call, a program must execute the trap instruction which jumps into the kernel, raises the privilege level to kernel mode, performs whatever privileged operations there are, and then the OS issues a return from trap instruction that returns into the calling user program while reducing the privilege back to user mode.
- The user program jumps to whatever address is specified by the trap table. User code can't specify the exact address to jump to, but rather must request a particular system call.
- One problem that we encounter is when a certain process is running, the OS doesn't have control of the CPU, the process does, so how can it make the decision to stop one process and then start another.
- A process transfers control to the CPU through either a system call or doing something illegal which generates a trap to the OS.
- The OS can also do a **timer interrupt** (which is ofc a privileged operation), where a device is programmed to raise an interrupt every so many milliseconds, then the current process is halted, and an interrupt handler in the OS starts running.
- When the interrupt occurs, it is the hardware's job to make sure to save the state of the current process so that when the return from trap instruction jumps back to the program, the running program will be able to resume correctly.
- When the OS regains control (through interrupt or sys call or whatever), the decision of what process next to run is made by the scheduler.
    - If the OS decides to switch processes, then the OS executes a low level piece of code called a **context switch**. It saves a few register values for the current process and restores a few for the future process.
- If during an interrupt or during a trap handle, another interrupt occurs, then what happens?
    - An OS could simply disable interrupts during interrupt processing.
    - There are also locking schemes that prevent concurrent access to internal data structures.
- **Limited Direct Execution** is basically where you run the program you want to run on the CPU, but you first set up the hardware to limit what the process can do without OS asistance.

Chapter 7 - Scheduling - Introduction

- The processes running in a system are collectively called the **workload**.
- **Turnaround time** of a job/process is time of completion - time of arrival.
- The most basic algorithm is FIFO scheduling.
- Convey effect is when short potential consumers of a resource get queued behind a heavyweight user of a resource.
- The other scheduling algorithm could be Shortest Job First, which can cut down the turnaround time. This is optimal if all jobs arrive at the same time.
- The best algorithm is Shortest Time to Completion First, which means that as jobs come in, you run the ones that have the least time left first. This technique has preemption because even after you start a job, you can block it and prioritize the shorter jobs.
- **Response time** is the time from when the job arrives in a system to the time that it is first scheduled.
- STCF and the previous ones are not great for response time.
- Round Robin scheduling is the idea that instead of running jobs to completion, RR runs a job for a time slice and then switches to the next job in the run queue.
    - The shorter the time slice, the more quickly that more jobs can be run (even for just a bit). However, this also increases the overhead of context switching.
- RR is one of the worst times if turnaround time is the metric.
- RR = good for response time, bad for turnaround time.
- SJF and STCF = good for turnaround time, bad for response time.

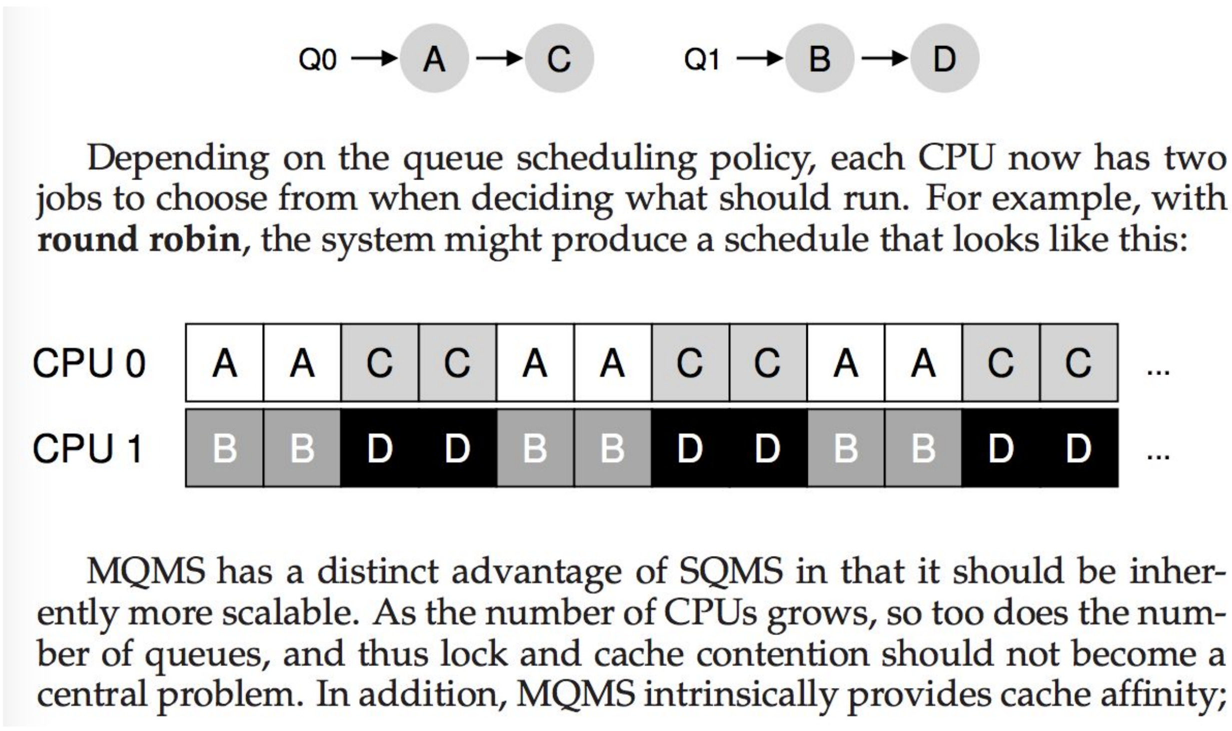
Chapter 8 - Scheduling: The Multi-Level Feedback Queue

- The **multi-level feedback queue** is another scheduler method.
    - Rule 1: If Priority(A) > Priority(B), A runs and B doesn't
    - Rule 2: If Priorities are equal, A and B run in RR.
    - Rule 3: When a job enters the system , it is placed at the highest priority.
    - Rule 4a: If a job uses up an entire time slice while running, its priority is reduced.
    - Rule 4b: If the job gives up the CPU before its time slice is up, then it stays at the same priority level.
- MLFQ varies the priority of a job based on it's observed behavior.
- It will use the history of the job to predict future behavior.
- (Idk, maybe add notes on the MLFQ rules)
- Basically, if a job comes in, OS doesn't know if it's going to be a short job or a long job, so it assumes it's a short job and gives it high priority, and then if it still doesn't finish then it will move down the queue.
- There is a problem of starvation in that if there are too many interactive jobs, then they will consume all CPU time, and the long running jobs would never receive CPU time.

Chapter 10 - Multi-CPU Scheduling

- When you have more than one CPU, that doesn't necessarily help you sometimes because typical applications use a single CPU, so having multiple CPUs doesn't make that particular program run any faster.

    - You’ll have to rewrite your program to use threads, which can spread work across multiple CPUs.
- A problem that arises is multiprocessor scheduling. How should the OS schedule jobs on multiple CPUs.
- Another problem is that of cache coherence where there is one main memory but multiple CPUs. If CPU 1, loads something from main memory, puts it in it's cache, and then modifies the value in the cache to something else (it will update the main memory later), there is an issue is CPU 2 wants that value but then looks in the main memory and accessing the old value instead of the new value.
    - The solution is bus snooping where each cache pays attention to memory updates by observing the bus that connects them to main memory.
- When accessing (and in particular, updating) shared data items or structures across CPUs, mutual exclusion primitives (such as locks) should likely be used to guarantee correctness
- Another issue with a multiprocessor cache scheduler is cache affinity which is when a program on a CPU builds up a fair amount of state in that CPU’s cache, and the next time the process runs, it is faster to run the process on that same CPU instead of switching.
- Building a scheduler for a multiprocesser system.
    - Single queue multiprocessor scheduling: Put all the jobs that need to be scheduled into a single queue.
        - Problems could be with bouncing around from CPU to CPU for a particular process, and not using that cache affinity to improve performance.
    - Multi queue multiprocessor scheduling: Same except you have a queue for each CPU. Each queue will follow a particular scheduling decision (like round robin).



$$\begin{tikzcd} Q0 \arrow[r] & A \arrow[r] & C & Q1 \arrow[r] & B \arrow[r] & D \end{tikzcd}$$

Depending on the queue scheduling policy, each CPU now has two jobs to choose from when deciding what should run. For example, with **round robin**, the system might produce a schedule that looks like this:

CPU 0	A	A	C	C	A	A	C	C	A	A	C	C	...
CPU 1	B	B	D	D	B	B	D	D	B	B	D	D	...

MQMS has a distinct advantage of SQMS in that it should be inherently more scalable. As the number of CPUs grows, so too does the number of queues, and thus lock and cache contention should not become a central problem. In addition, MQMS intrinsically provides cache affinity;

- The problem is load balancing, where if A and C finish early, then CPU 0 won't be used at all, while CPU 1 is trying to finish both B and D.
- Migration is when you move a job from one CPU to another.
- Work stealing is when a source queue that is low on jobs will peak at another target queue to see whether it can steal a job to help with the balance load.

Chapter 13 - Address Spaces

- Operating systems are hard because of the era of multiprogramming where multiple processes are ready to run at a given time.
  - This increases the effective utilization of the CPU.
- Address space is the running program's view of the system memory. Address space contains:
  - The code: The instructions
  - The stack: Keeps track of where it is within in the function call chain and is used to allocate local variables and pass parameters and return values to and from routines.
  - Heap is used for dynamically allocated memory, like with malloc.
- If two processes are properly isolated from one another, one can fail without affecting the other.
- Virtual memory needs to have
  - Transparency - Programs shouldn't be aware that the memory is virtualized.
  - Efficiency - Make sure that the virtualization is efficient in terms of space and time.

    - Protection - Protection in terms of not accessing or affecting memory contents outside of it's process's address space.
    - Is responsible for providing the illusion that every program or process has it's own memory address space when in fact, there's only one memory.
- Even when you print out the address of a pointer, that's it's a virtual address within the process's address space. Only the OS knows the actual physical location as to where the information is stored.

Chapter 14 - Memory API

- Stack memory is where the allocations and deallocations of variables and stuff are managed implicitly by the compiler for you (it's called automatic memory sometimes).
    - When you return from the function, the space automatically gets deallocated for you.
- Heap memory is the memory that is explicitly handled by the programmers
    - Usage of malloc and dealloc.
- The malloc() call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns NULL
- Compile time operation is when you pass in a sizeof(int) or sizeof(double) instead of an actual number of bytes to allocate.
    - Sizeof is operator, not a function call.
    - Figuring out the purpose of operators is done at compile time.
    - Function calls are evaluated at runtime.
- Forgetting to free memory is known as a memory leak.
- Freeing memory and then trying to use the pointer results in dangling pointers.
- Double freeing is undefined behaviour.
- The brk system call is used to change the location of the program's break aka the location of the end of the heap.

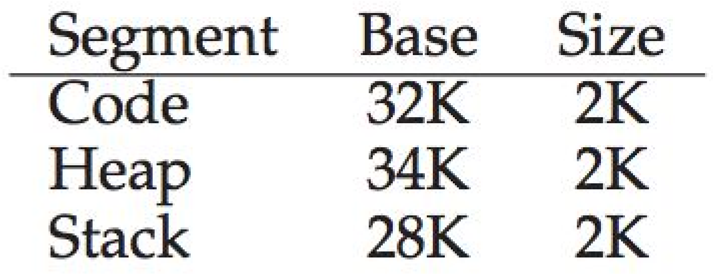
Chapter 15 - Address Translation

- Limited direct execution lets the program run directly on hardware, however at certain points, there will be a system call or a timer interrupt and the OS gets involved so that the right thing happens.
    - There is a tradeoff between efficiency in that you want the programs to be able to run without having to context switch into kernel mode, but at the same time, you want to make sure that the OS still has control over the program.
- In order to implement virtualization of memory, you have to do address translation, which is where the hardware transforms each memory access, changing the virtual address provided by the instruction to a physical address where the actual desired information is located.
    - The OS has to get involved so that the correct translations take place. The hardware can't do it on its own.

- Every memory reference generated by a process is a virtual address and then the hardware adds the contents of the base register (there is also a bounds register that makes sure none of the address values go beyond a certain point) to the address, resulting in a physical address that can be issued to the memory system.
- Relocation of the address happens at runtime. Because of that, address translation is also known as dynamic relocation.
- A free list is a data structure that is a list of the ranges of the physical memory which are not currently in use.

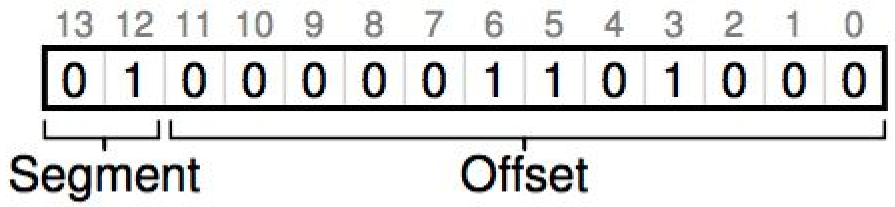
Chapter 16 - Segmentation

- The idea of segmentation is having a base and bounds pair per logical segment of the address space.
    - Segment is a contiguous portion of the address space of a particular length.
- We take the 3 segments of an address space (code, stack, heap) and then place those segments into physical memory separately so that only used memory is allocated space in physical memory.



Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

- Segment registers are used to let the programs know what segment (code, heap, or stack) we're referring to, as well as the offset.

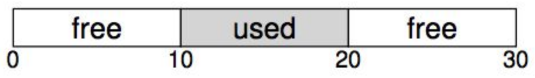


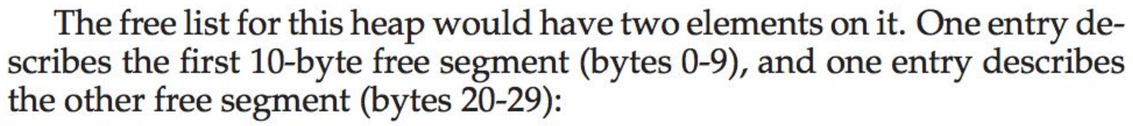
- The hardware also needs to know which way the data structures grow.
- To save memory, it is useful to share certain memory segments between address spaces.
- Segmentation is coarse-grained whenever it chops up the address space into large and few segments.
- Fine-grained segmentation has a large number of smaller segments.
- The main idea behind segmentation is that pieces of the address space are relocated into physical memory as the system runs, and thus a huge savings of physical memory is achieved relative to our simpler approach with just a single base/bounds pair for the entire address space. Specifically, all the unused space between the stack and the heap

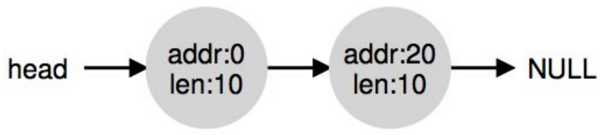
need not be allocated in physical memory, allowing us to fit more address spaces into physical memory.
    - The best upside of segmentation is that it avoids a huge waste of memory between logical segments (code, heap, stack) of the address space.
- The problem that comes up is of external fragmentation, where you have data scattered all over physical memory and nothing is really contiguous, thus making access somewhat slow.
    - A solution is to compact the physical memory by rearranging the existing segments.
- A problem with the above is that copying memory and moving it can be expensive.
    - A solution is to use free-list management algorithms that keep large extents of memory available for allocation.

Chapter 17- Free Space Management

- Free space management revolves around figuring out how to go about the problem of external fragmentation. You don't have situations where requests can't be satisfied because there is no single contiguous space for the data even though the total amount of free space exceeds the size of the request.
- Free lists can allow us to track the size of allocated regions quickly and easily in order to keep track of what is free and what isn't.




The free list for this heap would have two elements on it. One entry describes the first 10-byte free segment (bytes 0-9), and one entry describes the other free segment (bytes 20-29):



- If we have a request for anything smaller than 10 bytes, we'll use splitting, where we find a chunk of memory that can satisfy the request and split it into two. The first chunk will return to the caller and the second will remain on the list with it's length value changed.
- Coalescing happens after you free some memory, then add it to the list, and then make sure that the proper length values are increased.
    - The idea is simple: when returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly freed space sits right next to one (or two, as in this example) existing free chunks, merge them into a single larger free chunk.

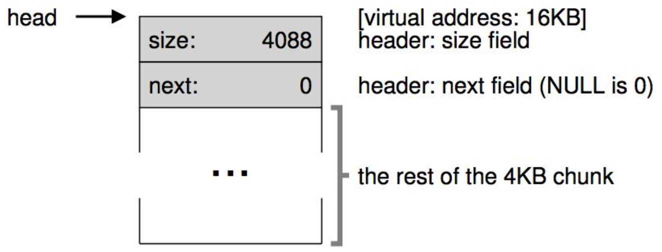


Figure 17.3: **A Heap With One Free Chunk**

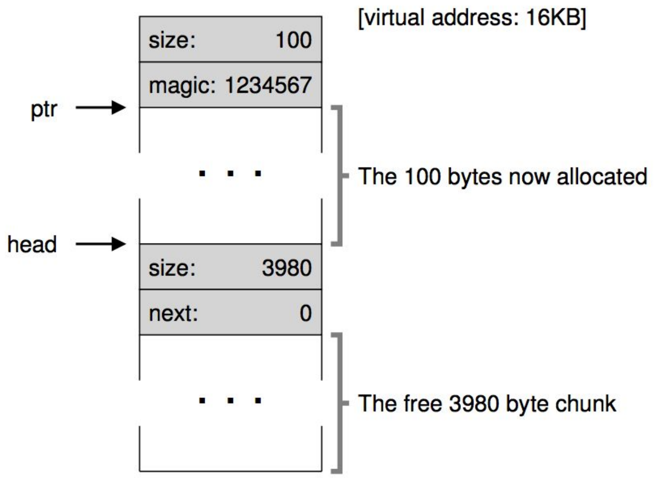


Figure 17.4: **A Heap: After One Allocation**

- Strategies for free list allocation
    - The best fit strategy is quite simple: first, search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates; this is the so called best-fit chunk (it could be called smallest fit too).
        - You waste the least amount of space with this method, but there is a performance penalty for having to do an exhaustive search for the correct block.
        - You also leave lots and lots of small chunks of free space.
    - The worst fit approach is the opposite of best fit; find the largest chunk and return the requested amount; keep the remaining (large) chunk on the free list. Worst fit tries to thus leave big chunks free instead of lots of small chunks free
        - A full search of the free space is still required though.
    - The first fit method simply finds the first block that is big enough and returns the requested amount to the user
        - Has the advantage of speed.
    - Instead of always beginning the first-fit search at the beginning of the list, the next fit algorithm keeps an extra pointer to the location within the list where one was looking last. The idea is to spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list

    - One interesting approach that has been around for some time is the use of segregated lists. The basic idea is simple: if a particular application has one (or a few) popular-sized request that it makes, keep a separate list just to manage objects of that size; all other requests are forwarded to a more general memory allocator

Chapter 18 - Paging Introduction

- To solve space management problems, you can use segmentation to chop things up into variable sized pieces or chop things up into fixed sized pieces (which is paging)
    - Paging mainly solves the problem of external fragmentation.
- Instead of splitting up a process's address space into some number of variable-sized logical segments (e.g., code, heap, stack), we divide it into fixed-sized units, each of which we call a page.
    - Physical memory is viewed as an array of fixed size slots called page frames.
- To record where each virtual page of the address space is placed in physical memory, the operating system usually keeps a per-process data structure known as a page table.
    - Provides address translations for each of the virtual pages, letting us know where in physical memory each page resides.
    - Avoids fragmentation caused by segmentation.
    - System can support abstraction of address space effectively regardless of the use of the process address space. Direction of stack and heap don't matter.
- To translate virtual addresses, we split it into two parts, a virtual page number (VPN) and an offset.

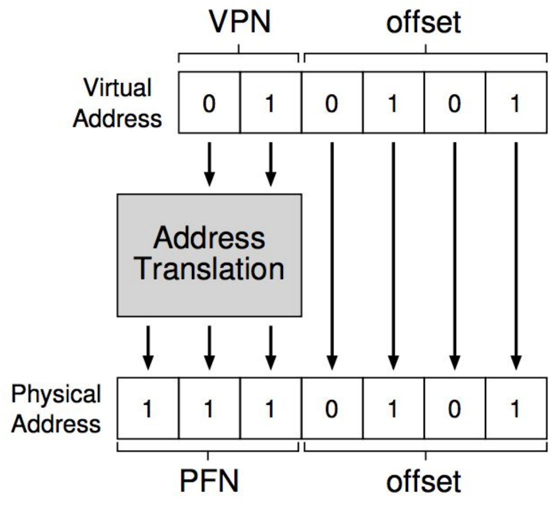


Figure 18.3: **The Address Translation Process**

- The page table is just a per process data structure that is used to map virtual addresses (or really, virtual page numbers) to physical addresses (physical frame numbers).
- Linear page table is just an array.

- The OS indexes the array by the virtual page number (VPN), and looks up the page-table entry (PTE) at that index in order to find the desired physical frame number (PFN).
- Each page table entry has a valid bit (indicate whether the particular translation is valid), protection bits (indicate whether the page could be read from/written to/executed from), present bit (whether page is in physical memory or disk), reference bit (track whether a page has been accessed).
- For any sort of instruction in assembly, the OS has to first translate the virtual address into the correct physical address, and then start loading the data.
- A linear page table is just an array of PTEs indexed by VPN; with such a structure, the entire linear page table must reside contiguously in physical memory.
  - Each page must have a fixed size and thus the page table can get really large.

Chapter 19 - Translation Lookaside Buffers

- One of the side effects of paging is that there needs to be a large amount of mapping information.
- To speed up address translation you can use translation-lookaside buffers or TLBs.
- Upon each virtual memory reference, the hardware checks the TLB first to see if the desired translation is held, and if so, we can do the operation quickly without having the consult the page table.
- If we have a TLB hit, all we have to do is extract the page frame number from the TLB entry and concatenate that to the offset of the virtual address, and thus we have our physical address.
- If we have a TLB miss, we need to access the page table to get the translation and update the TLB with the translation (for the future). This is costly because of the memory reference needed to access the page table.
- TLB hit rate = number of hits / number of accesses.
- The TLB improves performance due to spatial locality. The elements of the array are packed tightly into pages (i.e., they are close to one another in space), and thus only the first access to an element on a page yields a TLB miss.
  - Spatial locality - If a program accesses data at address x, it is likely to access data near address x
  - Temporal locality - If a program accesses data, the same time is likely to be accessed again.
- On a TLB miss, the hardware simply raises an exception, which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a trap handler. This trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special "privileged" instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction (resulting in a TLB hit).
  - When returning, we want to retry the instruction (instead of returning to the instruction after the trap - which is what happens in system calls)

- TLB commonly has a valid bit, which says whether the entry has a valid translation or not
- TLB contains virtual-to-physical translations that are only valid for the currently running process; Different processes may have different mappings, so be sure to make that note whenever there is a context switch.
- Solutions are to flush the TLB whenever you context switch or have the TLB contain process identifiers.
- If we want to add an entry to the TLB, we sometimes have to remove an entry as well. We can:
    - Evict the least recently used entry.
    - Remove a random entry.

Chapter 20 - Advanced Page Tables

- Linear page tables take up too much space sometimes.
    - One solution is to just use bigger pages, which means that you have less entries in the page tables.
        - However, this leads to internal fragmentation as there's waste within each page.
    - Hybrid approach is instead of having a single page table for the entire address space of the process, have one per logical segment. In this example, we might thus have three page tables, one for the code, heap, and stack parts of the address space.
- Unlike a linear page table, unallocated pages between the stack and the heap no longer take up space in a page table.
- You can also try using a multi-level page table.
    - The basic idea behind a multi-level page table is simple. First, chop up the page table into page-sized units; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the page directory. The page directory thus either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.
- With multi-level, we add a level of indirection through use of the page directory, which points to pieces of the page table; that indirection allows us to place page-table pages wherever we would like in physical memory.
    - The one problem is that on a TLB miss, two loads from memory will be required to get the right translation information from the page table (one for the page directory, and one for the PTE itself), in contrast to just one load with a linear page table.
- Inverted page tables is where instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system. The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.

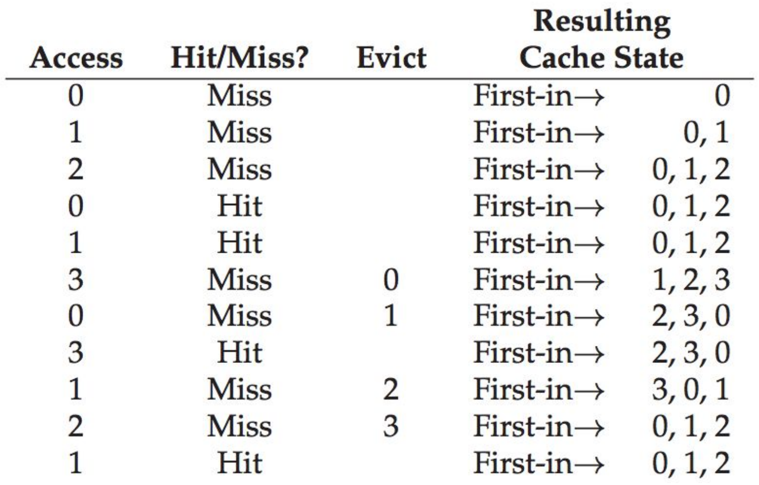
- Process of Execution
    - Extract VPN => Check if in TLB => TLB Hit
    - Extract VPN => Check if in TLB => TLB Miss => Use PBTR to find page table => Lookup PTE from VPN => If page in memory, extract PFN from PTE => install into TLB =>retry instruction => TLB Hit
    -

Chapter 21 - Swapping Mechanisms

- Accessing more memory than is physically present within a system, to do so requires more complexity in page-table structures, as a present bit (of some kind) must be included to tell us whether the page is present in memory or not. When not, the operating system page-fault handler runs to service the page fault, and thus arranges for the transfer of the desired page from disk to memory, perhaps first replacing some pages in memory to make room for those soon to be swapped in.

Chapter 22 - Swapping Policies

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages. Deciding which page (or pages) to evict is encapsulated within the replacement policy of the OS;
- Simplest policy is just FIFO



Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		First-in→	0
1	Miss		First-in→	0, 1
2	Miss		First-in→	0, 1, 2
0	Hit		First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2
3	Miss	0	First-in→	1, 2, 3
0	Miss	1	First-in→	2, 3, 0
3	Hit		First-in→	2, 3, 0
1	Miss	2	First-in→	3, 0, 1
2	Miss	3	First-in→	0, 1, 2
1	Hit		First-in→	0, 1, 2

- Problem with FIFO is that it will kick out a block (since it came first) even if it gets hit a bunch of times.
- The Least-Frequently-Used (LFU) policy replaces the least-frequently used page when an eviction must take place.
- Similarly, the Least-Recently Used (LRU) policy replaces the least-recently-used page.
- The above 2 are historical algorithms which need some sort of data structure to keep track of all the information for each table.
- To support this behavior, the hardware should include a modified bit (a.k.a. dirty bit). This bit is set any time a page is written.

- On demand paging is when the page is needed, it is demanded out of disk and into memory.
- Prefetching is when you predict if a page is necessary before actually calling to it.
- Clustering is when you gather a list of pages that need to be written to disk and then do it in one go.
- Thrashing is when the memory demands of running processes exceeds the available physical memory, which results in constant paging.

<u>**Chapter 26 - Concurrency**</u>

- A thread is an abstraction for a single running process.
- A multi-threaded program has more than one point of execution.
- Each thread is like a separate process except it shares the same address space and can thus access the same data.
- Each thread has a program counter to keep track of where the program is getting instructions from. It also has a private set of registers.
- When switching from one thread to another, a context switch must take place, which saves register state of T1 and restores state of T2.
- With processes, the saved state was in a process control block.
- With threads, it gets saved in
- a thread control block.
- The big difference with threads is that the address space remains the same.
- There will be one stack per thread in the address space.
    - Any variables or parameters or return values will be placed in thread-local storage, aka the stack of the relevant thread.
- The main reasons to use threads are because they allow you to do parallelization, and that they avoid blocking program progress due to slow I/O. While one thread in the program waits, you can have the other threads doing other computation. While you could use multiple processes, the plus side of threads is that since they share an address space, it’s easy to share data. Use processes when little sharing of data structures in memory is needed.
- You can use pthread_create to create a thread and then use pthread_join to wait for a particular thread to complete.
- The issue when you create threads is that the output of what thread finishes first and outputs what is nondeterministic. The scheduler is in charge of deciding the order in which to run.
- A race condition is when the results depend on the timing execution of the code.
- A critical section is a piece of code that accesses a shared variable and must not be concurrently executed by more than one thread. Mutual exclusion is a property that says that if one thread is executing within a critical section, then others will be prevented from doing so.
- One way to fix this is also to make an instruction atomic. That means that while the execution is executed, it performs the update as desired and nothing can interrupt it.

Chapter 27 - Thread API

- Just know how to correctly use pthread_create and pthread_join.
- Providing mutual exclusion to critical sections is done via locks.
- If no other thread holds the lock when pthread mutex lock() is called, the thread will acquire the lock and enter the critical section. If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock (implying that the thread holding the lock has released it via the unlock call).
- All locks have to be properly initialized
  - pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
- The trylock function returns failure if the lock is already held.
- The timedlock version returns after a timeout or after acquiring the lock, whichever happens first.
- Threads also have condition variable functions.
  - Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue.
- Pthread_cond_wait puts the calling thread to sleep and waits for another thread to signal it, usually when something in the program that the system cares about has changed.
  - When the thread gets put to sleep, it releases the lock that it currently has.
- If one thread calls Pthread_cond_wait(&cond, &lock), then it must wait until a different thread calls Pthread_cond_signal(&cond) to signal to the first thread that it should stop waiting.
  - Before the first thread reawakens, it must re-acquire the lock.

Chapter 28 - Locks

- Locks, at their most basic level, make sure that critical sections execute as if they were a single atomic instruction.
- The lock variable itself holds the state of the lock at any time.
  - It can either be available (unlocked or free) or acquired (locked or held). Exactly one thread holds the lock and is in that critical section.
- Calling the routine lock() tries to acquire the lock; if no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section
  - If another thread tries to call lock, it won’t return while the lock is being held by that other thread.
- The owner of the lock has to call unlock for the lock to become available.
- Another word for lock is a mutex, which is a way of saying that it provides mutual exclusion between threads.
- A coarse-grained locking strategy is when one big lock is used any time a critical section is accessed.
- A better way to go about this would be to use a fine-grained approach where you have different locks for protecting different data and data structures.

- You can evaluate locks for looking at 3 things
    - 1. Does it actually provide mutual exclusion? Aka does it prevent multiple threads from entering a critical section at the same time.
    - 2. Is it fair? Aka does each thread contending for a lock get it's fair share at acquiring it once it becomes free
    - 3. Is it good performance-wise? Aka is there a lot of overhead created by using locks (the whole process of acquiring and releasing locks).
- To implement a lock, you could basically just disable interrupts. This makes sure the once a thread enters a critical section, it won't get interrupted by anything and will execute the instructions basically atomically.
    - However, the downside is that you have to trust the program not to do anything bad when there are no interrupts. Turning interrupts on and off is a privileged operation and we don't want to give the programs too much power.
    - Also this doesn't work on multiple processors because threads on different processors could enter code at the same time and it won't really matter if interrupts are disabled or not.
    - Also, not being able to know if an interrupt has taken place is a bad thing and you can't have interrupts disabled for too long.
- Spin waiting is when you endlessly just keep checking what the value of the lock is (whether it's held or not). This is also considered polling (I think)
- The biggest thing when creating a lock implementation is that you want to have instructions executed atomically. You want to be able to check the value of the lock (and set it to another value if necessary) all in a single atomic instruction.
- A spin lock is a lock that simply spins, using CPU cycles, until a lock becomes available.
    - If you want to use a spin lock though, you have to have a preemptive scheduler which makes sure to interrupt the thread via a timer, so that other threads get run. If you don't have this, then the thread that is spinning will never let go off the thread.
- Spin locks don't provide fairness guarantees because a spinning thread may spin forever and this could lead to starvation for the other threads.
- The compare and swap instruction tests whether the value at the address of ptr is equal to expected; if so, update the memory location pointed to by ptr with the new value. If not, do nothing.
    - int CompareAndSwap(int *ptr, int expected, int new)
- Instead of just spinning and waiting for the lock's value to change, a thread can choose to yield, and give up the CPU instead.
    - While we are saving CPU cycles by just giving up the thread immediately instead of spinning, we are adding a substantial overhead in the context switching cost of waking up and sleeping a thread to sleep.
- You can also have 2 phase locks where if the lock is not acquired in the first spin phase, it will then put the thread to sleep, only to be woken up when the lock becomes free later.

- Want to avoid receive livelock ... aka you have too many requests at a single time and end up not answering any sicne you are so busy!
  - this happens because as you start to handle something, you get an interrupt and you go there, and so on so forth and nothing ends up happening
  - a good solution is to stop the interrupt handler when you fill the buffer and keep it stopped till the buffer is half empty or so
    - this way you can still try to handle as many as you can and its only ever problem if too many requests at once

Chapter 29 - Locked Data Structures

- Adding locks to a data structure makes it thread safe.
- The basic way to make programs thread safe is just to add a single lock, which is acquired when calling a routine that manipulates the data structure, and is released when returning from the call.
- Perfect scaling is when you can get your program to complete threads just as quickly on multiple processors as a single thread does on one. Everything is done in perfect parallel.
- Hand over hand locking is done with linked lists where instead of having a single lock for the entire list, you instead add a lock per node of the list. When traversing the list, the code first grabs the next node's lock and then releases the current node's lock.
  - It's nice, but the overheads of acquiring and releasing the locks are just too much.
- Concurrent queues use an approach where you have 2 locks, one for the head of the queue and one for the tail.
- Concurrent hash tables uses a single lock per hash bucket.
- A big lesson is that enabling more concurrency doesn't increase performance because of the possibility of added overhead through using the locks too much.

Chapter 30 - Condition Variables

- Besides locks, a thread can also check whether a condition is true before continuing execution.
- A condition variable is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition); some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition).
- To declare such a condition variable, one simply writes something like this: pthread cond t c;, which declares c as a condition variable (note: proper initialization is also required). A condition variable has two operations associated with it: wait() and signal(). The wait() call is executed when a thread wishes to put itself to sleep; the signal() call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition.
- When calling signal or wait, you have to hold the lock.

Chapter 31 - Semaphores

- A semaphore is an object with an integer value that we can manipulate with sem_wait() and sem_post().
- We must always initialize this value with something before using it.

Chapter 32 - Concurrency Bugs

- Just know how to correctly use pthread_create and pthread_join.

Chapter 38 - Redundant Disk Arrays (RAID)

- RAID is a technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system.
- Using multiple disks in parallel can greatly speed up I/O times
- Large disks are obviously useful for large data
- Spreading data across multiple disks (without RAID techniques) makes the data vulnerable to the loss of a single disk; with some form of redundancy, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.
- RAID Level 0: Striping
    - Spread the blocks of the array across the disks in a round-robin fashion.
- RAID Level 1: Mirroring
    - With a mirrored system, we simply make more than one copy of each block in the system; each copy should be placed on a separate disk, of course. By doing so, we can tolerate disk failures.
    - It’s expensive but generally no data loss.
    - Gotta deal with consistency issues in regards to writing to both disks and making sure that they’re both identical to each other.
        - Can use write ahead logs and recovery procedures.
- RAID Level 4: Saving Space With Parity
    - Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems.
    - For each stripe of data, we have added a single parity block that stores the redundant information for that stripe of blocks
- RAID Level 5: Rotating Parity
    - RAID-5 works almost identically to RAID-4, except that it rotates the parity block across drives

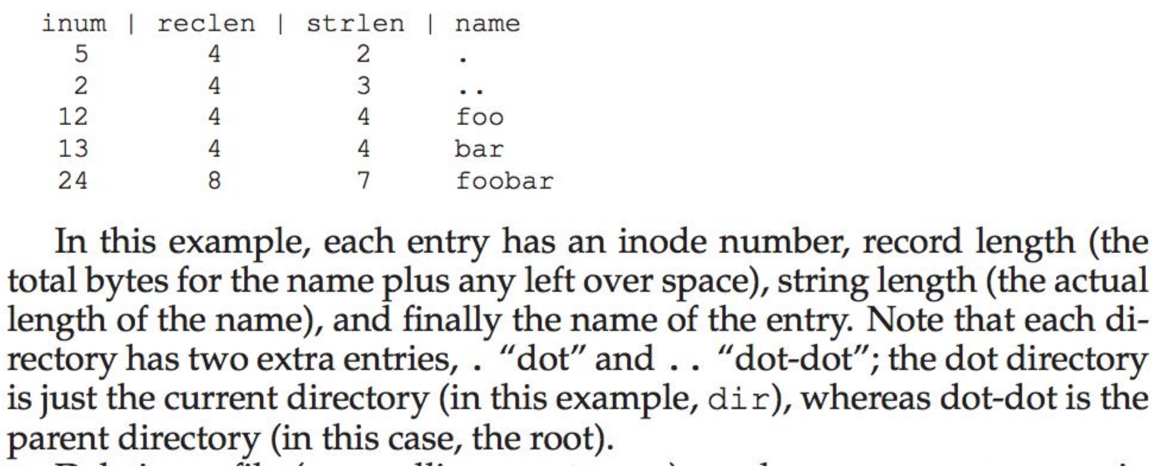
Chapter 39 - Files and Directories

- Persistent storage stores information permanently. It’s non-volatile so it keeps it's contents even when power is lost.
- A file is a linear array of bytes from which you can read or write.
    - The low level name of a file is referred to as its inode number.

- A directory also has an inode number and it contains a list of pairs (user-readable names, low level names).
- Each entry in a directory refers to files or other directories.
- You can create files with the open system call (and by passing the O_CREAT flag).
- The command “ln file file2” creates another name in the directory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file.

Chapter 40 - File System Implementation

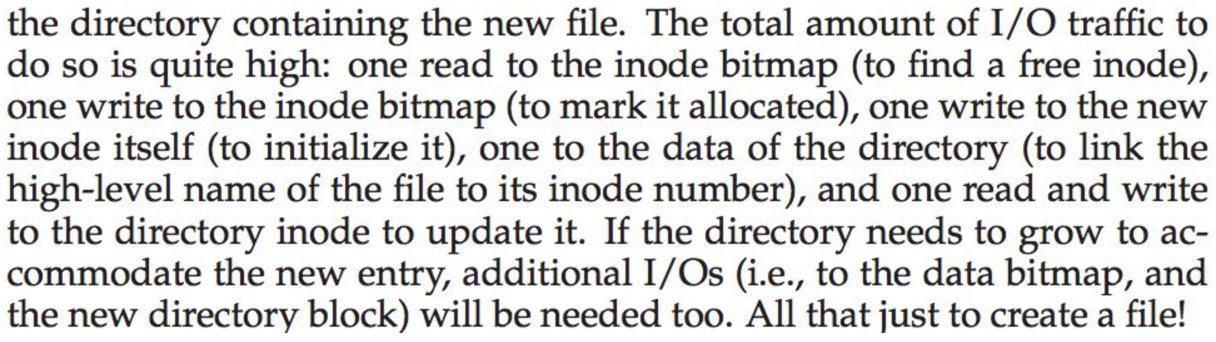
- Each file system has some sort of on disk organization.
- Each disk is divided into blocks normally of size 4 KB.
- Region of the disk that we use for user data is the data region.
- Information about each file is known as metadata and also has to be stored.
- This information is stored in inodes, which are stored in an inode table.
    - Each inode contains info about file type, file size, the number of blocks allocated to the file, permission info, and time info. All this is called metadata.
    - One of the important parts of the inode is that it needs to refer to the locations of data blocks. One approach is just to have a direct pointer to the disk block that contains the file. However, if the file is bigger than the block size, then this won’t work.
    - Indirect pointers are better because they are able to point to a block that contains more pointers to user data.
    - You could also have double indirect pointers where you have a pointer to a block that contains pointers to indirect blocks.
- When we create a file, we allocate an inode for that file.
    - The file system will search through the bitmap to find an inode that is free.
- A bitmap is a data structure that basically uses each bit to indicate whether the corresponding block/object is free (0) or in use(1).
    - There are bitmaps for both the data region and one for the inode table.
- A superblock is a one block region on disk which contains information about the file system itself, such as how many inodes and data blocks there are within the fs.
- When mounting a file system , the OS reads the superblock first to initialize parameters and then attaches the volume to the fs tree.
- Linux ext2 is a commonly used file system.
- Over the years, we’ve learned the following things.
    - Most files are small, average file size is growing, file systems contain lots of files, directories are typically small.



```
inum | reclen | strlen | name
  5       4        2      .
  2       4        3      ..
 12       4        4      foo
 13       4        4      bar
 24       8        7      foobar
```

In this example, each entry has an inode number, record length (the total bytes for the name plus any left over space), string length (the actual length of the name), and finally the name of the entry. Note that each directory has two extra entries, . "dot" and .. "dot-dot"; the dot directory is just the current directory (in this example, `dir`), whereas dot-dot is the parent directory (in this case, the root).

- File systems treat directories as special types of files.
- Every directory has an inode somewhere in the inode table.
- File system must keep track of what inodes and data blocks are free.
- If possible, file systems want to store a file with contiguous blocks of memory, b/c it helps performance.

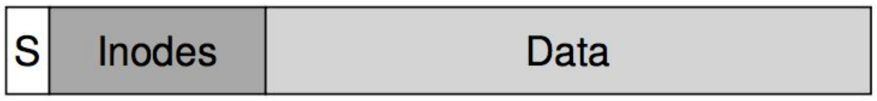


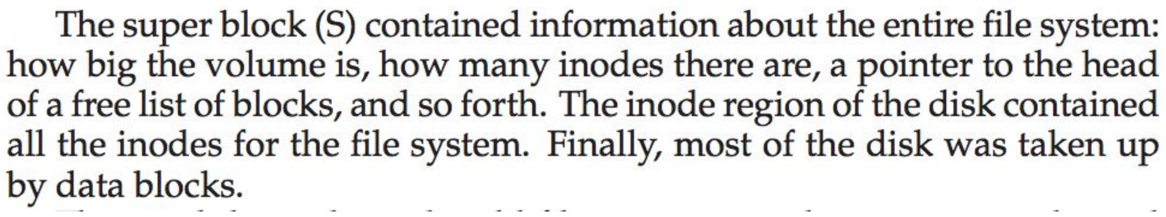
the directory containing the new file. The total amount of I/O traffic to do so is quite high: one read to the inode bitmap (to find a free inode), one write to the inode bitmap (to mark it allocated), one write to the new inode itself (to initialize it), one to the data of the directory (to link the high-level name of the file to its inode number), and one read and write to the directory inode to update it. If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too. All that just to create a file!

- To help performance, most file systems use system memory (DRAM) to cache important blocks.
    - Fixed size cache to hold the important blocks.
- The first open to a file may generate a lot of I/O traffic, but after that, opens to those files will hit the cache and will be much faster.

<u>Chapter 41 - Fast File System (FFS)</u>

- First basic file system was pretty much like this.

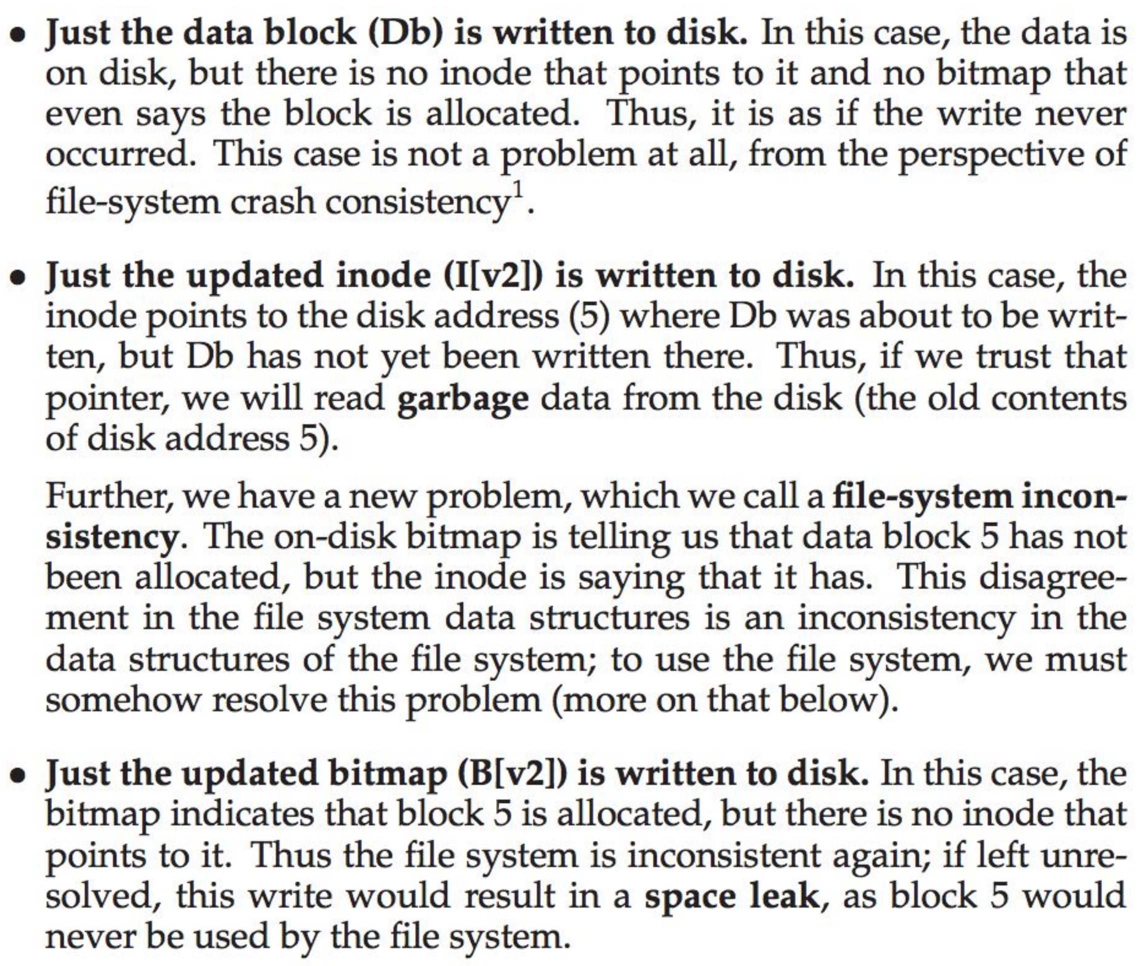



The super block (S) contained information about the entire file system: how big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth. The inode region of the disk contained all the inodes for the file system. Finally, most of the disk was taken up by data blocks.

- Problem was that data was spread all over the place and there was too much overhead in finding and accessing this data.
- A file system gets fragmented when free space is not carefully managed.
- The recurring problem is that once files get deleted and newer ones take their places, the newer ones aren't in contiguous locations. They are spread out across any free space left by the deletion of previous files.
- Internal fragmentation is waste of space within a block. If you have 4 KB blocks, but most of your files are 2 KB or lower, that's internal fragmentation.
  - Solution to this is just to use sub blocks.
- Fast FIle System (FFS) divides the disk into cylinder groups to help the previous fragmentation problem.
- Each cylinder group contains a superblock, inode bitmap, data bitmap, inodes, and data.
- Basically, what FFS wants to do is keep related stuff together and keep unrelated stuff apart. The related stuff will go in the same block group.
- Process of reducing an overhead by doing more work per overhead paid is called amortization

Chapter 42 - Crash Consistency: FSCK and Journaling

- A big job of file systems is that they have to be persistent. They have to keep working in the face of power outages and operating system bugs.
- One of the big problems is the crash-consistency problem where if a system crashes during an operation to a data structure, the structure is left in an inconsistent state.
- For example, to append something to a file, you have to write to update the inode (to change the size), the bitmap (to specify that there is another data block being used), and the data block itself needs to be written.
- 3 Situations are possible if only one of the writes ends up happening.



- **Just the data block (Db) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency[1].
- **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).

  Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. This disagreement in the file system data structures is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).
- **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

- FIle-system inconsistency is when the bitmap is telling us that the data block hasn't been allocated when the inode says that it has.
- Space leak is when the bitmap says that a space is being used, when in fact, it is really free.
- Ideally, we want the to move the file system from one consistent state to another atomically.
- Fsck is a tool for finding data inconsistencies, and then fixing them. The only thing it can really do is fix inconsistencies between the inode and the bitmap.
- The problem is that this process is way too slow. Scanning the entire disk to make sure everything is right takes days.
  - Analogy is dropping your keys in the hallway and then doing a full search-the-entire house algorithm to find them. It works, just terribly slow.
- Besides fsck, the other option is journaling (or write-ahead logging)
  - The idea is that before you write to the structures, write down a note somewhere describing what you're gonna do.
  - This way, if the system crashes, you'll know where specifically problems may have occurred.

- Journalling is in Linux ext3, but not in ext2
- A minor issue is that some files may get written twice if a system crashes during a journaling checkpoint.

Chapter 43 - Log-structured File Systems

- When writing to disk, LFS first buffers all updates in an in-memory segment. When the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather always writes segments to free locations.
- This basic idea, of simply writing all updates (such as data blocks, inodes, etc.) to the disk sequentially, sits at the heart of LFS.
- Before writing to the disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them to disk all at once, thus ensuring efficient use of the disk. This is write buffering.
    - The large chunk of updates LFS writes at one time is referred to by the name of a segment.
- The designers of LFS introduced a level of indirection between inode numbers and the inodes through a data structure called the inode map (imap). The imap is a structure that takes an inode number as input and produces the disk address of the most recent version of the inode.

Chapter 44 - File Integrity

- Two types of single-block failures are common and worthy of consideration: latent-sector errors (LSEs) and block corruption.
- LSEs arise when a disk sector (or group of sectors) has been damaged in some way. F
- There are also cases where a disk block becomes corrupt in a way not detectable by the disk itself.
- Latent sector errors are rather straightforward to handle, as they are (by definition) easily detected. When a storage system tries to access a block, and the disk returns an error, the storage system should simply use whatever redundancy mechanism it has to return the correct data
    - In a mirrored RAID, for example, the system should access the alternate copy.
    - In a RAID-4 or RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group.
- The primary mechanism used by modern storage systems to preserve data integrity is called the checksum. A checksum is simply the result of a function that takes a chunk of data (say a 4KB block) as input and computes a function over said data, producing a small summary of the contents of the data (say 4 or 8 bytes).
    - The goal of such a computation is to enable a system to detect if data has somehow been corrupted or altered by storing the checksum with the data and then confirming upon later access that the data's current checksum matches the original storage value.

- Another error is a misdirected write. This arises in disk and RAID controllers which write the data to disk correctly, except in the wrong location.
    - Answer is to add physical identifiers to the checksum.
- Another issue is a lost write, which occurs when the device informs the upper layer that a write has completed but in fact it never is persisted; thus, what remains is left is the old contents of the block rather than the updated new contents.
- Disk scrubbing is periodically reading through every block of the system, and checking whether checksums are still valid

Chapter 47 - Distributed Systems

- By collecting together a set of machines, we can build a system that appears to rarely fail, despite the fact that its components fail regularly.
- The simple technique of making sure that the server receives messages from the client is acknowledgment, or ack for short. The idea is simple: the sender sends a message to the receiver; the receiver then sends a short message back to acknowledge its receipt.
    - If the sender doesn't get an acknowledgement, then it waits for some time and then retries.
- With a sequence counter, the sender and receiver agree upon a start value (e.g., 1) for a counter that each side will maintain. Whenever a message is sent, the current value of the counter is sent along with the message; this counter value (N) serves as an ID for the message. After the message is sent, the sender then increments the value (to N + 1).
- Distributed shared memory (DSM) systems enable processes on different machines to share a large, virtual address space
- Remote procedure call packages all have a simple goal: to make the process of executing code on a remote machine as simple and straightforward as calling a local function even though there's a lot of client server interaction going on in the background.
- Specifically, typical RPCs are made synchronously, i.e., when a client issues the procedure call, it must wait for the procedure call to return before continuing.
- When an asynchronous RPC is issued, the RPC package sends the request and returns immediately; the client is then free to do other work, such as call other RPCs or other useful computation

Chapter 48 - Network File System (NFS)

- This simple goal is realized in NFSv2 by designing what we refer to as a stateless protocol. The server, by design, does not keep track of anything about what is happening at each client.
    - Rather, each client operation contains all the information needed to complete the request.
- File handles are used to uniquely describe the file or directory a particular operation is going to operate upon; thus, many of the protocol requests include a file handle.

- The heart of the design of crash recovery in NFS is the idempotency of most common operations.
    - Helps with lost and corrupted commands because you can just resend again without worrying.

Principles of Computer System Design

Didn't take as many notes on this textbook since I found 3 Easy Pieces a lot easier to follow along with

Chapter 4

- **Soft modularity** is when modules are correctly limited in their interaction but implementation errors can cause interactions that go outside their interfaces.
- To enforce modularity, sometimes you have to use hard boundaries where errors can't propagate from one module to another.
- The **client/service organization** provides hard modularity because errors can propagate only with messages, and clients can check for errors by just considering the messages.
- **Stack discipline** is when each invocation of a procedure must leave the stack as it found it.
    - There must be convention for who saves what registers, who puts arguments on stack, who removes them.
    - This convention is called the procedure calling convention.
- Caller/Callee relationship (which is soft modularity) can go sour if there is a problem in the callee that corrupts the caller's area of the stack.
    - Callee can also return somewhere that the caller doesn't want it to go.
    - Callee can store return values in different registers, not R0.
- **Fate sharing** is if the callee divides by zero for example, then the callee terminates, and therefore the caller may terminate as well.
- High level languages are a bit more helpful in enforcing modularity because it's compile and run time system perform all stack and register manipulation in accordance with the procedure calling convention.
- Enforced modularity is that that is enforced by some external mechanism.
- Client is a module that initiates a request. It builds a message that contains all the nexessary data and then sends it to a service to carry out the job. It has to extract the arguments, execute the operations, adn build a response message.
- An instance of a service running on a single computer is called a **server**.
- In order to send this package of operations and arguments, you have to convert it to a certain representation. This process is called marshaling. This is the reason that clients and services don't have to worry about writing to the same registers, return address, etc.
- Client/service organization not only separates functions (functional modularity), but it enforces it (through enforced modularity)

- Clients can protect themselves against services that fail to return.
- If a service fails, the client has a controlled problem. It knows that there's something wrong with that particular service, but not with the rest of it's program.
- Client/service organization is a sweeping simplification because it eliminates all forms of interaction other than messages.
    - Only way to go from client to service and vice versa is through well defined messages.
- It is the job of a good system designer to define a good interface between client and service so that errors don't propagate easily.
- Examples of client/service is a web browser being the client and the website being the service.
- When a service has multiple clients, you should have a trusted intermediary, which is a service that functions as the trusted third party for multiple clients. It can control shared resources in a controlled manner.
- A remote procedure call is a style of client/service interaction where each request is followed by a response.
    - It wants to look as close to an ordinary procedure call as possible.
- A **stub** is a procedure that hides the marshaling and communication details from the caller and callee.
- RPC's can reduce fate sharing between caller and callee by exposing failures of the callee to the caller so that the caller can recover.
    - Compared to ordinary system calls though, they introduce new failures and take a lot more time.
- In RPCs, there is a special thing called no response failure. It is when there is no response from the service and the client can't tell whether some failure occurred before the service even had a chance to be run or if the service performed the action and then the failure occurred.
- At least once RPC: If client stub doesn't receive response, keep trying as many times as necessary. For the services where the result is the same no matter how many times the request is made, those services are idempotent.
- At most RPC: If the client stub doesn't receive a response within some time period, then the stub returns an error to the caller.
    - Makes sure that either 0 or 1 requests get made.
    - More requests than that can be mad for money transfers or stuff like that.
- Exactly once RPC: If client doesn't receive a response, it will request the service to ask about the status of the request.

Chapter 5

- The client/service organization in the previous chapter required multiple computers. However, we also need ways to split a system into n modules and be able to run these modules without resorting to soft modularity.
    - Virtualization is the best way to achieve this.

- We create several virtual computers and then execute each module in it's own virtual computer.
- A program that virtualizes something stimulates the interface of the physical object, but creates many virtual objects through multiplexing one physical instance. Aggregating is the opposite. It’s when you provide one large virtual object by aggregating many physical instances.
- An example is the types of modules you have on a computer: text editor, keyboard manager, email reader, etc
- We want these modules to interact with each other so we present each module with it's own virtual computer, which executes independently from the other virtual computers.
- First step in virtualizing a computer is to virtualize the processor. To provide the editor module with a virtual processor, we create a thread, which is an abstraction that encapsulates the execution state of an active computation.
- You can stop and start threads, which provides a good way for the OS to multiplex the physical processor.
- The job of the thread manager is to multiplex the many possible threads on the limited physical processors in such a way that a programming error in one thread can’t interfere with the execution of another thread
- Threads have their own virtual memory.
- To allow client and service modules to communicate, you can use a bounded buffer of messages.
    - A thread can invoke SEND which inserts the message into a bounded buffer of messages.
- Emulation is a way of proving an application with it's own instance of physical hardware.

Locks

- The difference between a spin lock and a mutex lock is that a spin lock keeps checking the lock (busy waiting) to see if it’s been unlocked while the mutex puts threads waiting for the lock to sleep (blocking)

Other Stuff

- The difference between a system call and a procedure call is that system calls contain the trap instruction. This handles the part of giving the process certain control over those restricted operations in the OS.
- At boot time, the kernel is responsible for a lot. It has to set up the trap table, it has to make sure that interrupt handlers are in the right place.
- Atomicity is executing a sequence of steps that they appear to be done in one single, indivisible step. This step can be completed halfway, it’s all or nothing.
- Concurrent actions have the before and after property if their effect from the POV of the invokers is the same regardless of which action occurred completely before or after the other.

    - You want actions to happen completely before or after one another. You don't want the situation where action 1 completes halfway and then action 2 is reading from the same memory and it gets thrown off too.
- An action is atomic if there is no way for a high layer to discover the internal structure of its implementation
- Sched_yield() causes the calling thread to relinquish the CPU.
- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.
- Threads in the same process share:
    - Process instructions
    - Most data
    - open files (descriptors)
    - signals and signal handlers
    - current working directory
    - User and group id
- Each thread has a unique:
    - Thread ID
    - set of registers, stack pointer
    - stack for local variables, return addresses
    - signal mask
    - priority
    - Return value: errno
- pthread functions return "0" if OK.
- An operation is called idempotent when the effect of performing the operation multiple times is equivalent to the effect of performing the operating a single time.
- You can’t have a process ID of more than 65535 (not 2^16 or more)
- Policies (used to answer questions such as which process should run at a particular instant) and Mechanisms (The actual OS implementation of this scheduling).
- Abstraction is the separation of interface from internals of specification from implementation.
- Static libraries are a collection of object files (ending with a .a) and are linked during the compilation process and the entire code base is added to the final executable. This saves the need for recompilation but the final executable is much larger.
- Dynamically linked libraries are loaded when the program starts.
- Dynamically loaded libraries are libraries that are loaded into the program only when the program needs it.
- **Deadlock**: A situation in which two or more processes are unable to proceed because each is waiting for one the others to do something.
- **Livelock**: A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work

    - Two threads, trying and failing to acquire a lock
- Interrupts wait in a buffer, but traps don’t. Traps are caused by the CPU, and interrupts are caused by the I/O Bus.
- Limited Direct Execution just defines how the OS prepares the program running and hot it exits and deals with the program when it’s finished.
- When a child exits, some process must `wait` on it to get its exit code. That exit code is stored in the process table until this happens. The act of reading that exit code is called "reaping" the child. Between the time a child exits and is reaped, it is called a zombie. Zombies only occupy space in the process table.