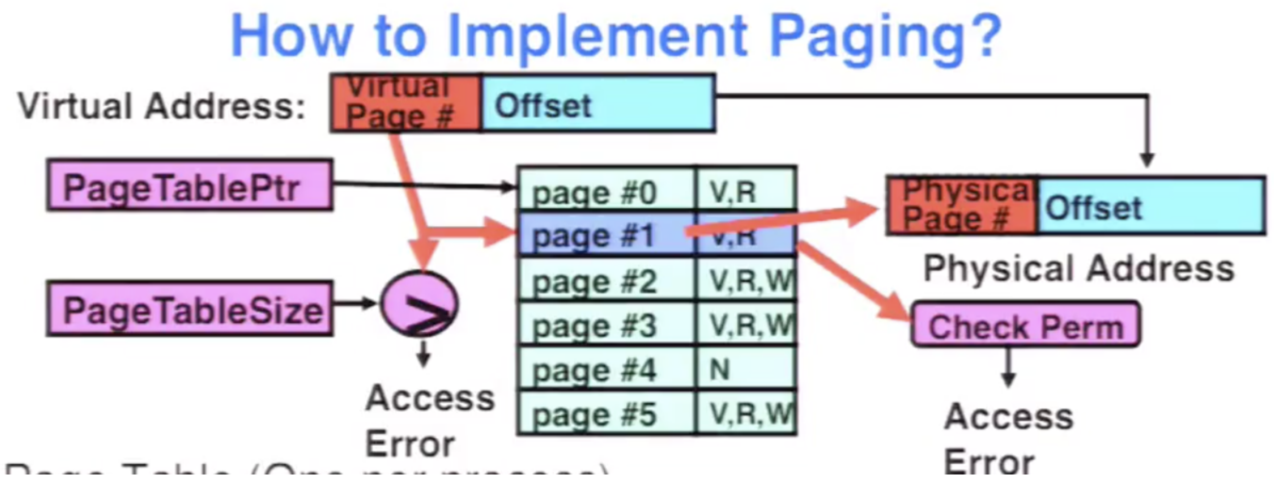
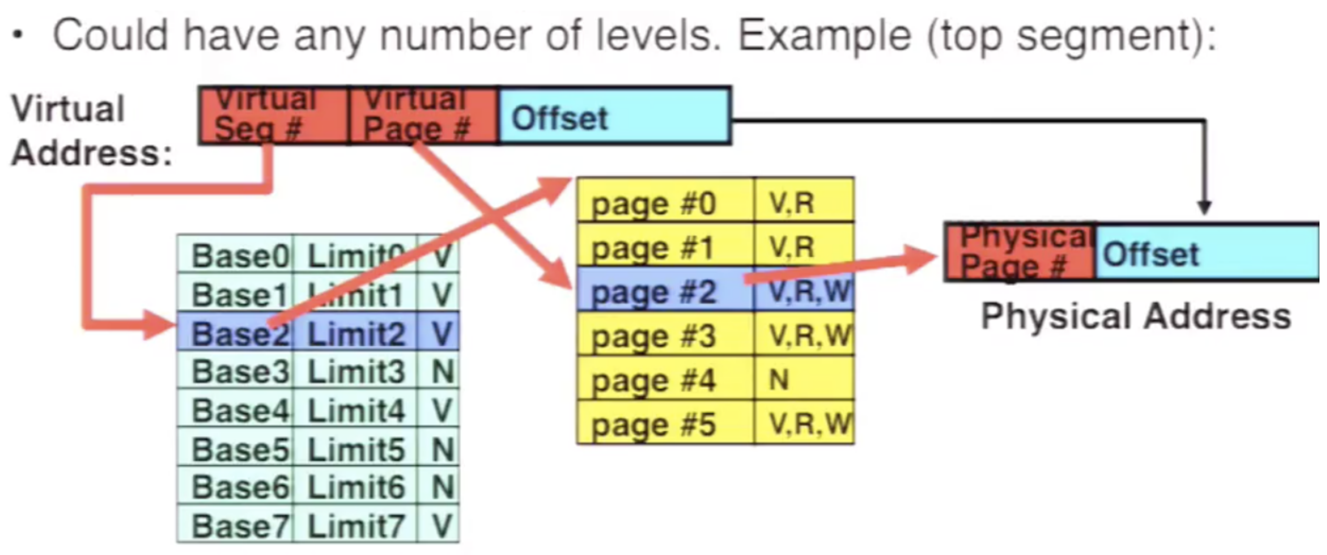
virtual memory and user/kernel space

- virtual process makes processes thinks they are the only one and they own everything. but we need to map this view onto real physical memory and do protection etc. this is the virtual memory system.
- if there were no translation process like this, think about the fact that compiled code would need to have physical locations hardcoded in it. hard to do!
- linking vs loading
    - linking involves replacing all the function calls with actual function references etc
    - loading is in the case of shared libraries: once a program is loaded, if it references a shared library, you have to check if that shared library has already been loaded, whats its address or load it if it hasn't been loaded because no other program is using it yet
- the hardware that helps do this translation is collectively called the memory management unit (MMU)
- to translate from virtual to physical addresses, early computers did simple things like adding a fixed offset to virtual to get physical address
    - this method offers protection from other processes accessing a process' data
- another approach is to split a virtual address into segment + offset. the segment maps to a segment in physical memory. we will then ensure that the offset is less than some limit (this limit is determined by how much memory starting at that physical segment is owned by the process. we will only allow the access if the memory is translated physical address is owned by the process, as determined by the limit.
    - the problem with this segments approach is that it creates a lot of fragments
- what is done in modern systems: **paging**. described below.
- paging involves making all segments (called a page) the same size → big enough so that the management table to keep track of state is small and small enough that fragmentations and swapping it between memory and disk is fast enough.
- implementing pages. the table below is called a page table.



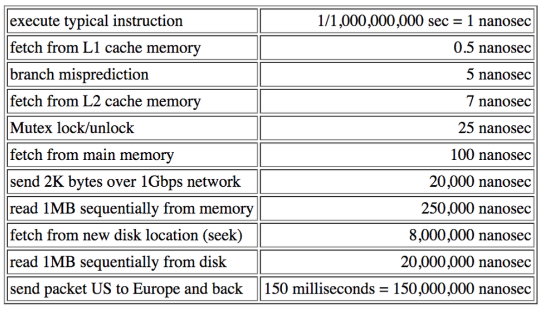
- a page size of 1K and 32 bit addresses, this means 4 million page table entries! that's a lot of memory just for a page table of 1 process. we'd like to make this sparse so as to only contain pages that are actually allocated by the process instead of a naive page table entry for all page tables. in a 64 bit computer, this would get really big!

- instead we can have a multi layer page table as below. only the virtual segments that are allocated need a corresponding page table.

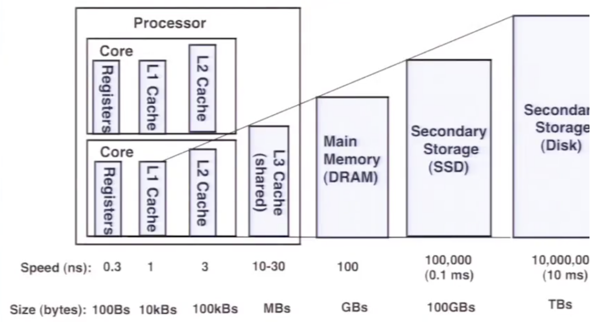


- Could have any number of levels. Example (top segment):

- note that in the above picture, the first level is a segment (which means it can be variable sized, as determined by the limit column entry). a page, on the other hand, is fixed size → determined by the number of bits allocated to the virtual page number part of it.
- in modern systems, we don't have segments but instead we have fixed page sizes at each level → so no need for the limit column anywhere.
- if required, we can divide this up into more levels in the tree, we just have to have corresponding splits in the virtual address → called 2-level paging, 3-level paging etc
- remember, page tables are also stored in memory. this is ofcourse bad to go to a page table in memory to access physical memory that might be cached → later we will discuss caching the page table lookups as well.
- good time to put jeff dean's numbers and the cache summarization diagram on the right

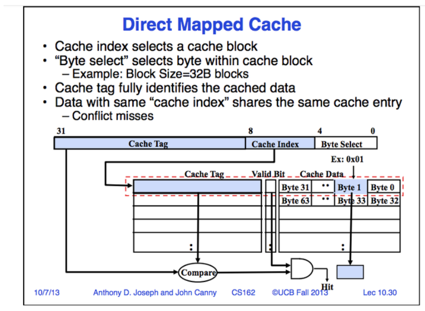


execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec



this has a logarithmic scale, not linear

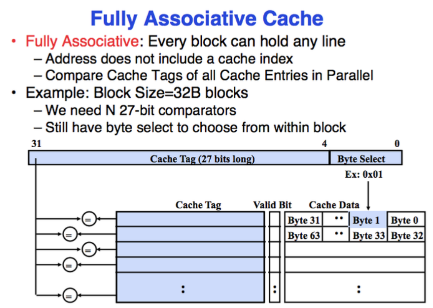
- the above table suggests memory is much worse than cache access. so we need caches! lets introduce them into the picture.
- caching should help take care of temporal locality and spatial locality
- cold start means how initially you might have a lot of cache misses if the cache is empty (or has irrelevant stuff)
- 3 varieties of caches



Direct Mapped Cache

- Cache index selects a cache block
- "Byte select" selects byte within cache block
  - Example: Block Size=32B blocks
- Cache tag fully identifies the cached data
- Data with same "cache index" shares the same cache entry
  - Conflict misses

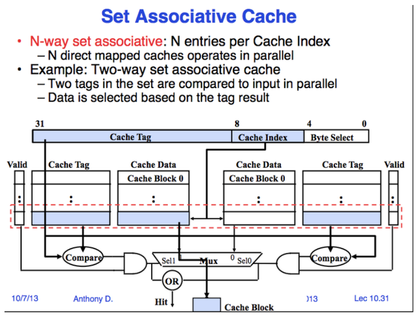
store exactly in 1 location (lots of cache misses)



Fully Associative Cache

- Fully Associative: Every block can hold any line
  - Address does not include a cache index
  - Compare Cache Tags of all Cache Entries in Parallel
- Example: Block Size=32B blocks
  - We need N 27-bit comparators
  - Still have byte select to choose from within block

can store anywhere (needs a lot of hardware)

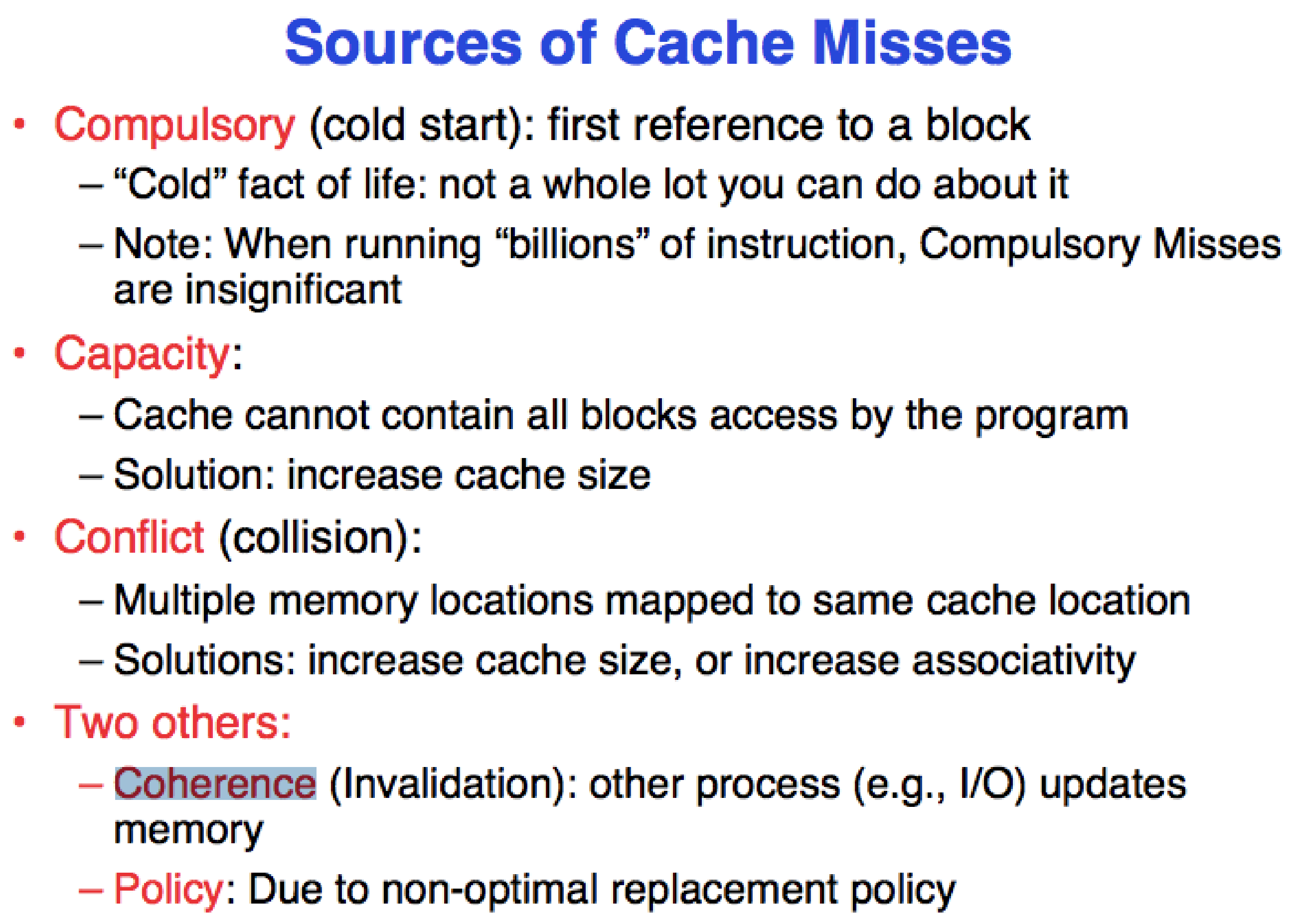


Set Associative Cache

- N-way set associative: N entries per Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Two tags in the set are compared to input in parallel
  - Data is selected based on the tag result

can store in n places (a tradeoff approach)

- the cache contents are fetched a few bytes at a time instead of byte wise, to take advantage of spatial locality → so when you write code it should take advantage of this.
- cache replacement policies:
  - random
  - LRU
  - LFU → least frequently used
- two strategies while writing to cache
  - write thru: update cache + lower levels of memory
  - write back: only write to cache, mark as dirty and on eviction, update lower level.
- TLB is translation lookaside buffer is basically a hardware cache for virtual to physical translations. the cache should be populated on misses as usual. (like with any cache)
- TLB can also be multi level. TLB also needs to be aware of ownership. a process that doesn't own it shouldn't be able to read it.
- For TLB it might make sense to use a higher associativity so as to get more hits
- we can organize the cache in such a way between index and tag bits such that we can parallelize the TLB lookup and the cache lookup and simply do a comparison later to see if its a valid access instead of sequentially doing translation and then cache lookup
- types of cache misses



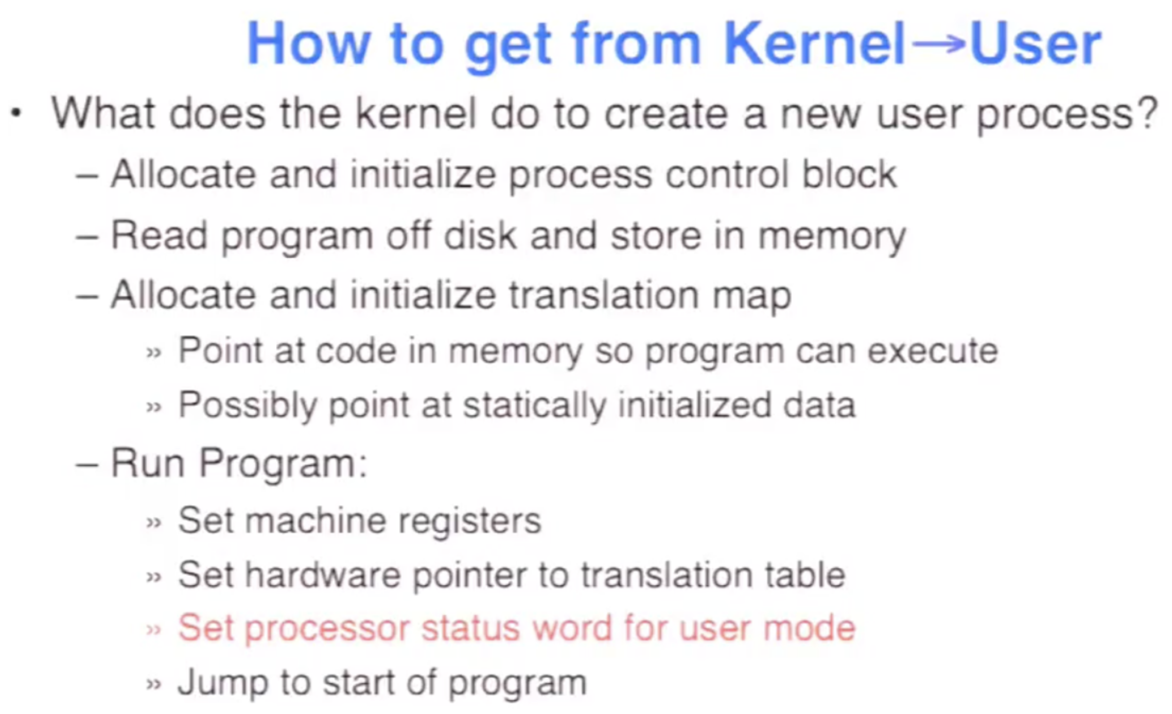
Sources of Cache Misses

- Compulsory (cold start): first reference to a block
  - "Cold" fact of life: not a whole lot you can do about it
  - Note: When running "billions" of instruction, Compulsory Misses are insignificant
- Capacity:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size
- Conflict (collision):
  - Multiple memory locations mapped to same cache location
  - Solutions: increase cache size, or increase associativity
- Two others:
  - Coherence (Invalidation): other process (e.g., I/O) updates memory
  - Policy: Due to non-optimal replacement policy

- if there are many levels in the page table, the disadvantage is that you have to do many hops with each access → bad!
- a structure called inverse page tables is used to map physical page table id to <virtual page numbers, pid>. this is used to decide which page to swap out of memory and onto disk while making space for new pages in memory
- the write policy for pages from memory to disk is write back → write thru would be way too slow
- so if you think of memory as a cache for pages with actual storage in disk, a page fault is when you are forced to go to disk. while this disk I/O happens, process gets swapped out for another process to run → so page faults really really suck.
- just like in caches, we have compulsory misses, capacity misses. we don't really have conflict misses since pages are fully associative. but page replacement policy does affect misses. also, no coherence misses here.
- again for page replacement algorithms, there is FIFO, LRU, random and a new one called clock algorithm
- the MIN replacement algorithm for pages is a theoretically really solid algorithm where we basically choose something that will be used the most ahead in the future. can be used for theoretical comparisons and benchmarking
- The LRU hopes to be like MIN: if you've been accessed recently you will probably be accessed soon enough so i won't remove you.
- Belady's anomaly is if you increase the memory (hence increasing the number pages that can be kept in memory) but yet the number of page faults increases. why? you can always construct access patterns such that you can screw up the algorithm you are using. so you aren't guaranteed that adding more space always increases hit rate for all access patterns. in other words, we can construct access patterns

where more memory is worse. this only happens for FIFO based replacement algorithms (think about why). LRU and MIN do not suffer from this.

- in practice, LRU is hard to implement efficiently because of the state tracking and manipulation needed. FIFO is efficient but not great. so we invent second chance algorithm → its basically FIFO with an additional referenced bit. each time a page is referenced, set the bit. while deciding which one to evacuate, start at the head of the queue and look for the first one whose referenced bit is not set. if all bits are set, you will do one cycle and come back to the beginning.
- a more efficient way to implement the second chance algorithm is called the clock algorithm. basically it involves thinking of the FIFO queue from above as a circular buffer to make more conceptual sense → its exactly second chance thought of as a circular buffer. when the clock ticks you just clear the referenced bit.
- its good if the clock's hand is moving slowly. this means that either we are not having so many page faults or if page faults happen, the replacement is being found quickly.
- An nth chance algorithm (instead of second chance) is a generalization. this means that each time a page is accessed, we increment its counter and each time the clock passes over it looking for a replacement it decrements. when it gets to 0 its ready to get kicked out. for large enough N, this is approaching LRU → think about why. the downside is that it could take a while to find a page as a replacement candidate.
- Working set is a concept in computer science which defines the amount of memory that a process requires in a given time interval → it is the active stuff, what's currently being used and ideally we have all of this cached.
- some part of the operating system runs in kernel mode → where it can access special state like page tables and stuff. applications run in user mode. the user mode itself can set permissions for different users on external resources like files and directories
- what happens while starting a new process

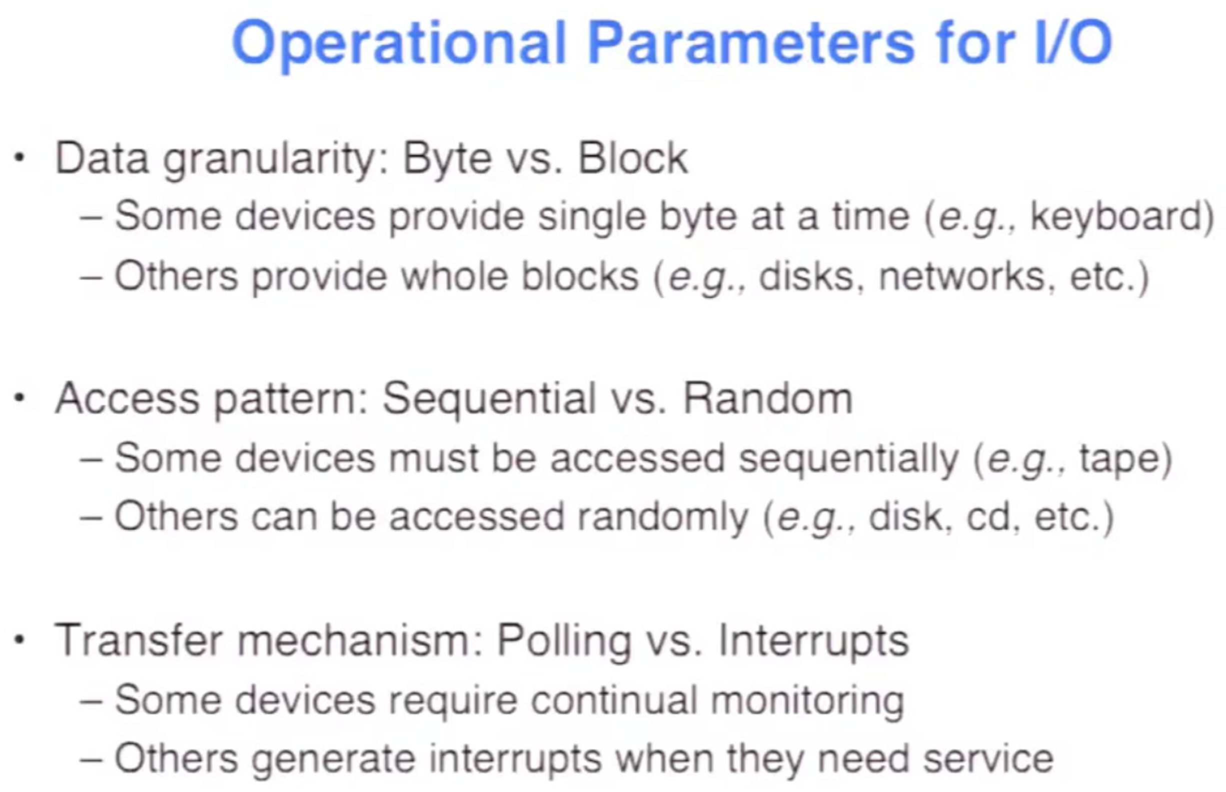


How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize process control block
  - Read program off disk and store in memory
  - Allocate and initialize translation map
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program

- if the application (running in user mode) needs to do something that involves calling a kernel mode code (like I/O), it does this thru a system call. the system call starts code in kernel mode but makes sure that the application can't do bad things (for the most part)

- Interrupts vs traps
- External devices have device drivers that help interract with the applications using the kernel. They are usually connected using the PCI bus. Conventional PCI, often shortened to PCI, is a local computer bus for attaching hardware devices in a computer. PCI is the initialism for Peripheral Component Interconnect and is part of the PCI Local Bus standard.
- PCIe (PCI express) is the newer version of PCI that has multiple serial lanes of different speeds.
- one of the main goals of an OS is to simplify interacting with the insane number of types of device drivers, thru memory mapped interaction, etc.
- here are the kinds of different things to think about to understand properties of an external device



Operational Parameters for I/O

- Data granularity: Byte vs. Block
  - Some devices provide single byte at a time (*e.g.*, keyboard)
  - Others provide whole blocks (*e.g.*, disks, networks, etc.)
- Access pattern: Sequential vs. Random
  - Some devices must be accessed sequentially (*e.g.*, tape)
  - Others can be accessed randomly (*e.g.*, disk, cd, etc.)
- Transfer mechanism: Polling vs. Interrupts
  - Some devices require continual monitoring
  - Others generate interrupts when they need service

- kernel code should avoid blocking calls because there is no one else to schedule for it!
- Read about DMA (direct memory access) vs memory mapped I/O
- "Everything is a file" describes one of the defining features of Unix, and its derivatives — that a wide range of input/output resources such as documents, directories, hard-drives, modems, keyboards, printers and even some inter-process and network communications are simple streams of bytes exposed through the filesystem name space.
- if you have lots of workers and each worker doesn't get too many notifications, interrupt based model makes sense. if you have few workers and each get a ton of notifications, polling probably makes more sense.