ECE 550D
Fundamentals of Computer Systems and Engineering

Fall 2023

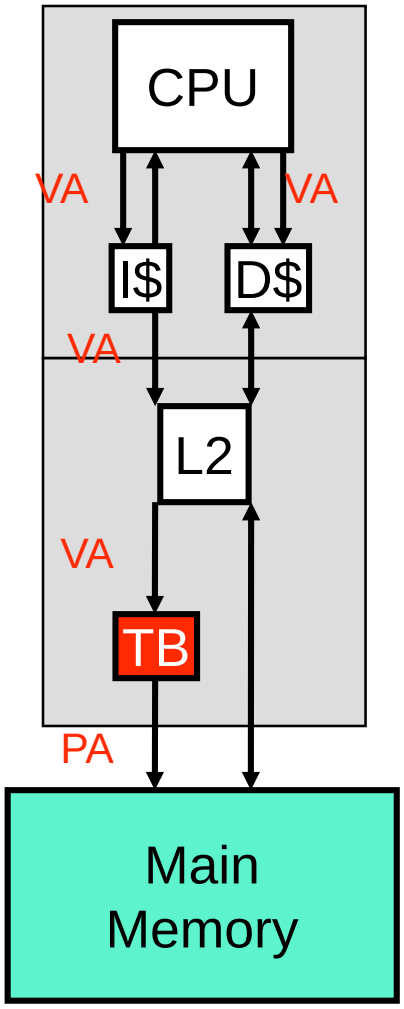
Virtual Memory, Exceptions and Interrupts

Xin Li & Dawei Liu
Duke Kunshan University

Slides are derived from work by
Andrew Hilton, Tyler Bletsch and Rabih Younes (Duke)

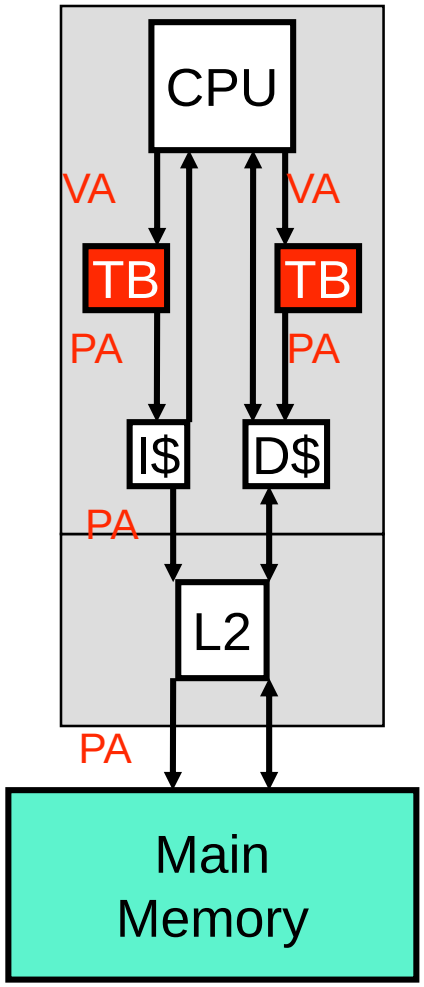
Translation Lookaside Buffer (TLB)

Virtual Caches



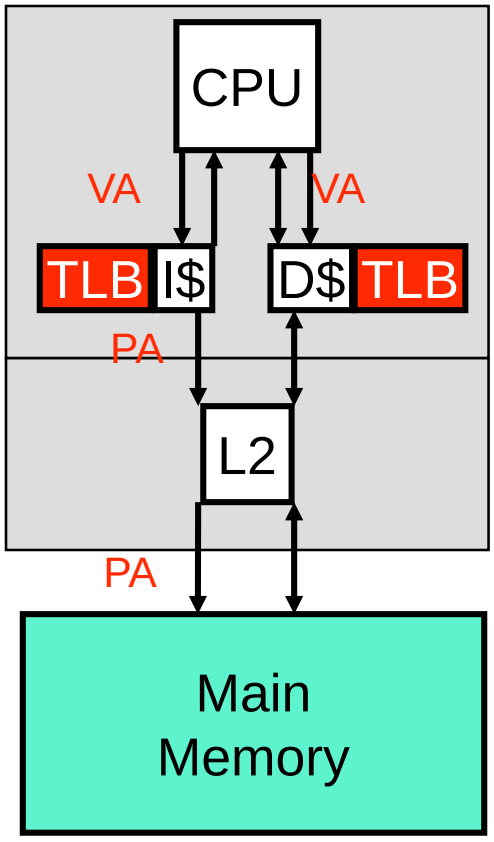
- Memory hierarchy so far: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access DRAM
  - \+ Fast: avoids translation latency in common case
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags?
- Does inter-process communication work?
  - **Aliasing**: multiple VAs map to same PA
    - How are multiple cache copies kept in sync?
    - Also a problem for I/O (later in course)
  - Disallow caching of shared memory? Slow

Physical Caches



- Alternatively: **physical caches**
  - Indexed and tagged by PAs
  - Translate to PA at the outset
  + No need to flush caches on process switches
    - Processes do not share PAs
  + Cached inter-process communication works
    - Single copy indexed by PA
  – Slow: adds 1 cycle to t_{hit}

Virtual Physical Caches



Compromise: **virtual-physical caches**

- Indexed by VAs
- Tagged by PAs
- Cache access and address translation in parallel

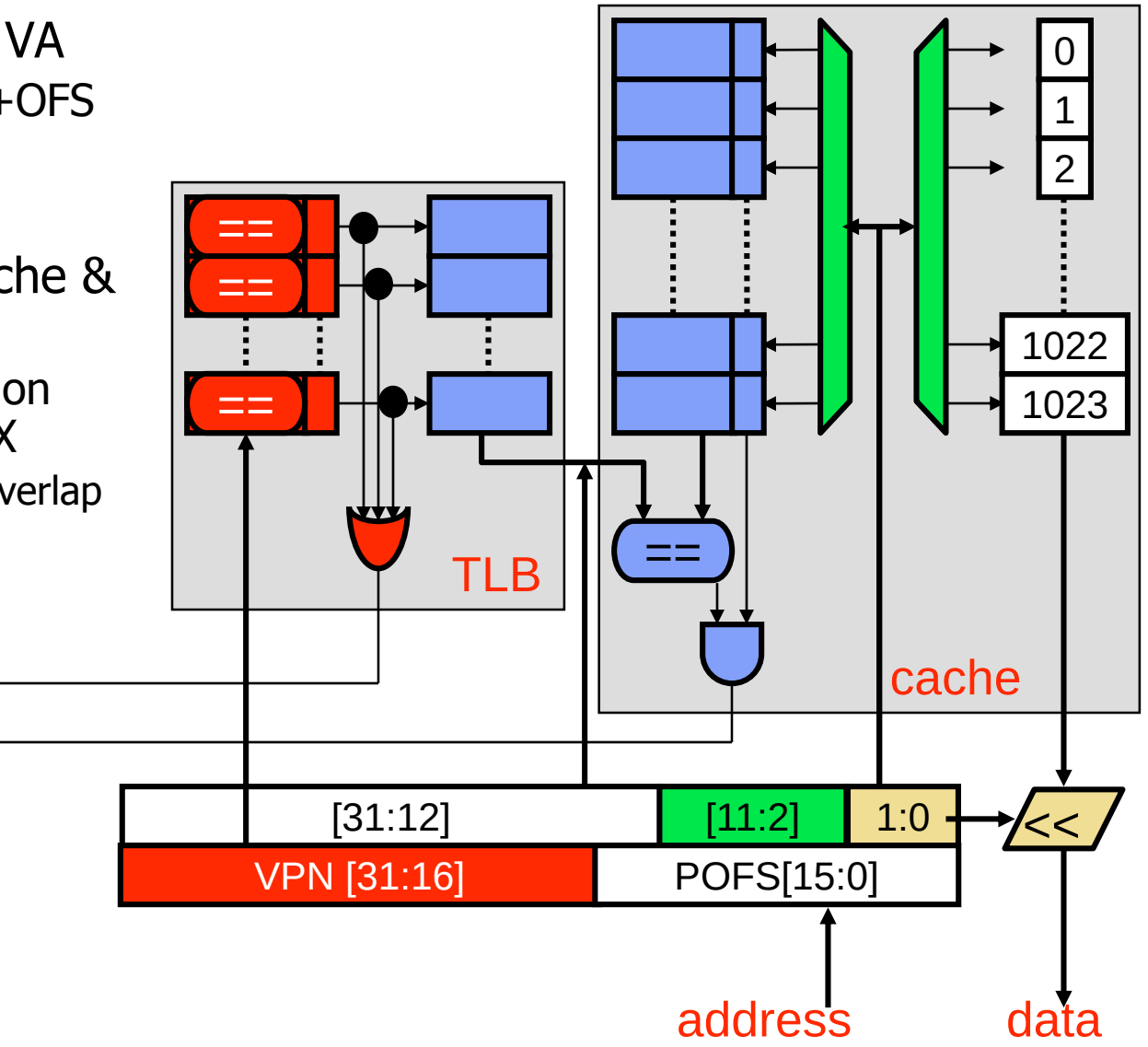
\+ No context-switching/aliasing problems

\+ Fast: no additional t_{hit} cycles

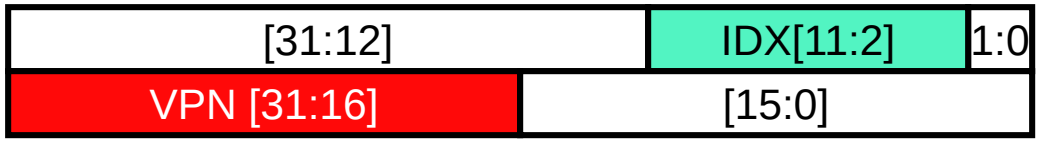
- A TB that acts in parallel with a cache is a **TLB**
  - **Translation Lookaside Buffer**
- Common organization in processors today

Cache/TLB Access

- Two ways to look at VA
  - Cache: TAG+**IDX**+OFS
  - TLB: **VPN**+POFS
- Can have parallel cache & TLB …
  - If address translation doesn't change IDX
  - → VPN/IDX don't overlap



Cache Size And Page Size



- Relationship between page size and L1 I$(D$) size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - I$(D$) size / **associativity** ≤ page size
  - Big caches must be set associative
    - Big cache → more index bits (fewer tag bits)
    - More set associative → fewer index bits (more tag bits)
  - Systems are moving towards bigger (64KB) pages
    - To amortize disk latency
    - To accommodate bigger caches

Flavors of Virtual Memory

- Virtual memory almost ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even some embedded (in non-computer) processors support it

- Several forms of virtual memory
  - **Paging** (aka flat memory): equal sized translation blocks
    - Most systems do this
  - **Segmentation**: variable sized (overlapping?) translation blocks
    - IA32 uses this
    - Makes life very difficult
  - **Paged segments**: don't ask

Performance and Thrashing

The Table of Time

Event	Picoseconds	≈	Hardware/target	Source
Average instruction time*	30	30 ps	Intel Core i7 4770k (Haswell), 3.9GHz	https://en.wikipedia.org/wiki/Instructions_per_second
Time for light to traverse CPU core (~13mm)	44	40 ps	Intel Core i7 4770k (Haswell), 3.9GHz	http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested/5
Clock cycle (3.9GHz)	256	300 ps	Intel Core i7 4770k (Haswell), 3.9GHz	Math
Memory read: L1 hit	1,212	1 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L2 hit	3,636	4 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: L3 hit	8,439	8 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Memory read: DRAM	64,485	60 ns	Intel i3-2120 (Sandy Bridge), 3.3 GHz	http://www.7-cpu.com/cpu/SandyBridge.html
Process context switch or system call	3,000,000	3 us	Intel E5-2620 (Sandy Bridge), 2GHz	http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html
Storage sequential read, 4kB (SSD)**	7,233,796	7 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage sequential read, 4kB (HDD)**	65,104,167	70 us	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (SSD)	100,000,000	100 us	SSD: Samsung 840 500GB	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Storage random read, 4kB (HDD)	10,000,000,000	10 ms	HDD: 2.5" 500GB 7200RPM	http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/whitepaper/whitepaper01.html
Internet latency, Raleigh home to NCSU (3 mi)	21,000,000,000	20 ms	courses.ncsu.edu	Ping
Internet latency, Raleigh home to Chicago ISP (639 mi)	48,000,000,000	50 ms	dls.net	Ping
Internet latency, Raleigh home to Luxembourg ISP (4182 mi)	108,000,000,000	100 ms	eurodns.com	Ping
Time for light to travel to the moon (average)	1,348,333,333,333	1 s	The moon	http://www.wolframalpha.com/input/?i=distance+to+the+moon

* Based on Dhrystone, single core only, average time per instruction
** Based on sequential throughput, average time per block

Performance of Demand Paging

Stages in Demand Paging:

- us **Trap** to the operating system
- ns Save the user registers and process state
- ns Check that the page reference was legal and determine the location of the page on the disk
- Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced
  - Wait for the device seek and/or latency time
  - Begin the transfer of the page to a free frame

  us ms
- While waiting, allocate the CPU to some other process
- Receive an interrupt from the disk I/O subsystem (I/O completed)
- ns Save the registers and process state for the other process
- ns Correct the page table and other tables to show page is now in memory
- ? Wait for the CPU to be allocated to this process again
- us Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Waiting on disk is bad

- If we frequently have to go to disk, the memory latency becomes like disk latency (1000x worse!)

- Need the vast majority of memory accesses to *be in memory*.

- Remember when our **working set** didn't fit in **cache**?
- It's like that, but now our **working set** isn't fitting into **RAM**!

- When this happens, it's called **thrashing**.
- Makes CPU appear under-utilized, as most time is spent waiting on a disk.

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back

- Thrashing ≡ a process is busy swapping pages in and out

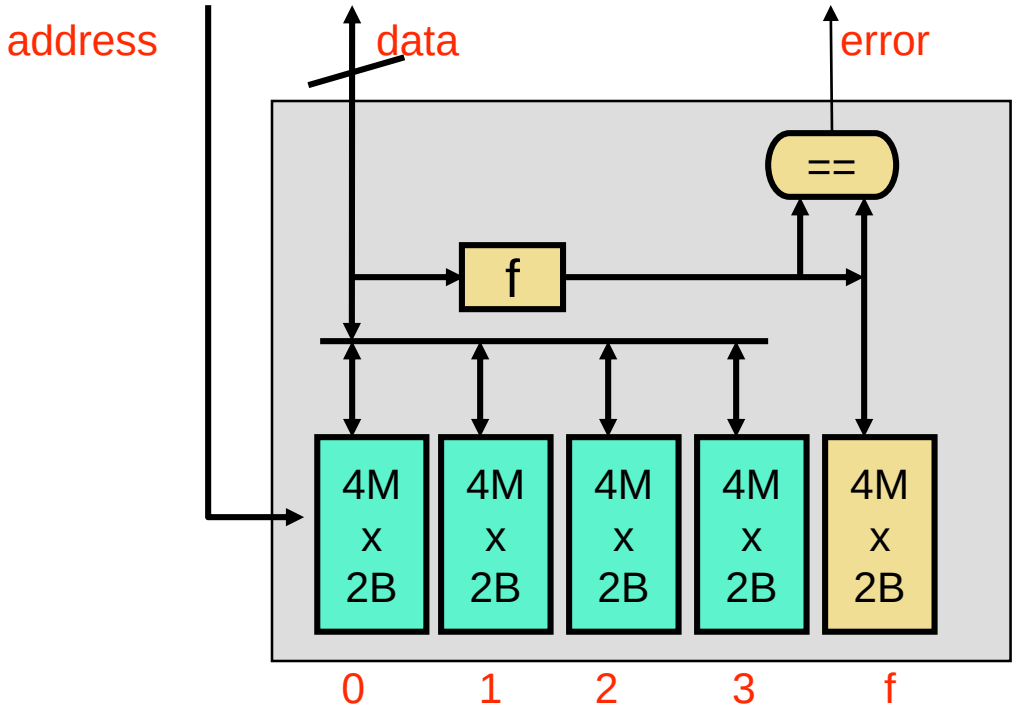
Error correction in DRAM

Error Detection and Correction

- One last thing about DRAM technology: **errors**
  - DRAM fails at a higher rate than SRAM or CPU logic
    - Capacitor wear
    - Bit flips from energetic α-particle strikes
    - Many more bits
  - Modern DRAM systems: built-in error detection/correction

- **Key idea: checksum-style redundancy**
  - Main DRAM chips store data, additional chips store f(data)
    - |f(data)| < |data|
  - On read: re-compute f(data), compare with stored f(data)
    - Different ? Error…
  - Option I (**detect**): kill program
  - Option II (**correct**): enough information to fix error? fix and go on

Error Detection and Correction

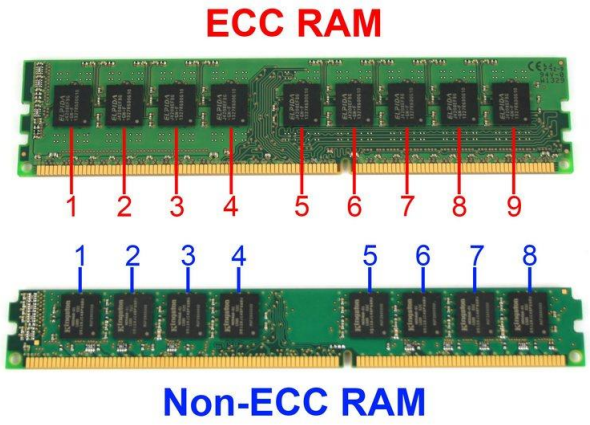


- Error detection/correction schemes distinguished by…
  - How many (simultaneous) errors they can detect
  - How many (simultaneous) errors they can correct

Error Detection Example: Parity

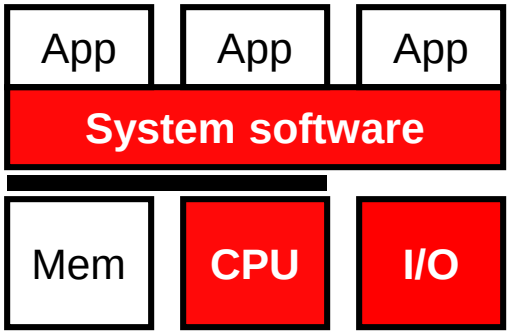
- **Parity**: simplest scheme
  - f($data_{N-1...0}$) = XOR($data_{N-1}$, …, $data_1$, $data_0$)
  + Single-error detect: detects a single bit flip (common case)
    - Will miss two simultaneous bit flips…
    - But what are the odds of that happening?
  – Zero-error correct: no way to tell which bit flipped

– Many other schemes exist for detecting/correcting errors
  – Take ECE 554 (Fault Tolerant Computing) for more info



Exceptions and Interrupts

Exceptions and Interrupts



- Interrupts:
  - Notification of external events
- Exceptions:
  - Situations caused by program, requiring OS
- Also:
  - A bit about the OS

External Events

- Focus so far: running an application
  - Low level coding (assembly)
    - We don't worry so much about C etc in this class
  - How to execute the instructions…
  - And store the data…
  - And give the illusion of a uniform address space…
- System software (OS) has to deal with external events
  - Which may come at un-expected times
  - Data arrives on network…
  - Disk complete read request…
  - Fixed interval timer…

First question: Finding out?

- Suppose we expect an outside event
  - E.g., requested disk drive read something…
  - It will get back to us later with data (think 10M cycles)

- How do we know when its done?
  - Option 1: **Polling**
    - Ask it periodically
    - "Are we there yet?" No… "Are we there yet?" No
    - Downside: can be inefficient (processor busy asking)
  - Option 2: **Interrupts**
    - "Read a book, I'll tell you when we are there"
    - External device signals to processor when it needs attention

Interrupts

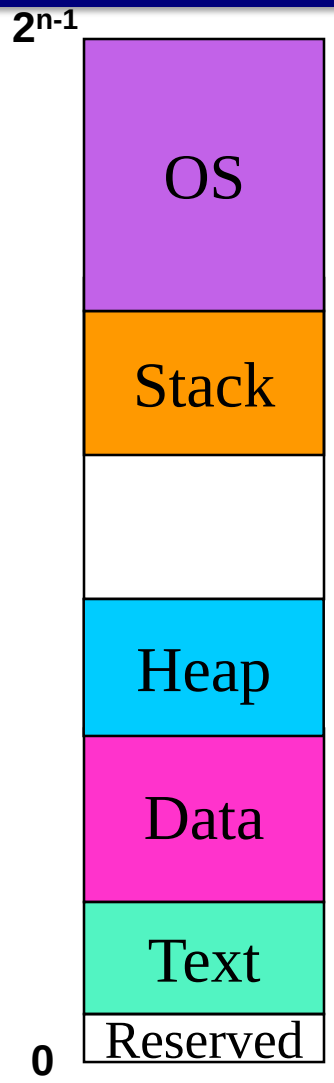
- Step 1: External device raises an interrupt
  - "Hey, processor! I need your attention!"
  - Different interrupt numbers, specifies which one it is
  - Multiple interrupts at once?
    - Interrupt controller prioritizes which one goes to processor
- Step 2: CPU transfers control to OS **interrupt handler**
  - Stops what its doing (drain pipeline: stall front end until empty)
  - Jumps into interrupt handler (and saves current PC)
  - Switches into **privileged mode**
- Step 3: OS runs interrupt handler
  - Software routine to do whatever needs to be done
- Step 4: OS returns from interrupt
  - Jumps back to application code, leaving privileged mode

Interrupt handlers

- How does processor know where to jump?
  - OS sets up **interrupt vector** in system startup
    - Array of PCs to jump to for interrupt routines
    - Indexed by interrupt number
  - What if….
    - Another interrupt happens while handling the first one?
    - Or an interrupt happens during interrupt vector is setup?
    - OS can enable/disable interrupts (privileged instruction)
      - Allows it to prevent problematic situations
      - "Look, this is important, don't bother me right now!"

Speaking of OS code… where is it?

- Where does OS code reside?
  - In memory….
  - But doesn't application think it has all of memory to itself?
  - Well sort of…
    - It doesn't think anything exists past the top of the stack…
    - So the OS "lives" there
    - Same physical pages mapped into all processes' address spaces
    - Privileged bit in page table prevents access by "normal" code
      - Mapping only "valid" when in privileged mode



Timer Interrupt: Heart of multitasking

- Common interrupt: timer interrupt
  - "Ticks" at fixed interval
  - Gives OS a chance to change currently running program
  - …and keep track of the current time
  - …and anything else it needs to do

- This is what lets your computer run multiple programs
  - The OS switches between them quickly
  - Enabled by timer interrupt giving control to OS

Exceptions: Like interrupts, but not…

- Interrupts: external events
  - Asynchronous—don't really "belong to" any current instruction
- Exceptions: unusual circumstances for an instruction
  - Belong to one particular instruction
  - Examples:
    - Page fault: load or store missing translation
    - Divide by 0
    - Illegal instruction
      - Bits do not encode any valid instruction
      - Or, privileged instruction from user code

Interrupts vs Exceptions

- Exceptions:
  - Processor must (typically) tell OS which instruction caused it
  - OS may want to restart from same instruction
    - Example: page fault for valid address (on disk)
  - Or OS may kill program:
    - Segmentation fault: (or other fatal signal)
    - Aside: OS sends "signals" to program to kill them
      - Segfault = SIGSEGV
      - Programs can "catch" signals and not die…
      - But not in this class…
- Interrupts: no particular instruction
  - But OS will always restart program after last complete insn
- Both require **precise state**: insns either done, or not
  - Division between "done and not done" in program order

Precise state

- Instructions either done or not: sounds obvious right?
  - Problem: "half done" instructions: pipeline splits into stages
    - Need to ensure **no** state change (reg or mem) if not done
  - Also: need "clean" division.
    - Instructions before exception, all done
    - Instructions after (and including) exception, no effect

- For interrupts:
  - Must be precise

Handling Exceptions

- Exceptions handled just like interrupts
  - Some ISAs just give them interrupt numbers
  - Others have separate numbering for exceptions

System calls: Exceptions on purpose

- Programs need OS to do things for them
  - Read/write IO devices (including printing, disks, network)
  - Tell "real" time of day
  - Spawn new processes/execute other programs
  - …
  - Any interaction with the "outside world"
- Make a system call (syscall) to do this
  - Special instruction which **traps** into OS
  - Basically just causes exception—specifically for this purpose
  - OS gets control (in privileged mode), and does what program asked
    - Knows what program wants by arguments in registers
    - May deny request and not do it…then returns an error

System calls: Kind of slow

- Bothering OS for stuff: kind of slow
  - Empty pipeline…
  - Transfer control/change privilege
  - Have OS figure out what you want…
  - Then do it…
  - Then drain pipeline again
  - Then jump back into program
- For long tasks, overhead to enter/leave is amortized
  - Reading disk (very slow)
- For short tasks, overhead is very high
  - Get current time of day

Avoiding slowness

- Userspace (not OS) libraries help avoid by buffering
  - Example: malloc
  - Malloc does not ask OS for more memory on every call
  - Instead, malloc asks OS for large chunks of memory
  - Then manages those chunks itself (in user space)

Vsyscalls: a slick trick

- Linux has a slick trick: vsyscalls
  - Don’t actually make a system call!
  - Example: get current time of day
    - Just needs to read an int (time in seconds)
  - OS maps vsyscall page into all processses
    - Read/execute only
    - All processes map to same physical page
  - OS writes current time to fixed location on this page
    - On each timer interrupt
  - gettimeofday “system call” actually not a system call
    - Just library function which jumps onto vsyscall page
    - Code there reads time and returns it

Summary

- Virtual memory
  - Virtual, physical, and virtual-physical caches and TLBs
- Error correction
- Interrupts: Notification of external events
- Exceptions: Unusual things for an instruction
- Both: handled by OS, very similar behavior
- System calls: Ask OS to do something (also, like exception)