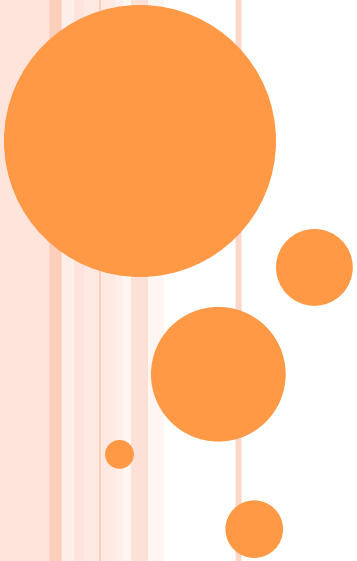


CS24: Introduction to Computing Systems

Spring 2015

Lecture 26

CS24 Final Exam

- **Final exam format:**
  - 6 hour overall time limit, multiple sittings
  - Open book/notes/slides/homework/solutions
  - Can use computer to implement and test your work
    - (although the final usually has very limited coding)
  - No collaboration!
- **Important Internet usage notes:**
  - Can use any material on CS24 Moodle website, or pages directly referenced by CS24 Moodle website
  - Can use Internet resources referenced by final exam
  - You cannot generally search for material related to the final exam on the Internet
  - You can't get help from other people over the Internet

CS24 Final Exam (2)

- Potential topics include:
  - **Generally, anything covered in last half of class**
    - (except for static and dynamic linking material)
  - Memory hierarchy, hardware caches, locality, cache utilization
  - Processes, process status, process management, scheduling
  - Locking and concurrent programming
  - Exceptional control flow, hardware exceptions, signals
  - Virtual memory systems
- May see some topics from first half of class
  - Not in as much detail as in midterm, though

CS24 Final Exam (3)

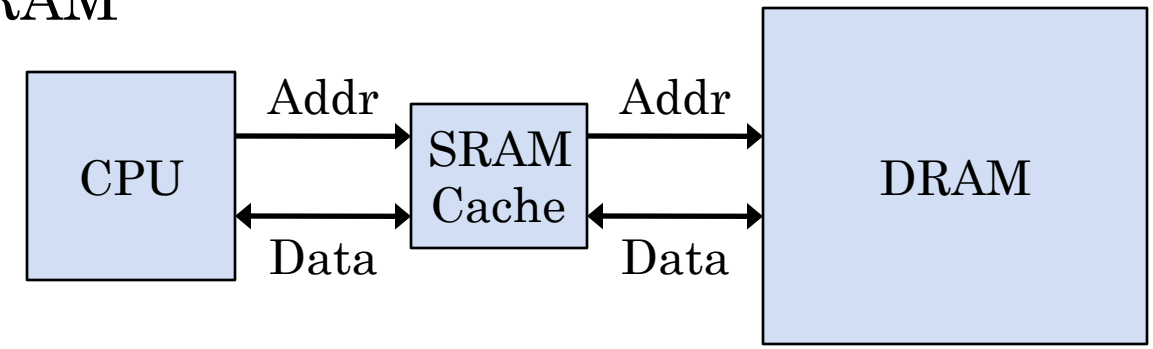
- Final exam tends to be more conceptual in nature
  - Systems explored in last half of term *very* complex
  - (Think about how much supporting code the assignments required…)
- Mainly design/analysis/conceptual questions
- Make sure you are familiar with basic concepts and overall themes discussed in last half of term!
- Knowing technical details always helps too…
  - The more detailed (and accurate) your answers are, the more likely you will get full credit ☺

Computer Memories

- Explored computer memory principles and technologies in Lecture 13
- Basic principles:
  - Small memories are faster than large memories
    - Physics: signal propagation and addressing logic delays
  - Large memories can be made denser and cheaper than small memories
    - Large memories have simpler data representations
    - Can be packed more densely, but typically require more work to retrieve a given data value
- The Processor-Memory performance gap:
  - CPU performance has been increasing 50% faster per year than DRAM performance
  - This gap is widening

Caching

- Solution: Introduce an SRAM cache between CPU and DRAM

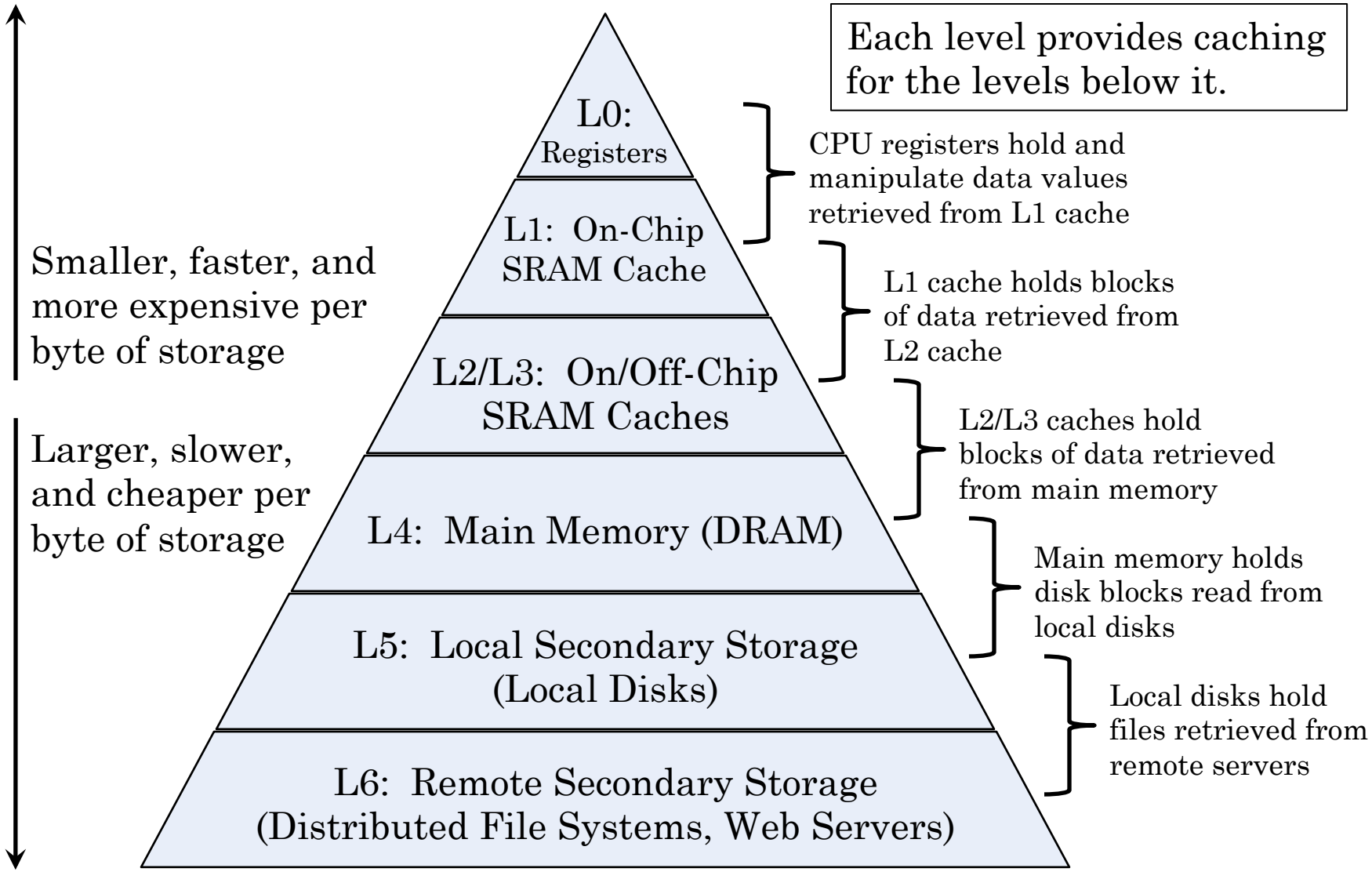


  - Larger and slower than memory on CPU (the register file)
  - Smaller and faster than DRAM main memory
- When a program accesses a data value:
  - Also fetch nearby values from DRAM into the cache
- Assumption: if a program accesses a given value…
  - Likely to access the same value again soon
  - Likely to access nearby values soon

Caching and Locality

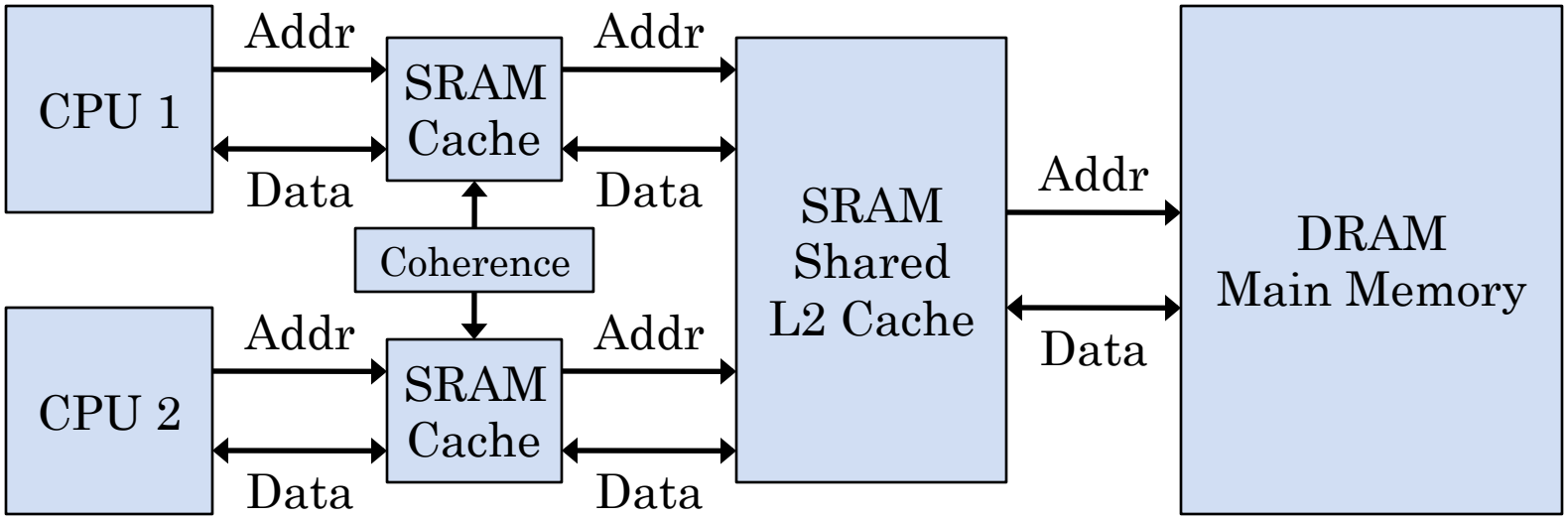
- Programs must exhibit good locality if they are going to utilize the cache effectively
- Spatial locality:
  - Program accesses data items that are close to other data items that have been recently accessed
- Temporal locality:
  - Program accesses the same data item multiple times
- Programs with good locality tend to run faster than programs with poor locality
  - Can leverage processor caches more effectively
- Effectively gives us a memory the size of DRAM, with (almost) the performance of SRAM cache

The Memory Hierarchy



Computer Caches

- Multiple levels of caching in modern computers

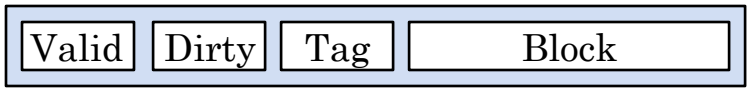


- L1 and L2 caches are managed in hardware
  - Very fast access times
    - L1 often 1-3 clocks, L2 often 10-25 clocks
  - Don't have much time to figure out where cached data should go, or where to look for data

Hardware Cache Terminology

- Hardware caches store blocks of data
  - e.g. Pentium L1 and L2 caches store 32-byte blocks
- Cache also needs to track the state of each cached block…
- Cache lines include the block of data, plus additional state bits:

Cache line:



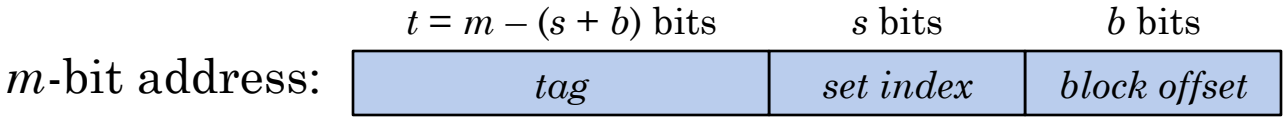
- When a cache is accessed, it receives an address
  - Most expensive step is determining which cache line contains the value specified by the address
- Each cached block of data is identified by a tag, taken from the block's original address

Cache Placement Policies

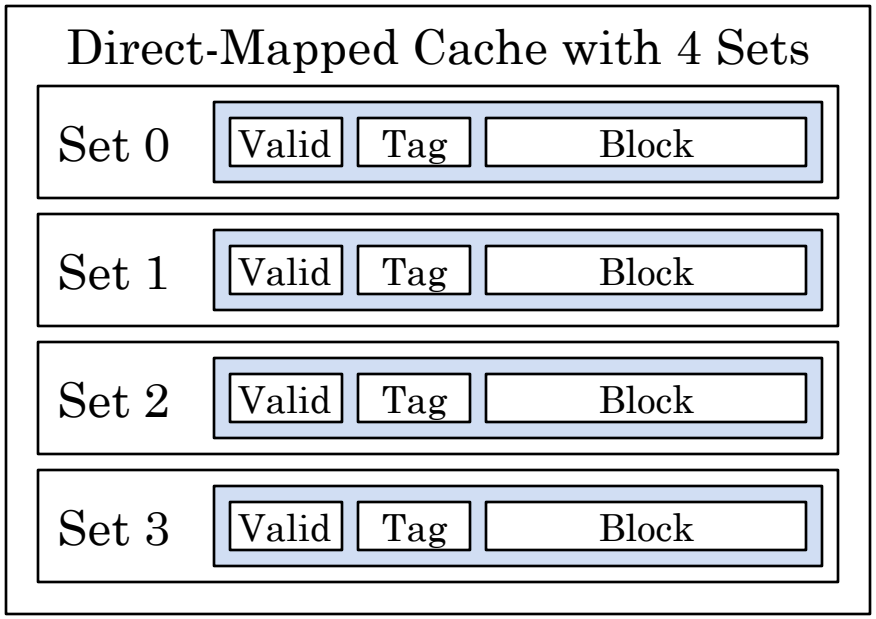
- Hardware caches use stricter placement policies
  - Blocks from level $k + 1$ can only be stored into a subset of locations in level k
  - Makes it much faster and easier to determine if a block is already in the cache
- Cache lines are grouped into cache sets
  - Every block from level $k + 1$ maps to *exactly one* set in the cache at level k
  - Within the set, block can be stored in *any* cache line
- Two extremes for our cache organization:
  - S cache sets, each of which contains exactly one line
    - Called "direct-mapped caches"
  - One cache set which contains all E cache lines
    - Called "fully associative caches"

Direct-Mapped Caches

- Direct-mapped caches have S cache sets, but each set contains only one cache line

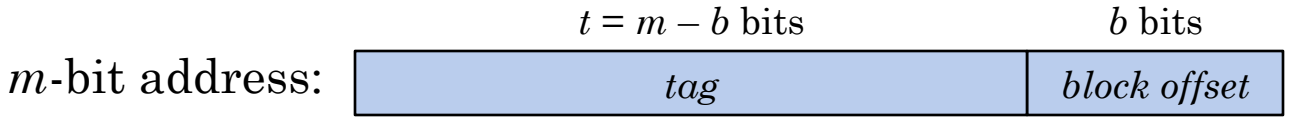


- Example: direct-mapped cache with 4 sets
  - 2 bits in set index
- Very fast to map an address to a cache set and a cache line
- Very easy to have access patterns that cause lines to be replaced frequently
  - Causes *conflict-misses*

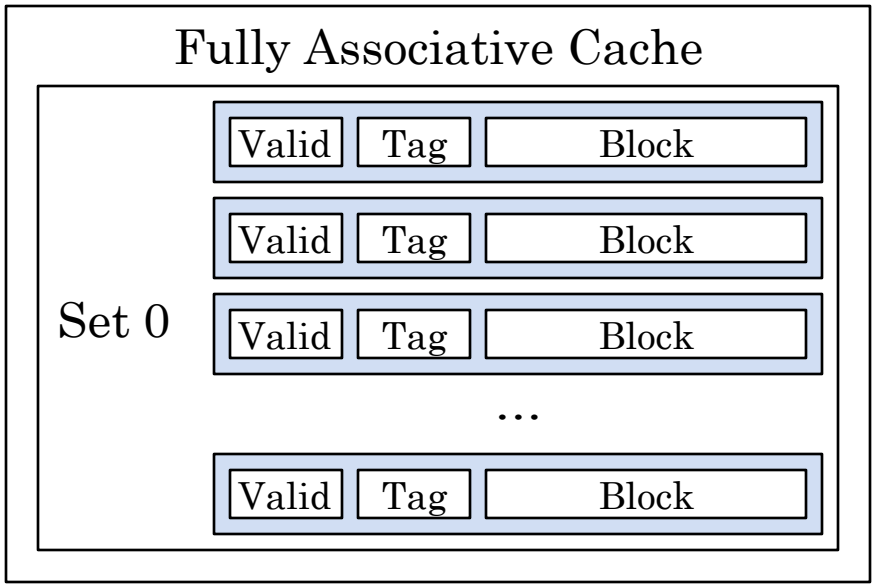


Fully Associative Caches

- Fully associative caches have only one cache set, which contains all cache lines
  - $S = 2^s = 1 \Rightarrow s = 0$. No bits used for set index!



- Still fast to map address to a cache set, but much more complex to find a given block in the cache
  - Must examine all cache-line tags, which is slow
  - Needs complex logic: an *associative memory*
    - Maps tags to blocks

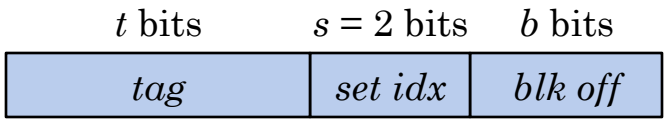


Set-Associative Caches

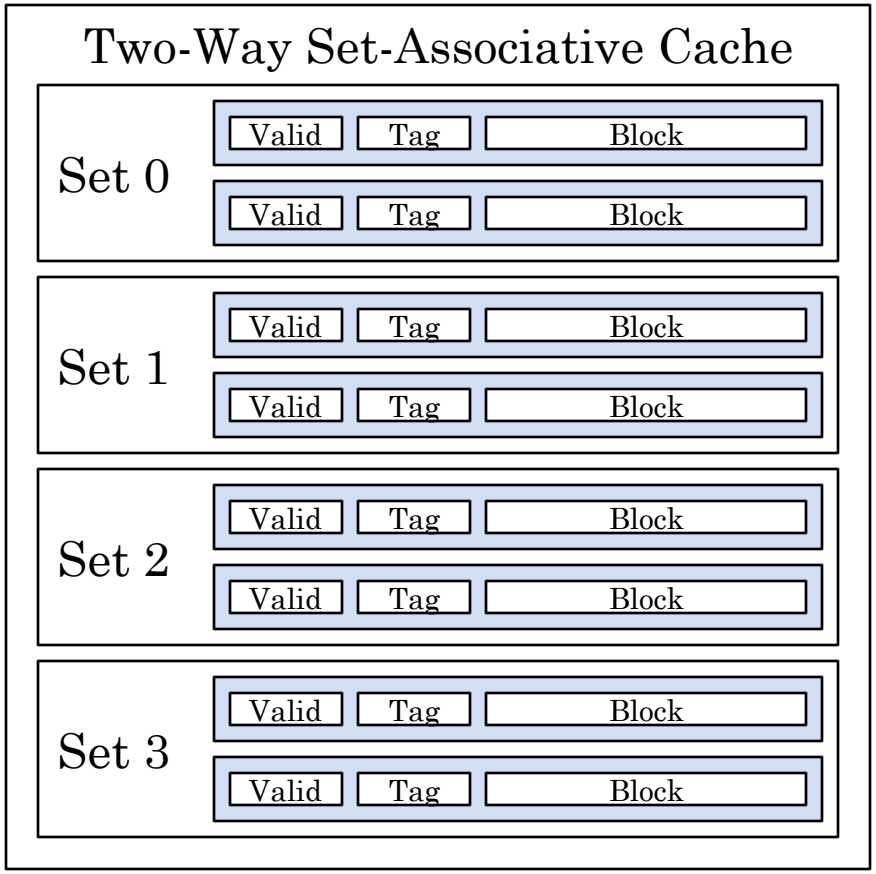
- Two designs with different strengths:
  - Direct-mapped caches have fast and simple lookups, but can easily generate many conflict misses
  - Fully associative caches have fewer cache misses, but have complex logic to identify blocks by their tags
- Set-associative caches combine capabilities of both approaches into a single cache
  - Employ S cache sets, where $S > 1$
  - Each cache set contains E cache lines, where $E > 1$
- *A common pattern in computer system design!*
  - Multiple designs, each with different strengths
  - Combine these designs into approach that maximizes strengths, minimizes weaknesses of original designs

Set-Associative Caches (2)

- Example: a two-way set-associative cache
  - E is number of cache lines in each set…
  - Cache is called "E-way set-associative cache" when $S > 1$
  - $E = 2$ for this example
- For an m-bit address:



- Use set index to find cache set to examine
- Only have to check tags on small number of lines



Line Replacement Policies

- When a cache miss occurs, must load a new block into the cache
  - Store into some cache line
- If the cache is set-associative or fully associative, choose a line to evict based on a replacement policy
  - Least Recently Used (LRU) policy
    - Evict cache line that was accessed furthest in the past
  - Least Frequently Used (LFU) policy
    - Evict cache line that was accessed the least frequently, over some time window
  - *Many other policies too…*
- Hardware caches use simple policies
  - Performance is critical; cache miss often relatively cheap
- Software caches use much more sophisticated policies
  - Cache miss is typically *much* more expensive than a hit, so want to spend extra time to avoid misses, if possible!

Writing to Caches

- Different policies for handling cache writes, too
- Write-through policy:
  - Every write to cache causes cache to write the entire block back to memory
  - Problem: *every single write* to the cache causes a write to main memory! Can't exploit data locality!
- Write-back policy:
  - Add a "dirty flag" to each cache line
  - When a cached block is written to, set the dirty flag
  - When a cache line is evicted, write the block back to main memory
  - Can exploit locality with writes, as well as reads

Cache Performance Analysis

- Cache performance modeling can be extremely complicated
  - Mathematical models tend to be very simple, and capture only the most basic high-level ideas
  - Best solution is to measure cache access performance as the system runs your program on your data
- Miss rate: the fraction of memory references that result in cache misses
  - Miss rate = # misses / # references ($0 \leq$ miss rate ≤ 1)
- Hit rate: the fraction of memory references that hit the cache
  - Hit rate = 1 – miss rate

Cache Performance Analysis (2)

- Hit time: time to deliver a value from the cache to the CPU
  - Includes all necessary steps for delivering the value!
  - Time to select the appropriate cache set
  - Time to find the cache line using the block's tag
  - Time to retrieve the specified word from block data
- Miss penalty: time required to handle cache miss
  - Need to fetch a block from main memory (or, the next level of cache)
  - May need to evict another cache line from the cache
  - (Evicted line may be dirty and need written back…)
  - Other associated bookkeeping for storing cache line

Cache Performance Analysis (3)

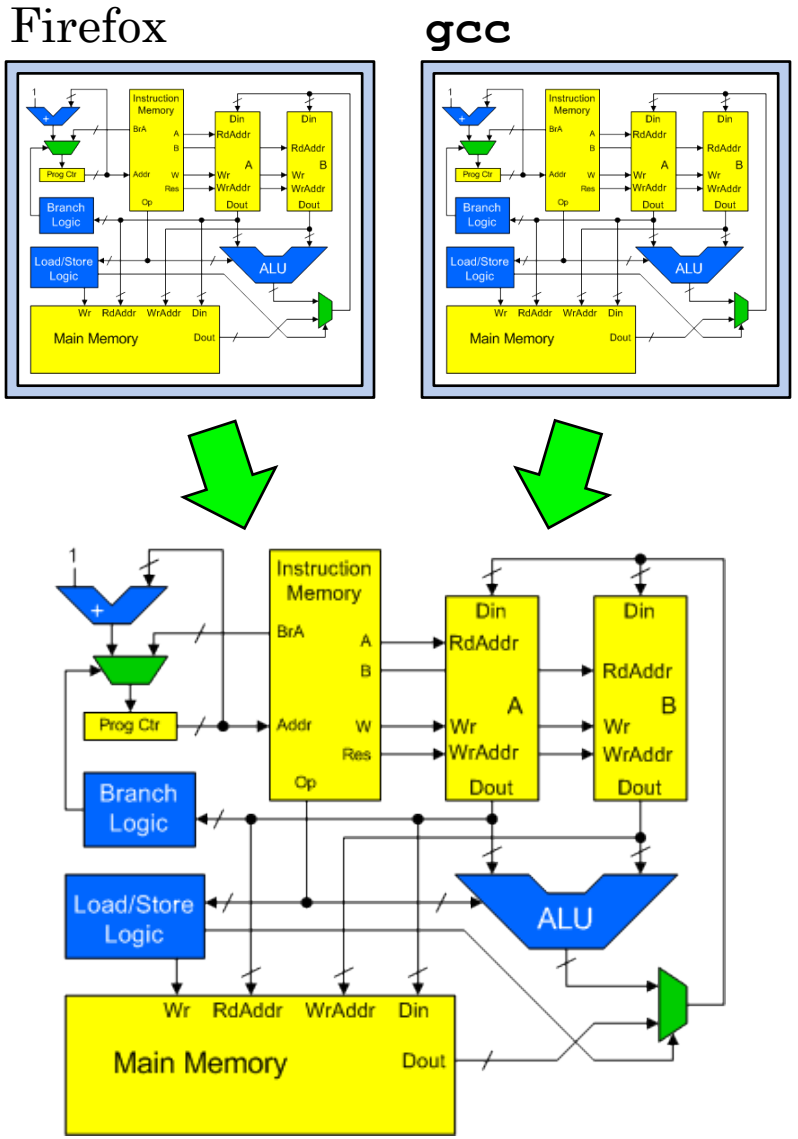
- Simple example to see benefit of caching:
  - Hit time = 1 clock (typical goal for L1 hits)
  - Miss penalty = 50 clocks (main memory usu. 25-100)
- If all reads were from cache, each read is 1 clock
- Hit rate of 80%
  - 0.8 × 1 clock + 0.2 × 50 clocks = 10.8 clocks/access
- Hit rate of 90%
  - 0.9 × 1 clock + 0.1 × 50 clocks = 5.9 clocks/access
- Hit rate of 95%
  - 0.95 × 1 clock + 0.05 × 50 clocks = 3.45 clocks/access
- Hit rate is very important!
  - For programs with low miss-rate (good data locality), get a large memory at (nearly) the cost of a small one!

Virtualization

- Rest of term, talked about *virtualization*
- Want to run multiple programs on computer at once…
  - Different programs, or multiple instances of same program
- Virtualize the processor
  - Make it look like we have multiple processors
  - Each program runs on "its own processor"
- Similarly, virtualize the computer's main memory
  - Each program appears to have sole access to main memory
  - Each program's memory is isolated from other programs
- *The computer that the program sees is different from the actual computer.*
  - Requires careful management of hardware on behalf of programs running on the computer

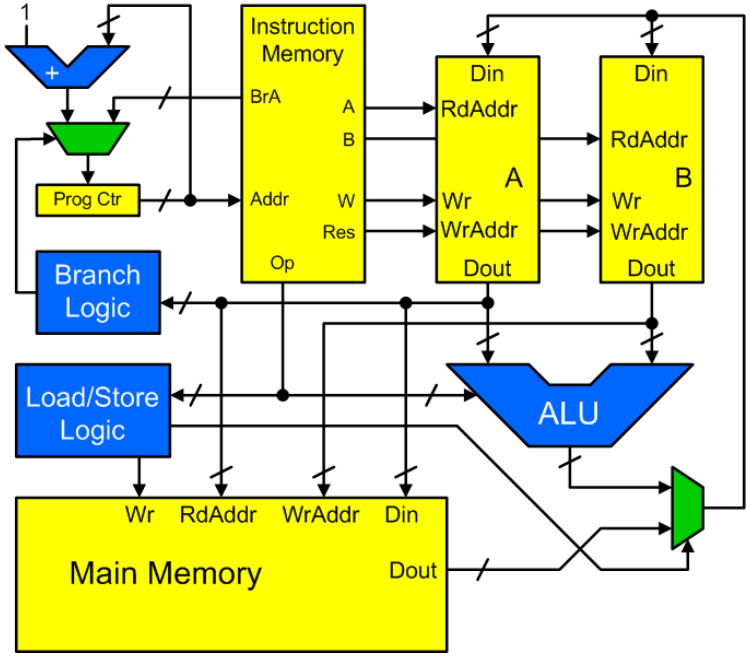
Processes

- The notion of a program running on a virtual processor is called a process
- A process is "an instance of a program in execution"
  - The program itself – code, read-only data, etc.
  - *All state* associated with the running program
- The running program's state is called its context
  - Each process has a context associated with it



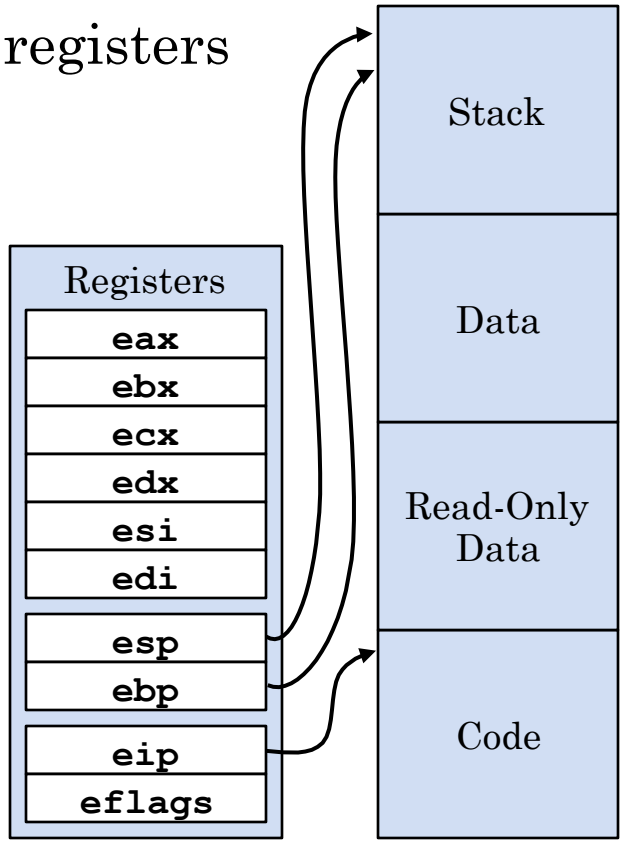
Program Context

- The physical processor can still run only one program at a time…
  - Only one program counter, instruction memory, ALU, register file, main memory, etc.
- Simulate multiple concurrent programs by giving each one its own turn to run on the physical processor
- When a process is running, it has exclusive access to the processor hardware, until it is suspended
- A *kernel* or *operating system* manages the hardware, on behalf of all running processes



Program Context (2)

- On IA32, the program's context contains:
  - Current state of all general-purpose registers
    - **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**
    - *(Also floating-point registers, etc.)*
  - Current program counter: **eip**
  - Current stack pointers: **esp**, **ebp**
  - Also, current state of **eflags** register (see **pushfl**/**popfl**)
- Context also includes the program's in-memory state
  - Virtual memory abstraction makes it fast and easy to record the process' memory state



Switching between Processes

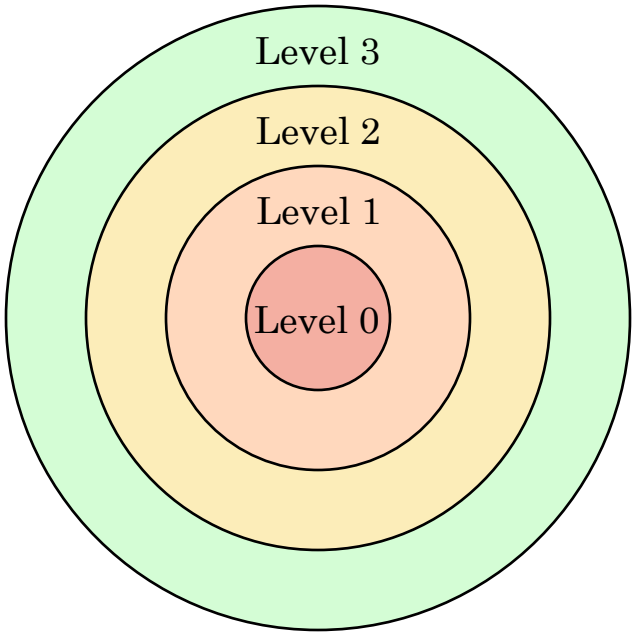
- Can switch the processor from running one process to running another, by performing a context switch
  - Stop the current process' execution via an exception
  - Save all context associated with the current process
  - Load the context associated with another process
  - Resume the new process' execution
- Two main ways to switch between processes
- Cooperative multitasking
  - Each process voluntarily gives up the processor
  - Problem: one selfish process affects the entire system!
- Preemptive multitasking
  - Processes are forcibly interrupted after a certain time interval, to give other processes time to run

Operating Modes

- Kernel must be able to do more than the application processes should be allowed to do
- Processor hardware provides *operating modes*
- Kernel mode:
  - Program can do *everything* the processor supports
  - Access all of memory, use special instructions, etc.
  - Also known as "protected mode" or "privileged mode"
- User mode:
  - Program has a restricted view of the world
  - Can only access its own memory
  - Special control instructions are disallowed
    - e.g. ones that set the processor mode
  - Also called "normal mode"

IA32 Operating Modes (2)

- IA32 provides four different operating modes
- Reason:
  - Some software components need more privileged access to the processor, but they don't need to access *everything*…
  - Device drivers, specific operating system services, etc.
  - Partition OS code into privilege levels
- Modes form a security hierarchy:
  - Lower number = higher privileges
  - Each privilege level has its own stack and memory areas
- OS kernel runs at level 0
- OS services run at levels 1 and 2
- Applications run at level 3



Exceptional Control Flow (2)

- Must be able to interrupt normal program flow, for several reasons
  - e.g. kernel preempts a process, disk controller finishes reading data, a timer fires an alarm signal for a process
- Interrupts
  - Caused by hardware signaling to the processor
  - e.g. external I/O devices that have completed operations
  - Usually not caused by execution of a specific instruction
  - (Software can also invoke an interrupt handler manually, if desired…)
- Exceptions
  - Caused by a program executing an instruction
  - Exception can be intentional, to perform a task…
  - Or, exception may be unintentional, if error occurred

Exception Classes

- **Interrupts** are caused by hardware
  - Example: a periodic timer interrupt
  - Handler can respond to the hardware interrupt
  - Then, control returns back to interrupted program
- **Traps** are intentional exceptions caused by programs
  - Frequently used to implement operating system calls
  - Caller specifies requested service when invoking the exception
  - Processor switches from user-mode to kernel-mode when jumping to the exception handler
  - Operating system can provide requested service…
  - Then, control returns back to interrupted program

Exception Classes (2)

- **Faults** are unintentional exceptions caused by software
  - Represent error conditions that might be recoverable
- Example: virtual memory that is paged to disk
  - Program accesses a page that isn't in memory
  - Processor causes a page fault, which invokes the page fault handler
  - Handler loads requested page from disk into memory
  - Execution resumes with the instruction that caused the fault (not the next instruction!)
- If a fault handler cannot recover from a fault, the program is usually terminated

Exception Classes (3)

- **Aborts** are unrecoverable fatal errors
  - Frequently used to handle hardware errors
  - Example: IA32 Machine-Check exception is an abort
  - Handler never returns to the interrupted program
- In fact, entire system may grind to a halt!
  - Windows "Blue Screen of Death" and Linux kernel-panic can both occur because of an abort

IA32 Exceptions

- IA32 processors support 256 different kinds of exceptions
  - Each is assigned an integer from 0 to 255
- Exception types 0 to 31 are IA32 architecture-defined interrupts and exceptions
  - e.g. divide-by-zero fault, page fault, general protection fault, machine-check abort
- Exception types 32-255 are user-defined exceptions
  - Can be assigned to hardware devices, used by the operating system, etc.
  - On UNIX platforms, exception 128 (0x80) is a trap used for making operating system calls

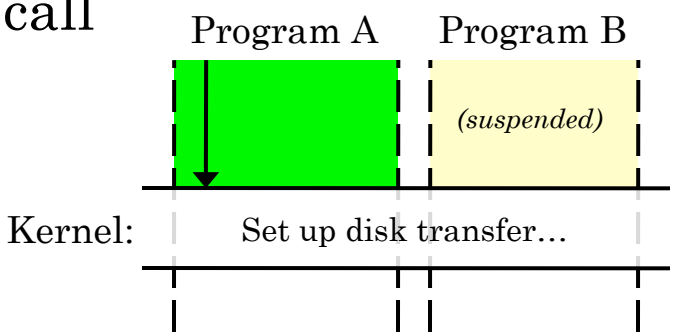
IA32 Exceptions (2)

- Can invoke handler for any exception type with IA32 instruction **`int n`**
  - ***n*** is the type of the exception
- **`int`** instruction also allows a change to a higher privilege level
  - User-mode process can invoke kernel to perform some privileged operation
  - Kernel verifies the request, performs the operation, then returns to calling process

- *Operating system is an intermediary between the computer hardware and application programs.*

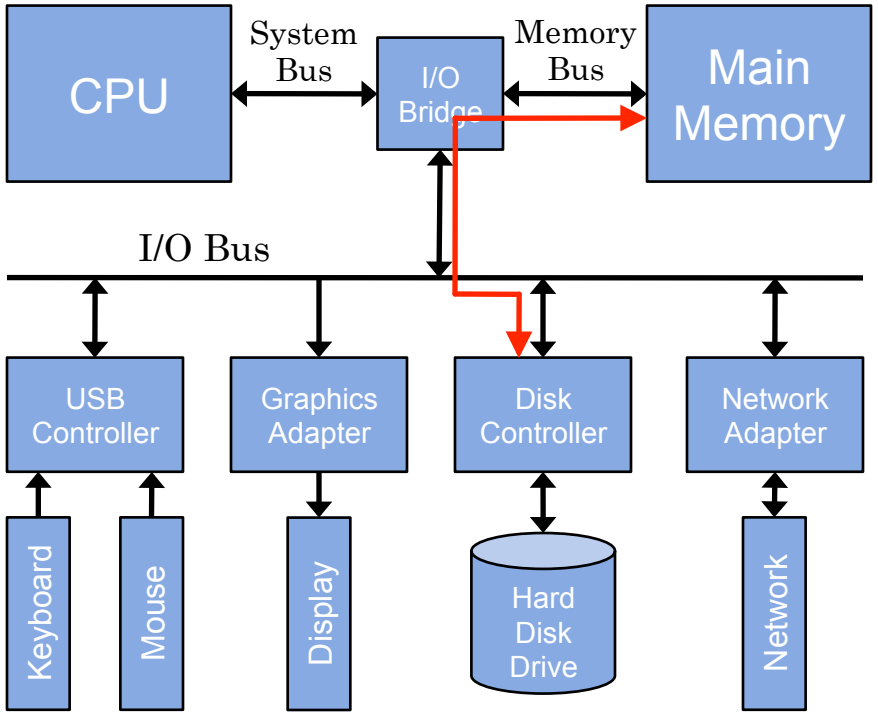
UNIX System Calls

- Common theme across many system calls:
  - A large number involve long-running operations
  - Some involve interacting with other processes
- The kernel frequently performs context-switches when system calls are made
- Example: two programs running concurrently
  - Program A executes a **`read()`** call
    - Read a block of data from disk
    - Will be waiting 8+ ms for data!
  - Program A transfers control to the kernel…
    - Kernel initiates the disk read, and then goes to do other stuff in the meantime



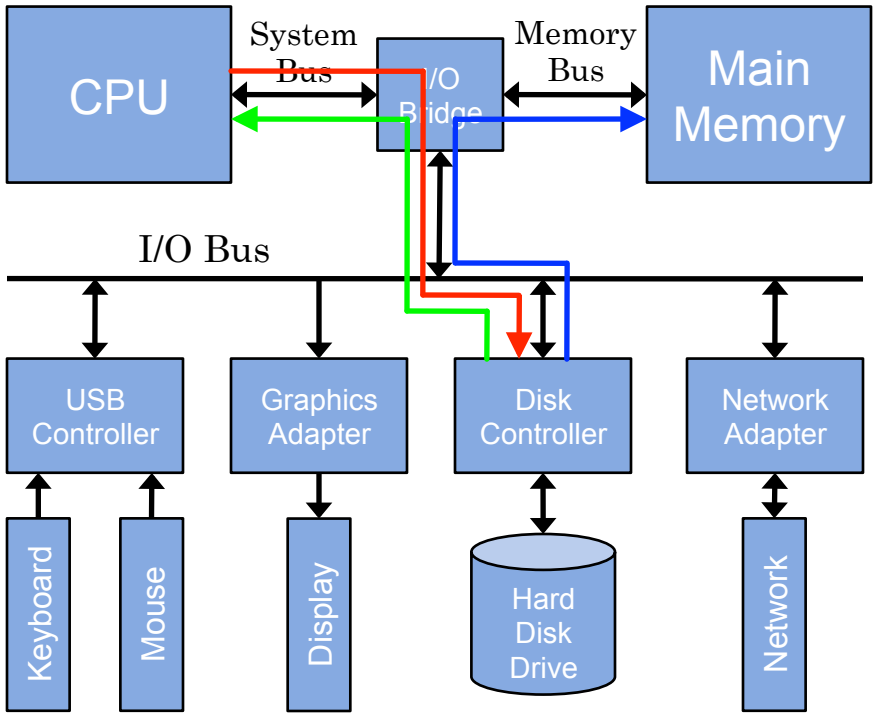
Direct Memory Access

- Some peripherals can access memory directly
  - Allows CPU to do other things during the data transfer
- Example:
  - CPU tells disk controller to move a disk block to main memory
  - Disk controller moves data to main memory on its own, without CPU intervention
- Called Direct Memory Access (DMA)
  - Interaction between disk controller and memory is called a DMA transfer



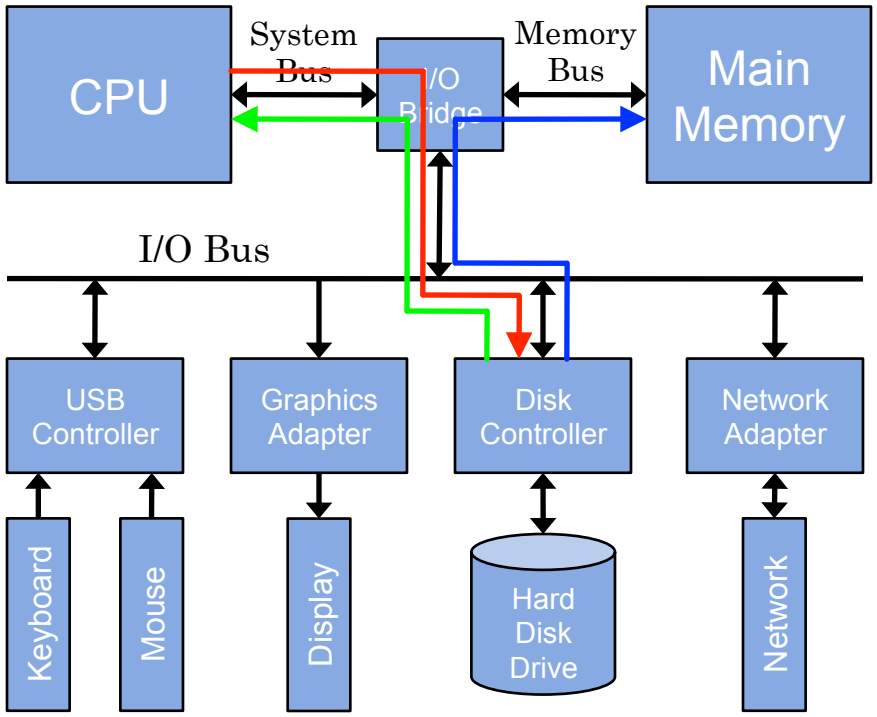
DMA Transfer Sequence

- Step 1: CPU tells disk controller to read a block of data into memory
- Step 2: Disk controller performs a DMA transfer into memory
- Step 3: Disk controller signals an interrupt to inform CPU that transfer is complete
- Result:
  - Operating system can do other work while slow operations take place!



Direct Memory Access Notes

- Direct Memory Access is essential for modern high-performance computing!
  - Used by disk controllers, graphics cards, sound cards, networking cards, etc.
- Buses must support multiple "bus masters"
  - An arbiter must resolve concurrent requests from multiple bus masters
- While DMA transfers take place, CPU access to memory is slower
  - CPU will hopefully be using its caches…



UNIX Signals

- Would like to provide exceptional control flow to user-mode programs as well
- UNIX platform provides signals for processes
  - Some signals are hardware exceptions routed through the kernel to a specific process
    - e.g. timer interrupts, memory access faults
  - Other signals originate directly from the kernel or other processes
    - e.g. kill or suspend a process
- Programs register a signal handler for the signal
  - Signal handler must call only reentrant functions – functions that can be executed multiple times concurrently
  - e.g. **`malloc()`** and **`printf()`** are *not* reentrant!

UNIX Processes

- UNIX provides a powerful yet simple abstraction for processes
- Every process has a unique Process ID (PID)
  - Also has a parent process, and a process group
- Can start a process by calling **`fork()`**
  - **`fork()`** is called once and returns twice!
  - Child process is an identical duplicate of the parent, except for process ID, parent process ID, etc.
- A process can terminate by calling **`exit()`**
- Send signals to other processes using **`kill()`**
  - Depending on signal, and on handlers in the process, may or may not terminate the receiver

Process Termination and Reaping

- A child process doesn't immediately go away when it terminates
  - Child process terminates with some status value…
  - Parent process may need to find out the child's status
- A terminated child process is called a zombie
  - The process is dead, but it hasn't yet been reaped
- Parent processes reap zombie children by calling:

```
pid_t wait(int *status)
```

  - Waits for some child process to terminate

```
pid_t waitpid(pid_t pid, int *status, int options)
```

  - Waits for a specific child process to terminate
  - Can also wait on children in a process-group, or all children

- Unreaped zombie children are reaped by **`init`**
  - The first process started by the operating system, PID = 1
- Orphaned child processes are also inherited by **`init`**

Loading and Running Programs

- The **`execve()`** function is used to load and run a program in the current process context

```
int execve(char *filename,
           char *argv[], char *envp[])
```

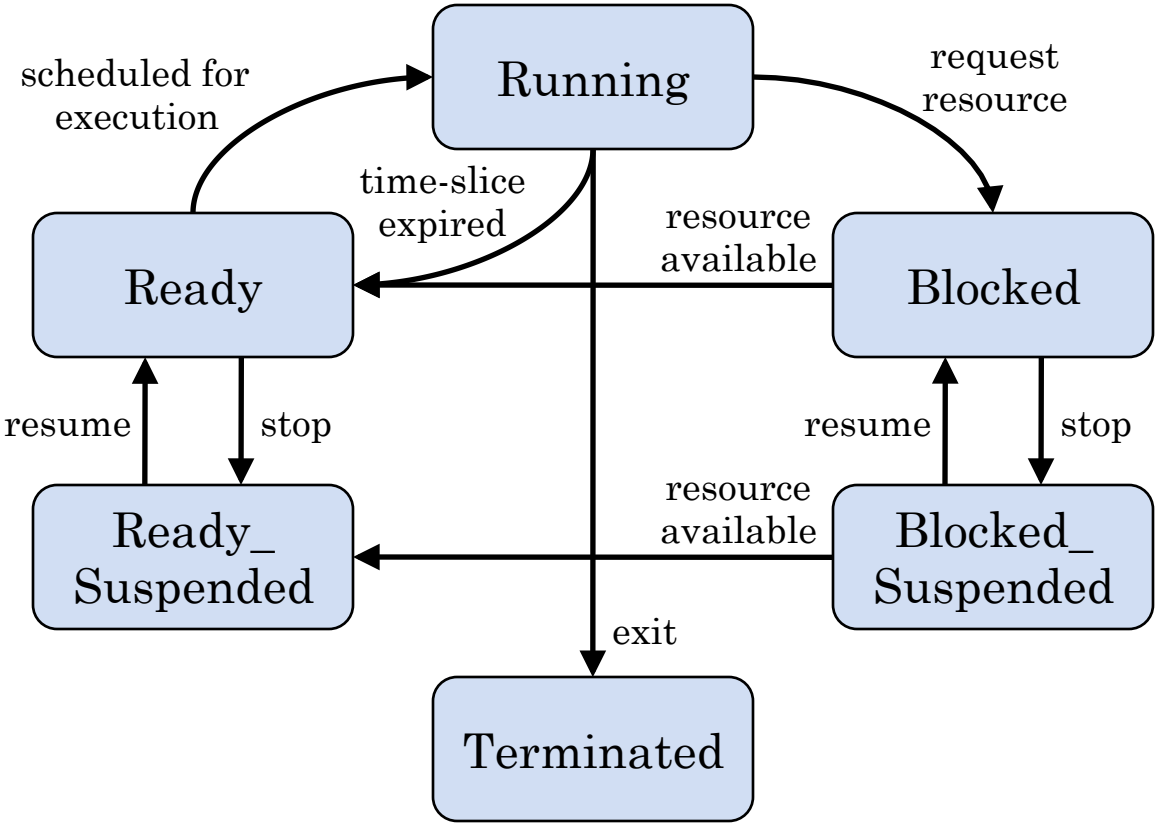
  - Loads the program into the current process context
  - Program arguments are specified by **`argv`**
    - These are the command-line arguments
    - Last element in argv must be **`NULL`**
  - The program's environment is specified in **`envp`**
    - Each environment variable is a "NAME=VALUE" pair
    - Last element in envp must be **`NULL`**
- On success, loaded program *replaces* the current process' context and starts running from its start
  - On success, **`execve()`** does not return!

Process Control Block

- Each process has a Process Control Block (PCB) associated with it
  - Contains all information necessary for managing the process, and for performing context-switches
- The PCB can contain a lot of information:
  - Process ID, parent and child process IDs
  - When not running, register and memory state of process
  - Information about resources the process is using
  - Pending resource-requests that need to be filled
  - Scheduling information
  - *etc.*
- All necessary to allow kernel to coordinate processes using different resources on a single physical system

UNIX Process State Diagram

- Process states follow a well-defined model
  - Only one Running process per CPU/core! Other processes are Ready, or Blocked on slow resources
  - Can also suspend processes, or terminate them



Process Scheduling

- Kernel must also decide what process to run next
  - Of all processes currently in the Ready state, choose one that gets to use the CPU next
- Scheduler must be fair, and must properly balance needs of interactive processes, compute-intensive processes, and real-time processes
  - Scheduler also needs to choose next process quickly…
- Several different scheduling algorithms for different kinds of processes

Process Scheduling (2)

- Round-Robin scheduling:
  - Each process gets a relatively large time-slice on the CPU, in sequence
  - Best for compute-intensive processes, but terrible for interactive or real-time processes
- Shortest Job First scheduling:
  - If the job length can be estimated, always execute the shortest job first
  - Great for interactive processes, but slow and even unfair for compute-intensive processes
- Earliest Deadline First scheduling:
  - Process that is closest to its next deadline is run first
  - Designed for real-time processes, which often have a periodic scheduling requirement

Process Scheduling (3)

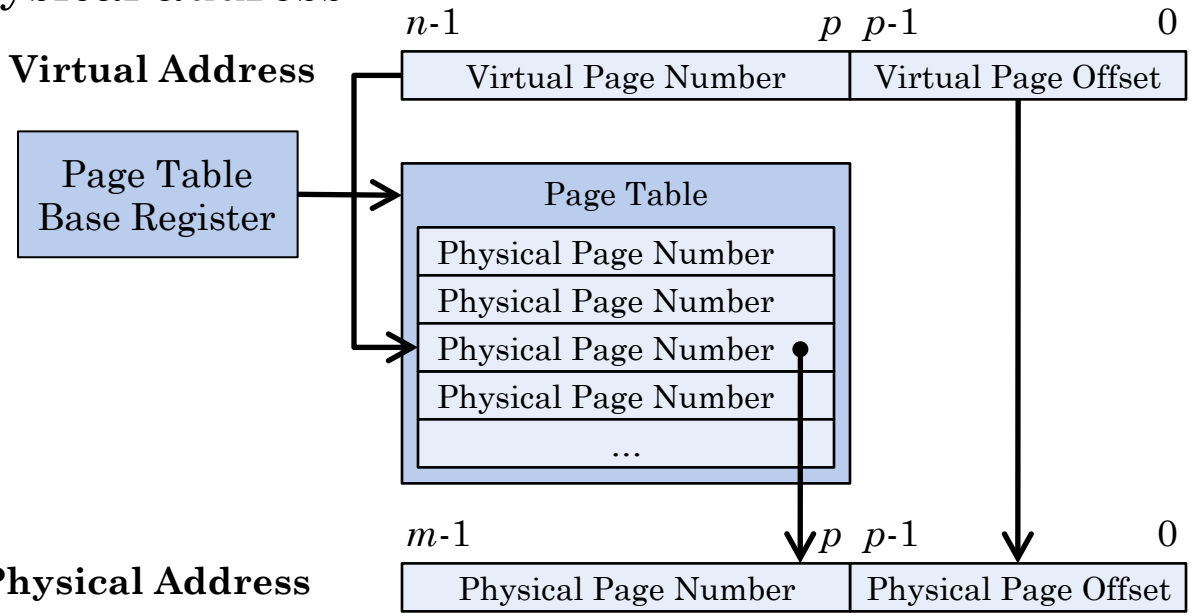
- As usual, find a way to combine these approaches to provide a generally capable scheduler
- Multi-Level Feedback Queue scheduling:
  - Multiple queues, each with its own time-slice size
  - Move processes between queues based on whether or not they stay within their queue's time-slice size
  - MLFQ scheduler quickly categorizes all running processes based on their recent behavior
  - Can identify the interactive and compute-intensive processes, and handle them appropriately
- Must still tell the scheduler some details…
  - e.g. manually specify that a process is real-time
  - Specify process' priority, if user expects it to take long

Virtual Memory

- Many CPUs also support *memory virtualization*
  - Instead of directly addressing physical memory, instructions use *virtual* addresses that are mapped to physical addresses
- Main memory provides a *physical address space*
  - Size of M bytes (frequently, $M = 2^m$, but not required)
  - Computer provides an m-bit address space
- Define a *virtual address space* of size N bytes
  - $N = 2^n$, so this is an n-bit virtual address space
  - Not required that $M = N$
- Divide memory up into pages of size P, $P = 2^p$
- CPU maps each virtual page to a physical page

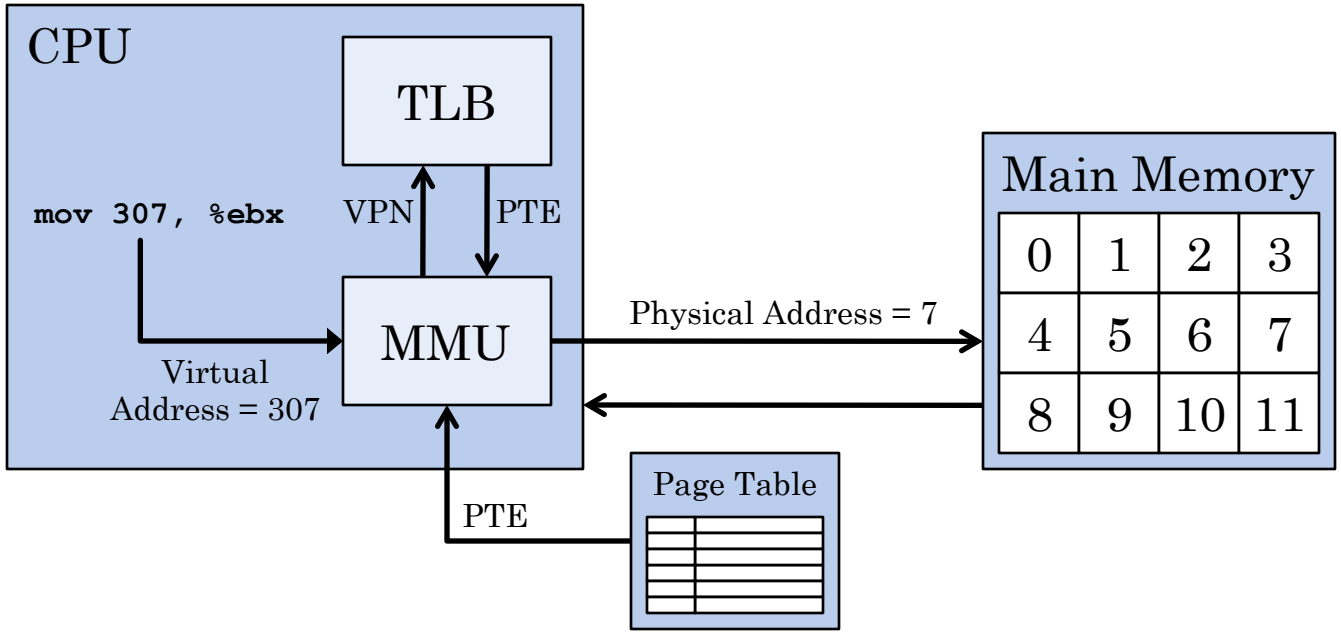
Address Translation

- Address translation is implemented in hardware
  - Memory Management Unit on the CPU
- MMU uses a page table to translate virtual pages to physical pages
  - Page table is indexed with the virtual page number
  - Page table entry contains the physical page number
  - Combine physical page number with virtual page offset to get a physical address



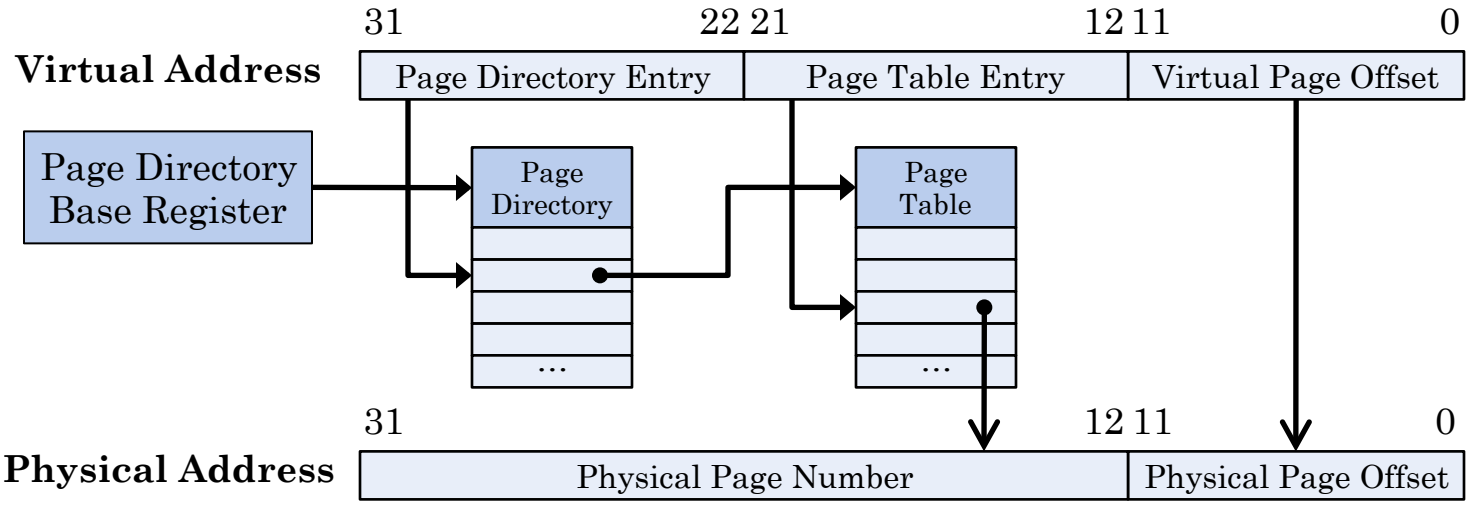
Translation Lookaside Buffer

- To maximize translation performance, MMU includes a Translation Lookaside Buffer (TLB)
- TLB stores one page table entry per cache line
  - Input is virtual page number, output is page table entry
- TLB typically has a high degree of associativity
  - Maximize chance that TLB contains needed page table entry!



Multi-Level Page Tables

- With small pages (e.g. 4KB) and a large address space (e.g. 4GB), page tables can be prohibitively large
- Use a hierarchy of page tables to take advantage of typically sparse process memory layout
- Intel IA32 implements a two-level hierarchy
- IA32 calls the Level 1 table a page directory
  - Top 10 bits of virtual address is page directory entry
  - Next 10 bits used for page table entry
  - Bottom 12 bits used for virtual page offset



Page Table Entries

- Page table entries include an address to physical memory, as well as a *valid* bit
- *valid* = 1 means page is cached in DRAM memory
  - Corresponding address is the start of the page in DRAM physical memory
- *valid* = 0 means the page is not cached in DRAM
  - If stored address is 0, the virtual page is not allocated
  - If address is not 0, the virtual page is allocated, and the address is the page's location on disk
- IA32 page-table entries call this the **Present** bit, i.e. "Is the page present in main memory?"

Processor and Operating System

- *A very important detail about virtual memory facility:*
- The processor provides *some* hardware support for virtual memory
  - Address translation is performed in hardware, using the Memory Management Unit
  - MMU must refer to a page table to perform address translation
- The processor cannot completely support virtual memory in hardware, on its own!
  - What to do when a virtual page is not in main memory? CPU has no idea how virtual memory pages are cached.
- The operating system must also provide support for many virtual memory operations
  - CPU does as much as it can on its own…
  - CPU invokes exceptions when it needs the kernel's help!

Virtual Memory and Faults

- MMU raises a fault for two main reasons:
  - Page table entry says that the page isn't present
  - Page table entry says that the access isn't allowed
- Kernel is informed, and resolves the fault:
  - If access doesn't refer to a valid address, send a segmentation fault signal to the program
  - Otherwise, if the access is invalid, send a protection fault signal to the program
    - …unless the memory area is marked private copy-on-write!
  - Otherwise, the access is to a valid address and is a valid operation, but the page is not in DRAM
    - Choose a page to evict from physical memory
    - Load the requested page from disk into physical memory
- CPU resumes instruction that caused the fault

Virtual Memory Capabilities

- Virtual memory is a simple idea with many benefits:
  - Context-switches are *much* faster
  - Processes have their own isolated virtual address spaces
  - CPU handles mapping of virtual addresses to physical addresses automatically
  - A process' physical memory layout doesn't have to be contiguous
  - Can map multiple virtual pages to the same physical page
- With file-backed virtual memory:
  - Loading and running programs is greatly simplified
  - Multiple processes can share a single copy of the program binary, and shared libraries that many programs use
  - Can easily provide features like private copy-on-write
  - Allows programs to use very fast memory-mapped I/O

CS24 Wrap-Up

- That's it for CS24… hope you enjoyed the class!
- More importantly, hope you have a much deeper understanding of computers you use every day
  - What basic features and optimizations are provided by the CPU and computer hardware?
  - How does your program get translated into instructions the computer can understand?
  - How does the CPU execute your program?
  - What role does the operating system play, and what CPU/hardware capabilities does it rely on?
- Hope you have a better idea of how to get most out of the computer hardware and software, too
  - How to make your programs faster, more scalable, and more reliable

CS24: The Big Ideas

- Several very important system design themes during the term
  - Beyond the details of specific subsystems and hardware/software features…
  - General approaches to problems that are very useful
- Common-case optimization:
  - "Make the common case fast."
    - Specialized hardware to optimize system performance
  - "Make the fast case common."
    - Programmer and compiler adjust program behavior to take advantage of high-performance hardware capabilities
  - e.g. hardware caches and locality in data access, virtual memory and context-switches

CS24: The Big Ideas (2)

- Hybrid solutions:
  - Often have multiple techniques for solving a given problem, with different strengths and weaknesses
  - *Combine* these techniques to provide a generalized and broadly applicable solution
  - e.g. garbage collection, set-associative caches, multilevel feedback queue process-schedulers
- Virtualization:
  - Present an abstracted version of computer hardware to the programs that are using it
  - Requires an operating system to mediate access to the hardware, on behalf of user applications
  - e.g. processes, virtual memory