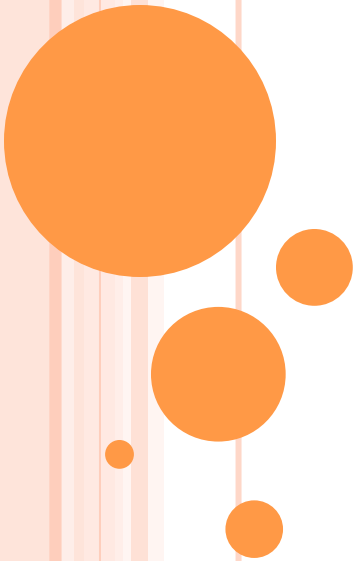


CS24: Introduction to Computing Systems

Spring 2015

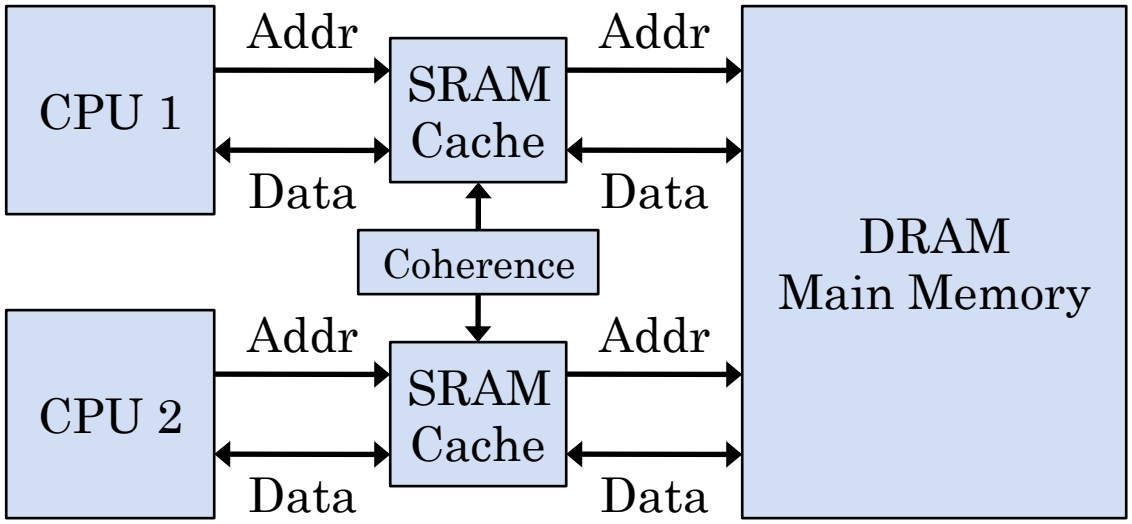
Lecture 16

Last Time

- Processor clock speeds have leveled off…
  - Physics imposes practical limits on clock frequencies
  - Instead, focus on increasing processor capabilities
  - "Do more with less"
- One increasingly common technique: multi-core
  - Put multiple processors into a single chip
  - e.g. Core 2 Duo (2 cores) or Core 2 Quad (4 cores)
  - Core i7: multiple cores, plus Hyper-Threading to interleave execution of multiple threads
- Problem: CPU is still *way* faster than memory
  - Still need caches between main memory and the CPU

Multicore and Cache Coherence

- Multiple caches of a single shared resource:
  - Obviously requires coordination between independent caches to avoid consistency issues



- Implement a *cache coherence protocol* to coordinate memory accesses between caches

Cache Coherence Protocol

- Intel and AMD multi-core processors use variants of the MSI protocol to coordinate memory accesses
- In each cache, a cache line is in one of these states:
  - **Modified** – the line contains data modified by the CPU, and is thus inconsistent with main memory
  - **Shared** – the line contains unmodified data, and appears in at least one CPU's cache
  - **Invalid** – the line's data has been invalidated (e.g. because another CPU wrote to their cache), and must be reread
    - …either from main memory, or from another cache
- Can use these states to determine how to respond to various cache-access scenarios
  - (See previous lecture for details!)

MSI Protocol Variants

- Several variants of MSI protocol, that implement various optimizations
- MESI variant introduces an Exclusive state
  - Cache line contains unmodified data, and it only appears in one cache
  - Idea: a processor doesn't need to tell other caches to invalidate the line if it's in the Exclusive state
  - Intel multi-core processors use MESI protocol
- MOSI variant introduces an Owned state
  - A modified block in a cache can be marked as Owned instead of Modified, if reads and writes are expected
  - If other cores read the modified block, the owning cache serves the data to the other cores
  - Idea: reduce frequency of cache write-backs

MSI Protocol Variants (2)

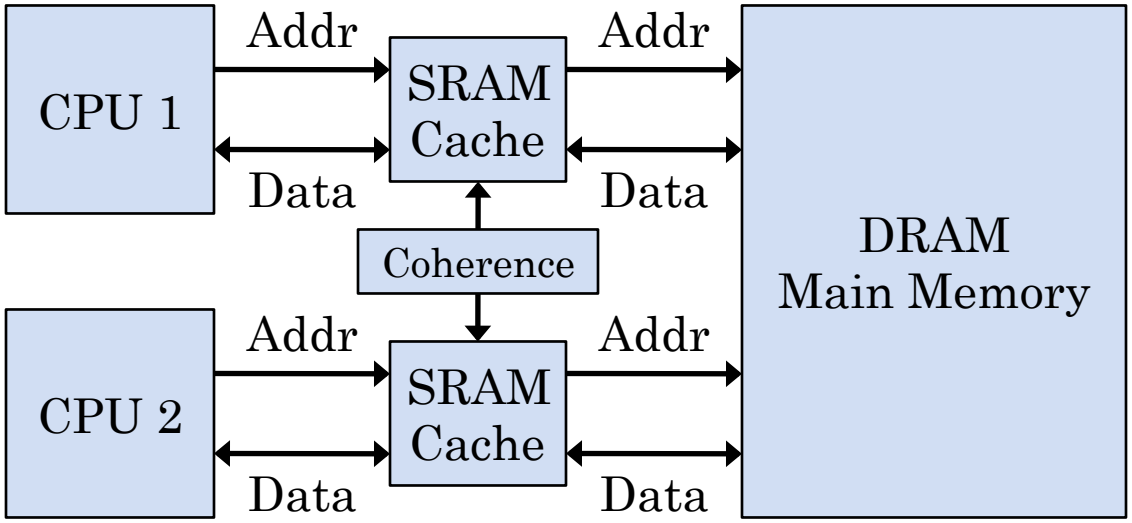
- AMD processors use MOESI coherence protocol
  - Achieves benefits of both MESI and MOSI
- Also *many* variations on implementation details
  - Some allow moving of cache lines directly between caches, instead of only through main memory
    - Like MOSI; provides a much faster data path between cores
  - Some caches use *bus-snooping* (a.k.a. *bus-sniffing*) to monitor the state of other caches
    - Observe other caches' operations to figure out what to do
    - Some caches use *bus-snarfing* to update their own cached data when another core writes to its own cache
  - Others use central directory to share cache-line state
    - Directory-based approach scales much better than bus-snooping as the number of cores increases

MSI Protocol Variants (3)

- Intel Core i7 processors use MESIF protocol
  - Like MESI protocol, but introduces a Forward state
- Some caches can share data with each other…
  - When a cache needs to load a new line, other caches can serve the line if they have it in the Shared state
  - Problem: if multiple caches have a line in the Shared state, the requesting cache gets multiple responses
- Forward state:
  - When more than one cache has a given line in Shared state, one cache has the line in the Forward state
  - That cache is responsible for serving the line to other caches, if they request it

Coherent Isolated Caches

- Can definitely solve the cache coherence issue



- What other problems can we run into?
  - Have resolved our correctness issue…
  - And now on to the performance issues…

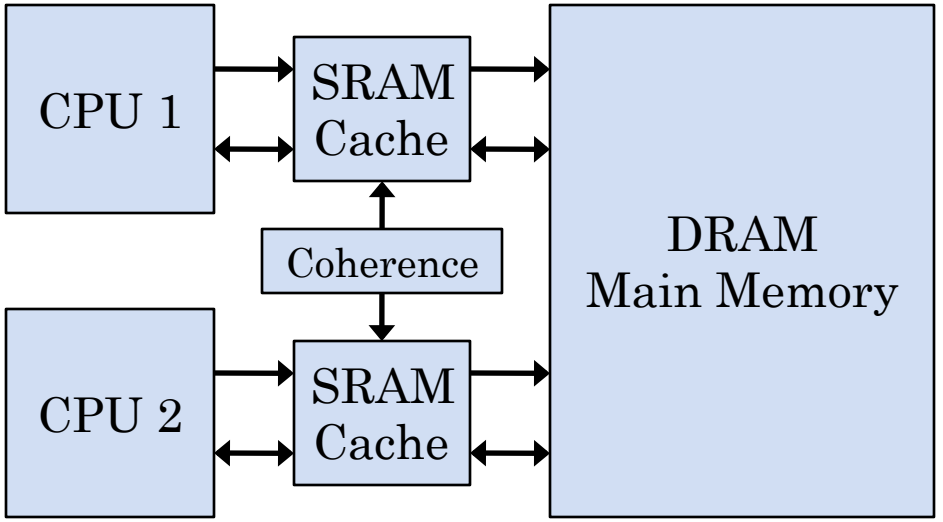
Single-Threaded to Multi-Threaded

- Example scenario:
  - Want to update a program from a single-threaded implementation to a multi-threaded implementation
  - Multiple threads allow us to take advantage of multi-core
- Our example program stores its data in an array
  - `float x[1000];`
  - `sizeof(float)` is 4 bytes
  - Good data locality for our single-threaded program
- Program can update elements of `x` independently of each other…
- Idea:
  - Have different threads update different elements of `x`
  - Then, different threads can run on different cores
  - *Profit!*

Multi-Threaded Program

- Now our program updates **x** in multiple threads
  - **x** is an array of 1000 floats
  - Each float uses 4 bytes
  - Contiguous sequence of 4000 bytes in memory
- Our multi-core processor:
  - Each cache line stores 16-dword blocks
- Problems?

Thread 1 (on CPU 1)	Thread 2 (on CPU 2)
… x[3] = compute_x(…); … x[5] = compute_x(…); …	… … x[4] = compute_x(…); … x[6] = compute_x(…);



False Sharing

- Each thread is manipulating completely independent data

Thread 1 (on CPU 1)	Thread 2 (on CPU 2)
… x[3] = compute_x(…); … x[5] = compute_x(…); …	… … x[4] = compute_x(…); … x[6] = compute_x(…);

  - Thread 1 doesn't care about thread 2's values
  - Thread 2 doesn't care about thread 1's values
- However: the independent values being updated happen to reside in the same cache lines!
  - To maintain coherence, caches are doing *tons* of work!
- Problem is called false sharing
  - A single cache line contains several independent values, updated by different processors
  - Caches must move cache line back and forth to compensate

False Sharing (2)

- Caches must coordinate operations on cache lines
  - When a CPU writes to a cache line, other caches must invalidate the line
  - If modified, must write cache line back to memory

Thread 1 (on CPU 1)	CPU 1 Cache	Thread 2 (on CPU 2)	CPU 2 Cache
...		...	
x[3] = ...;	Load line x[0..7].	...	
...	*Invalidate line!*	x[4] = ...;	Load line x[0..7].
x[5] = ...;	Load line x[0..7].	...	*Invalidate line!*
...	*Invalidate line!*	x[6] = ...;	Load line x[0..7].

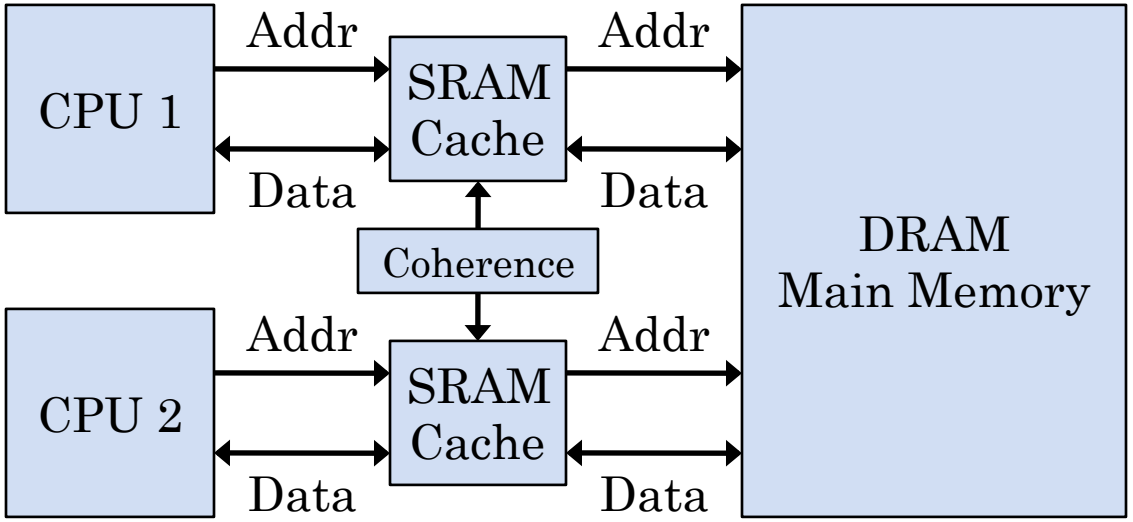
- Cache line *ping-pongs* between CPU 1's cache and CPU 2's cache
- False-sharing overhead severely impacts performance
  - One test showed a 100x slowdown due to ping-ponging…
- (Protocols with Owned state mitigate this somewhat)

Avoiding False Sharing

- Simple solution to false-sharing problem:
  - Make sure that each cache line only contains data updated by one thread
- For the example program:
  - Threads should update a contiguous group of `x[i]` values whose size is a multiple of the cache-line size
  - If program can't predict which elements threads will update, just pad array-elements out to cache-line size
    - Wastes space, but makes the program much faster
- Moral:
  - With multi-core, simple data locality is no longer the sole consideration!
  - Must also think carefully about impacts of cache coherence on cache behavior

Cache Utilization

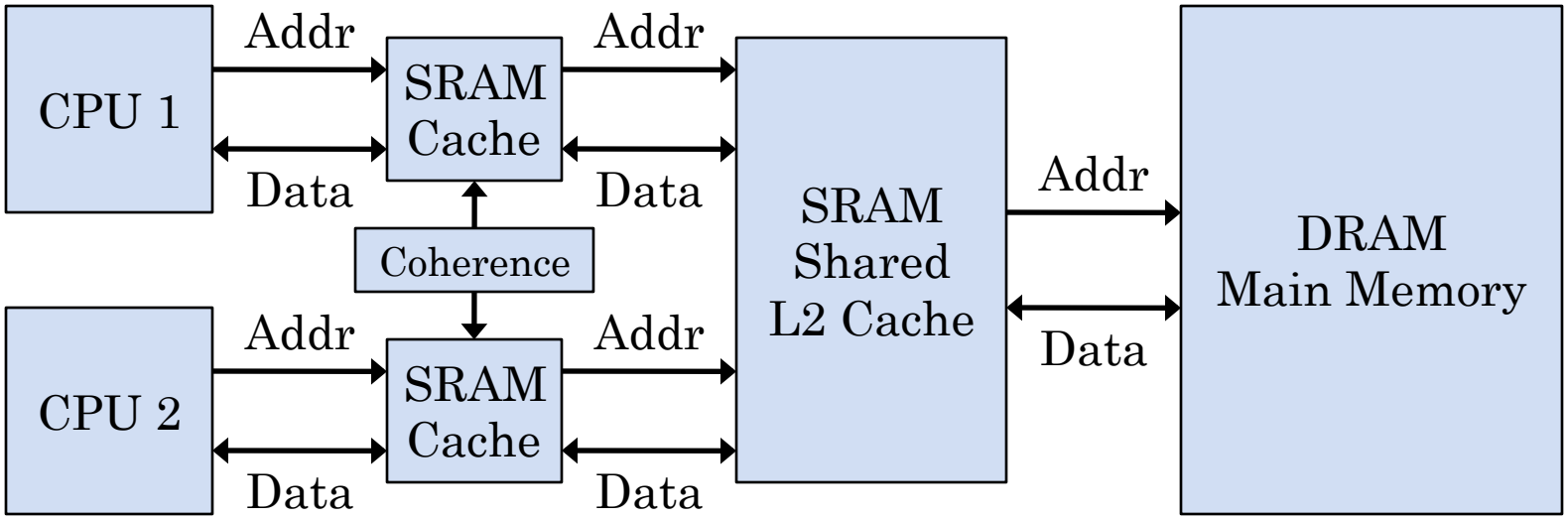
- Different cores aren't always performing similar tasks



- Example:
  - CPU 1 is running your MATLAB program…
  - CPU 2 is running the web browser while you wait
- CPU 1 is utilizing its entire cache; CPU 2 is not.
- Would like to dynamically shift cache resources between processors as they need it!

Shared L2 Cache

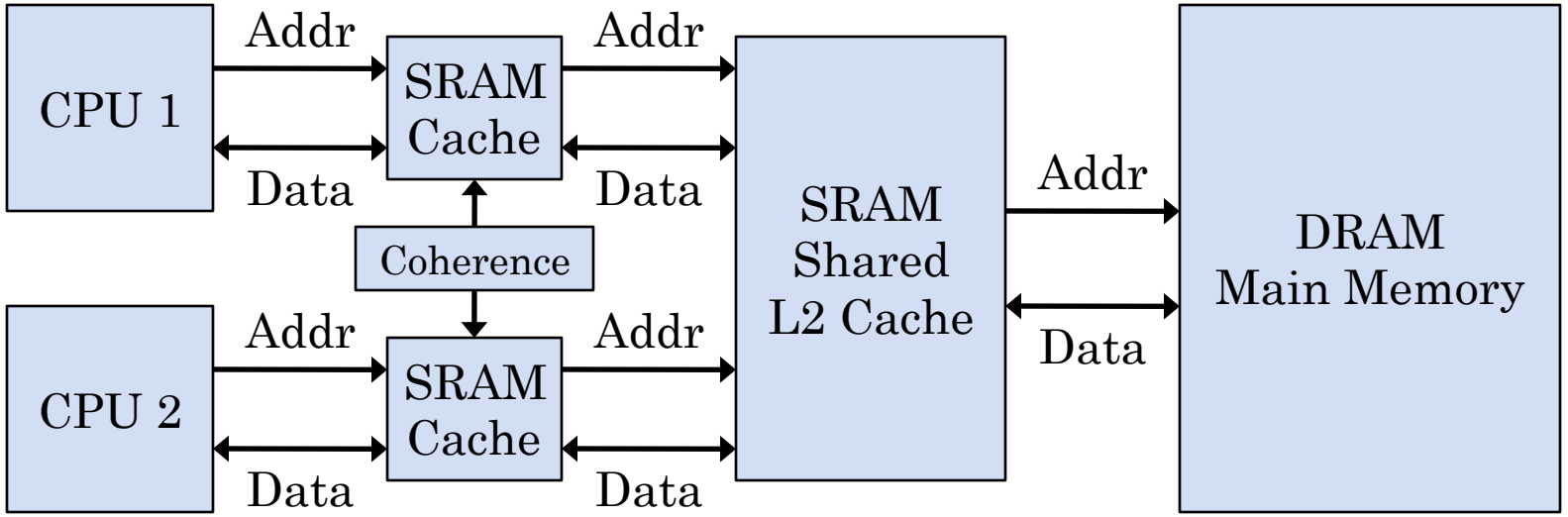
- Provide a shared L2 cache for all cores to use
  - Typically, L2 cache is on-chip for max performance



- Now, CPUs will *proportionally* utilize the shared cache based on their needs
  - While CPU 1 runs MATLAB, it uses most of the shared L2 cache

Shared L2 Cache and Data Sharing

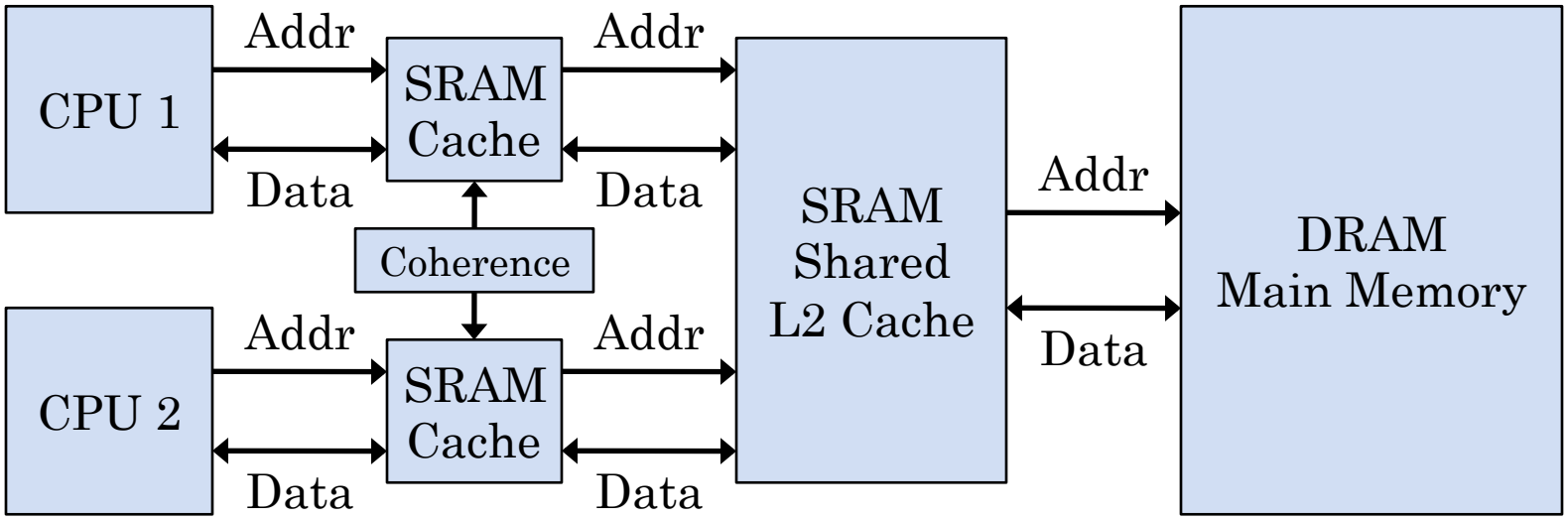
- Shared L2 cache also provides interesting opportunity for cores to share data through high-speed L2 cache



- Simple example:
  - CPU 1 writes a small chunk of data to memory; blocks get cached in its own L1 cache
  - CPU 2 reads the same memory; blocks are moved thru L2 cache to CPU 2's L1 cache without involving main memory

Shared L2 Cache and False Sharing

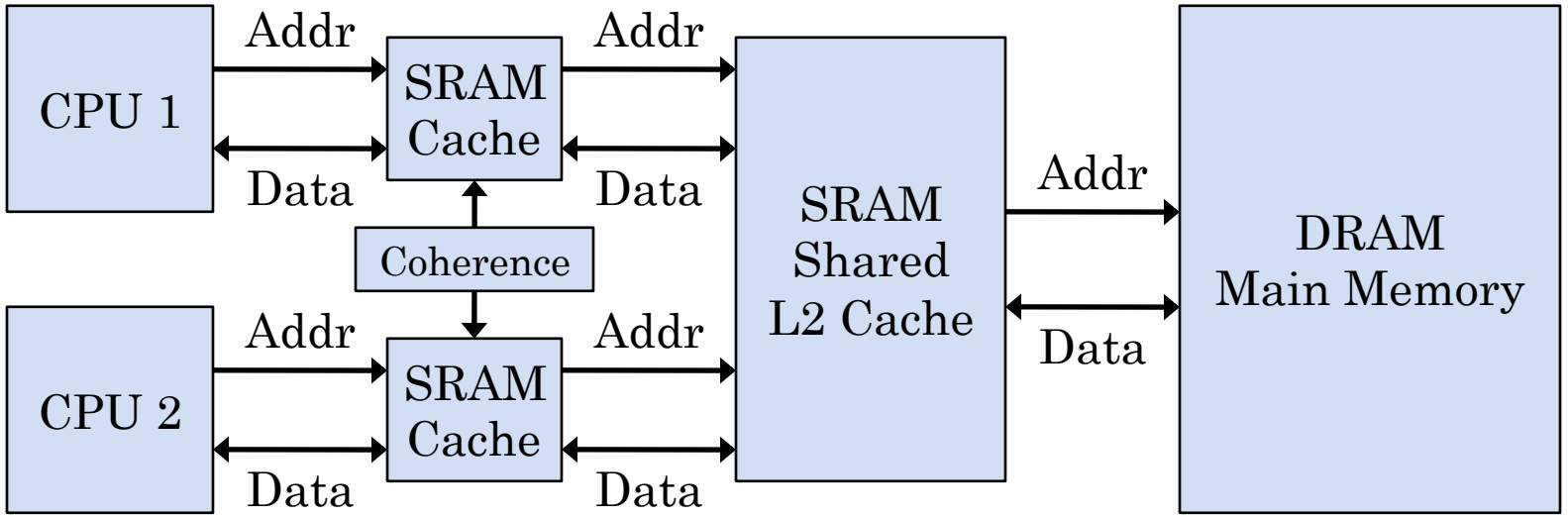
- False sharing can still occur with a shared L2 cache, but penalty is greatly mitigated



- Only a few extra clocks per L1 cache-miss, since data is in L2, instead of 50-100 clocks per L1 cache-miss!
- Even with shared L2 cache, eliminating false sharing can still produce significant speedups
  - Core 2: L1 cache-hit = 3 cycles, L1 cache-miss = 14 cycles. Nearly 5x slower to ping-pong cache lines through L2 cache!

Intel Core 2 Caching

- Intel Core 2 processor uses this architecture



  - Independent L1 caches. 32KB; 8-way set-associative; blocks are 8 dwords in size.
  - Shared L2 cache. 2-6 MB; 8-way set-associative; blocks are 8 dwords in size.
  - MESI cache-coherence protocol to coordinate L1 cache writes between processors

Summary: Multi-Core

- As usual, multi-core introduces interesting wrinkles into hardware caching details
- Multiple L1/L2 caches must be kept consistent
  - Cache coherence protocols such as MSI and variants
  - Significantly increases the complexity of cache logic!
- Several benefits from including a shared cache
  - Improves overall cache utilization when cores require different amounts of cache
  - Provides a high-speed channel for cores to share data
- Important new caching performance issue:
  - False sharing will dramatically slow down a program!
  - Must avoid potential for a cache line to contain independent values updated by different processors

Processors and Programs

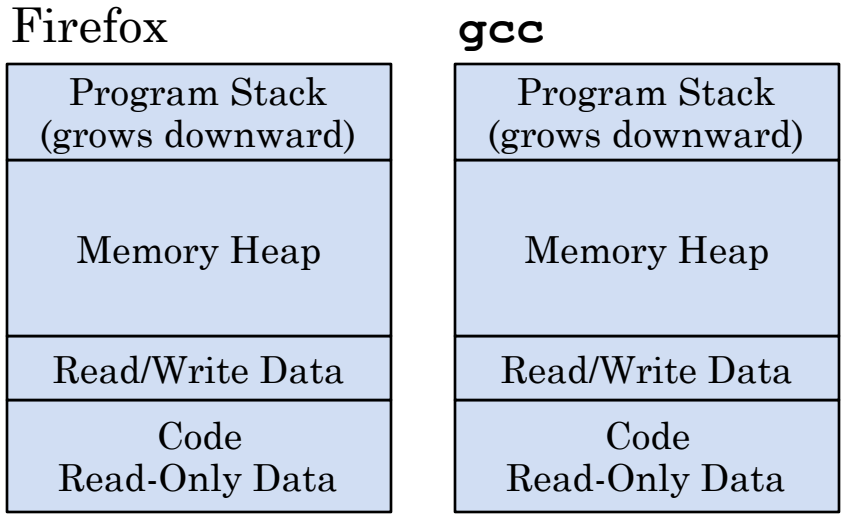
- So far, have run only one program on the processor at a time
- Programs don't normally consume all resources the computer has to offer!
- Programs spend a *lot* of time waiting:
  - For data to be read/written to disk (10s of ms disk latency)
  - For data to be read/written to network (10s to 1000s of ms latency)
  - For user interactions! (Seconds, minutes, hours!)
- Even more obvious with multi-core processors
  - Clearly, a dual-core processor can run at least two programs at once…

Processors and Programs (2)

- Want to be able to run multiple programs on our computer at once
  - Different programs, or even multiple instances of the same program
- What constraints should the computer enforce on concurrently running programs?
  - *Running programs shouldn't meddle with each other!*
  - Shouldn't be able to access each other's data
  - A crash shouldn't cause other programs to crash
  - Need to isolate running programs from each other

Loading Multiple Programs

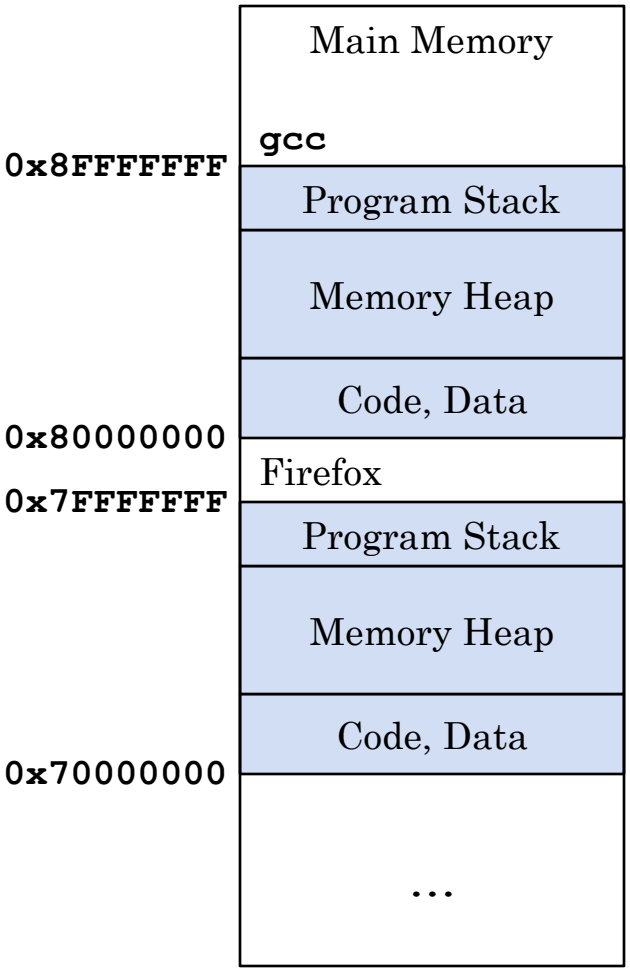
- How do we load and run concurrent programs?
- Example: want to run Firefox, **`gcc`** at same time
  - Each program has its own code and data
  - The code needs to refer to its state, somehow…
  - Variables are normally turned into absolute addresses at compile time



- How do we load both programs into our computer's single, unified address space?

Loading Multiple Programs (2)

- One idea: assign each program a specific address range
  - Firefox always gets addresses **`0x70000000-0x7FFFFFFF`**
  - **`gcc`** always gets addresses **`0x80000000-0x8FFFFFFF`**
- This has *all kinds* of problems!
- Here is a small list:
  - What if a program's memory needs grow?
  - What if a computer has less, *or more,* memory?!
  - What if I want to run <u>two</u> instances of **`gcc`** at same time?!

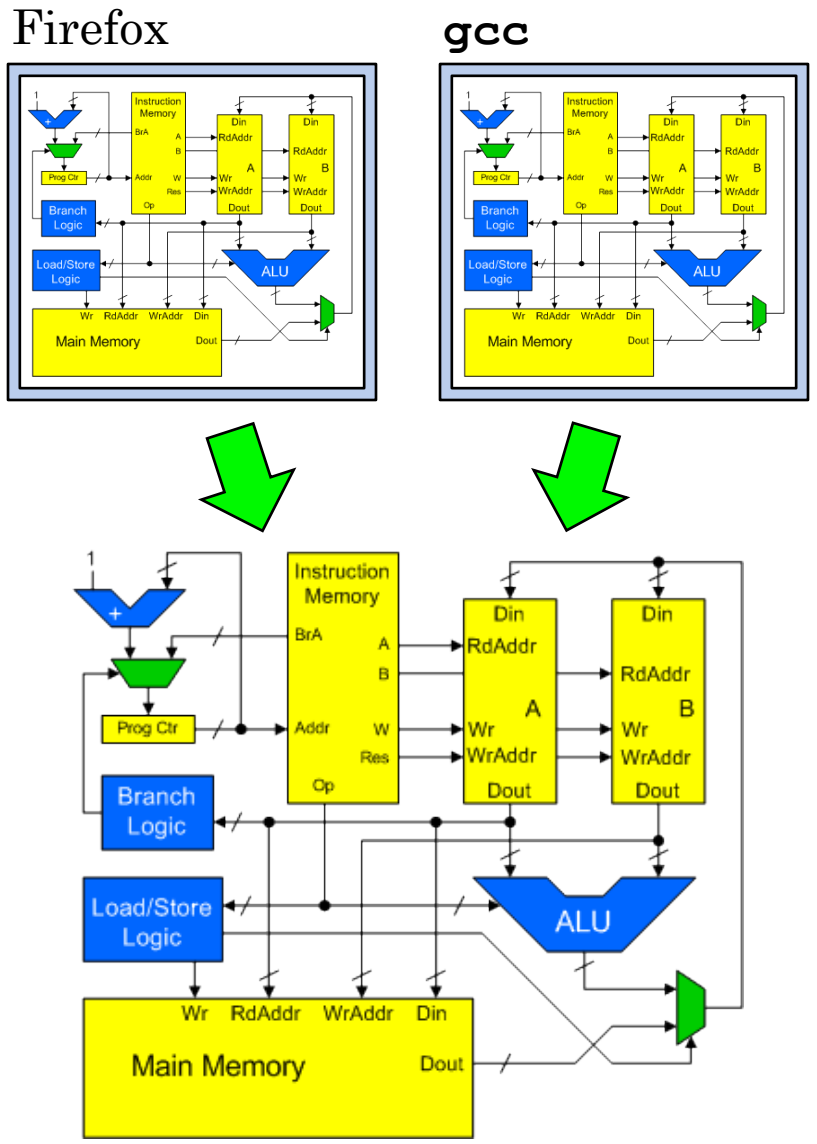


Processor and Memory Abstraction

- What we really need:
  - Want to give each program the perception that it is *the only* program running on the computer
- Programs have completely isolated address spaces from each other
  - Can even store data at "the same address" as other programs
  - Somehow, the processor will sort this out for us ☺
  - This is *essential* to be able to run multiple instances of the same program on one computer
- Programs also have independent views of the processor from each other
  - e.g. Firefox doesn't have to worry about what registers **`gcc`** uses… It just does whatever it wants.

VIRTUALIZATION

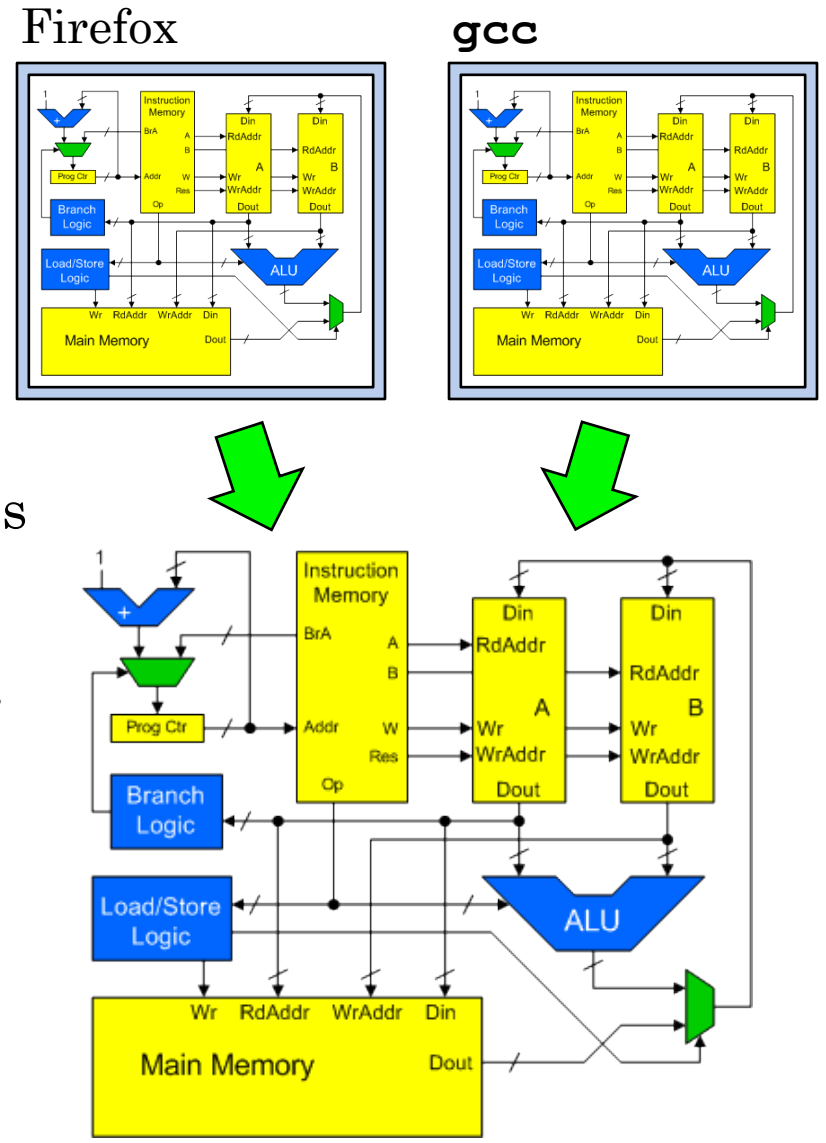
- Virtualize the processor
  - Make it look like we have multiple processors
  - Each program runs on "its own processor"
- Introduce another level of abstraction
  - The machine that the program sees is different from the actual machine
- Implement a mechanism that allows us to share one physical processor across multiple virtual processors



VIRTUALIZATION (2)

- Similarly, virtualize main memory
  - Make it look like each program has sole access to main memory
  - Each program's memory is isolated from other programs
  - Programs can use whatever memory layout they wish, without affecting each other

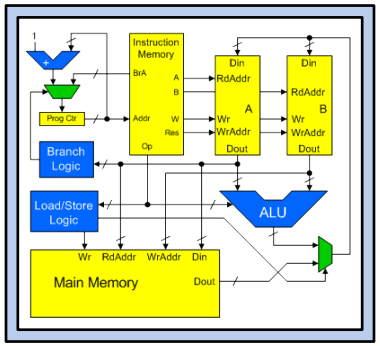
- Concept of virtualization is central to modern computers and OSes



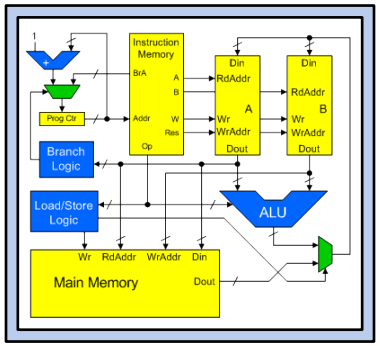
Processes

- This 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

Firefox



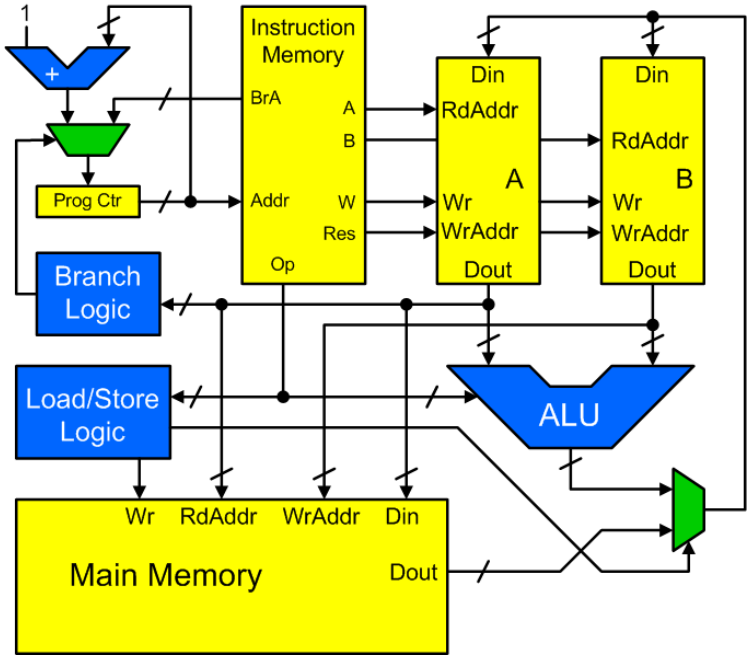
gcc



- The running program's state is called its context
  - Each process has a context associated with it

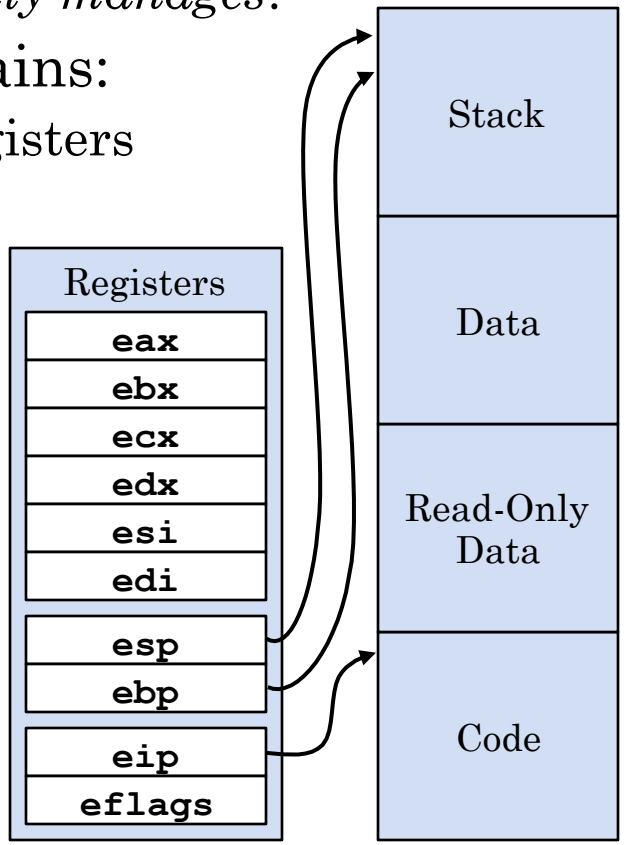
Program Context

- The physical processor can still run only one program at a time…
  - Only one program counter, instruction memory, ALU, register file, main memory, etc.
- But, if we can capture each running program's context:
  - We can *simulate* concurrently executing programs by giving each program its own turn to run on the physical processor
- When a process is running, it has exclusive access to the processor hardware…
  - …until it's suspended and another process is given a turn.



Program Context (2)

- What state does a running program actually have?
  - *What is the state that a processor actually manages?*
- On IA32, the program's context contains:
  - Current state of all general-purpose registers
    - **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**
    - (Also need to capture floating-point registers, etc! Ignore for now…)
  - Current program counter: **eip**
  - Current stack pointers: **esp**, **ebp**
  - Also, current state of **eflags** register (see **pushfl**/**popfl**)
- Context also needs to include the program's in-memory state
  - Virtual memory abstraction makes this easy to solve (more on this later!)

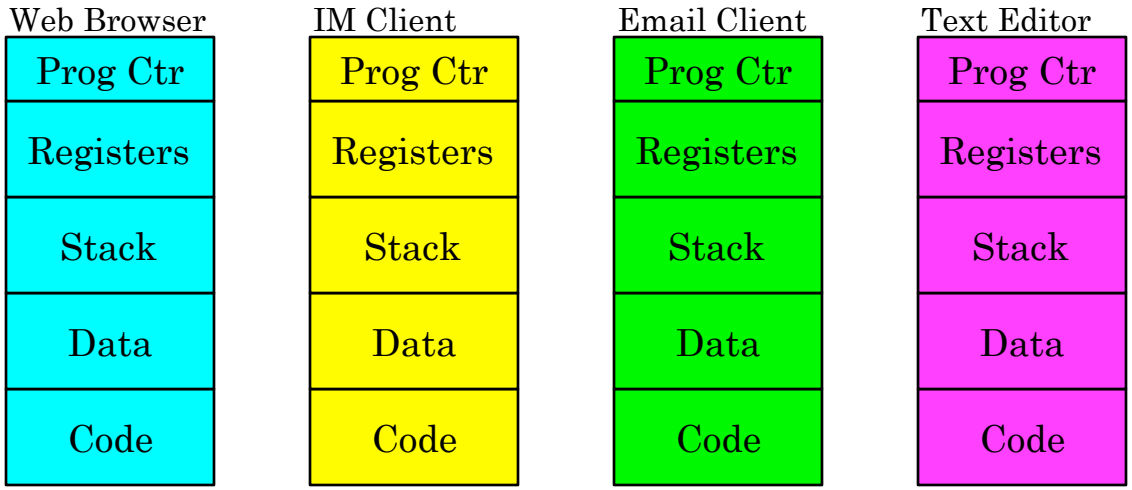


Switching between Processes

- We can capture all state associated with a running program (i.e. a process)
  - Save it into memory somewhere for later use
- Can switch the processor from running one process to running another, by performing a context switch
  - Stop the current process' execution, *somehow*…
  - 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

A Simple Example

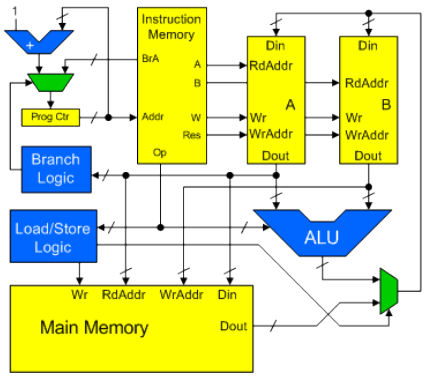
- Our computer can run several programs at a time
- Example: Four processes with four contexts:

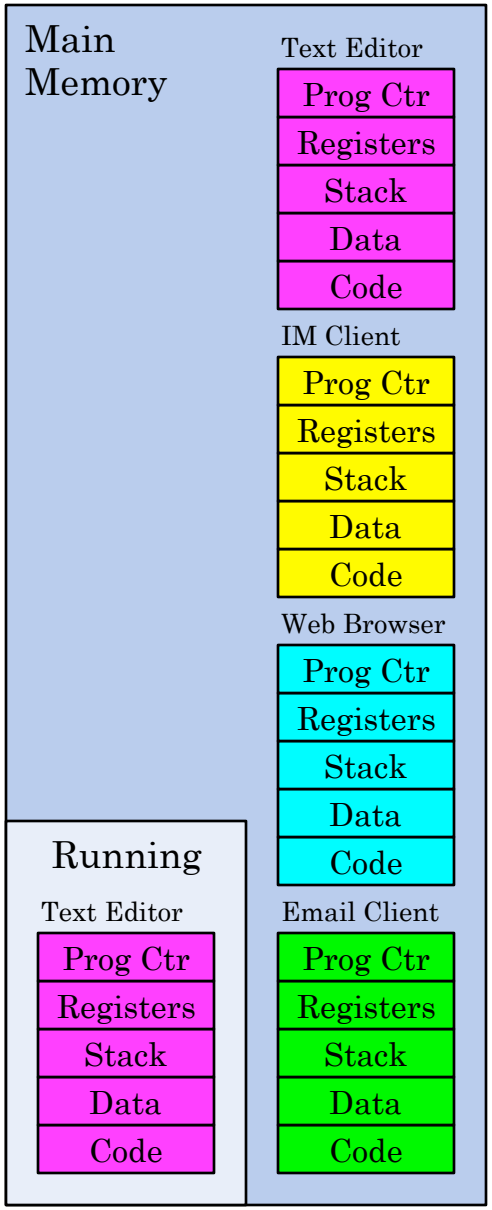


- None of these programs completely consume the processor
  - All must periodically wait for user, network, etc.

A Simple Example (2)

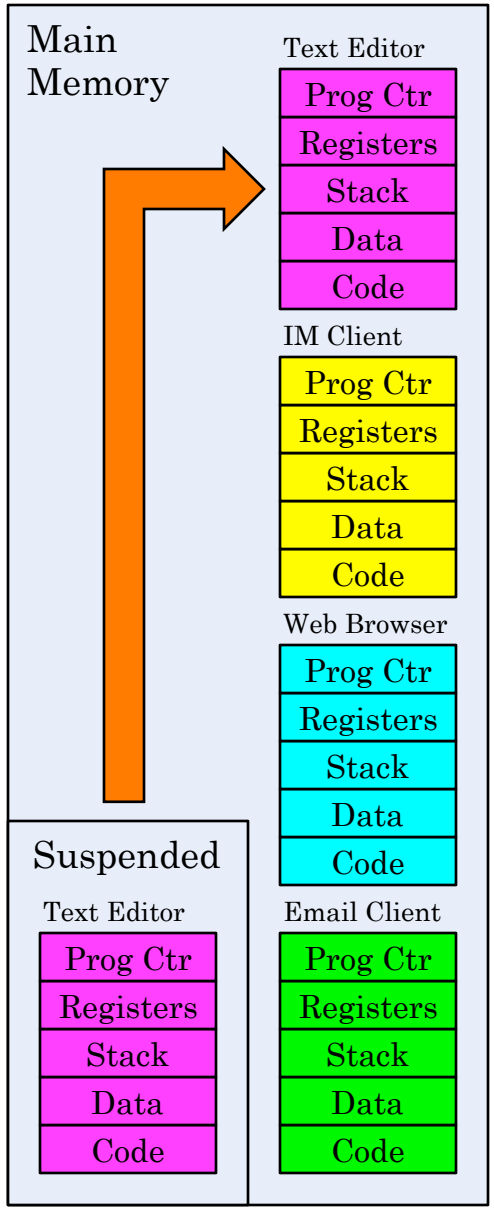
- Store context of each process in main memory
  - Need lots of memory, but oh well…
  - (We'll solve that problem later!)
- Only one process is currently running
  - Process has exclusive access to CPU
  - Process can only access its own data and code (what's inside the box)
  - Process doesn't have to worry about incompatibilities with how other processes are laid out in memory

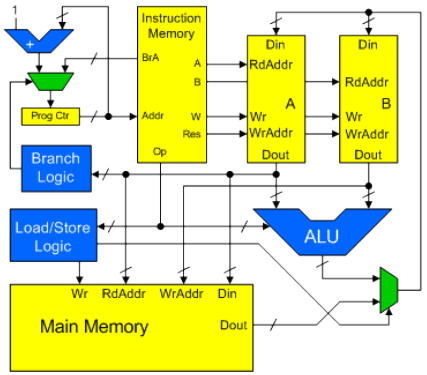




Context Switch

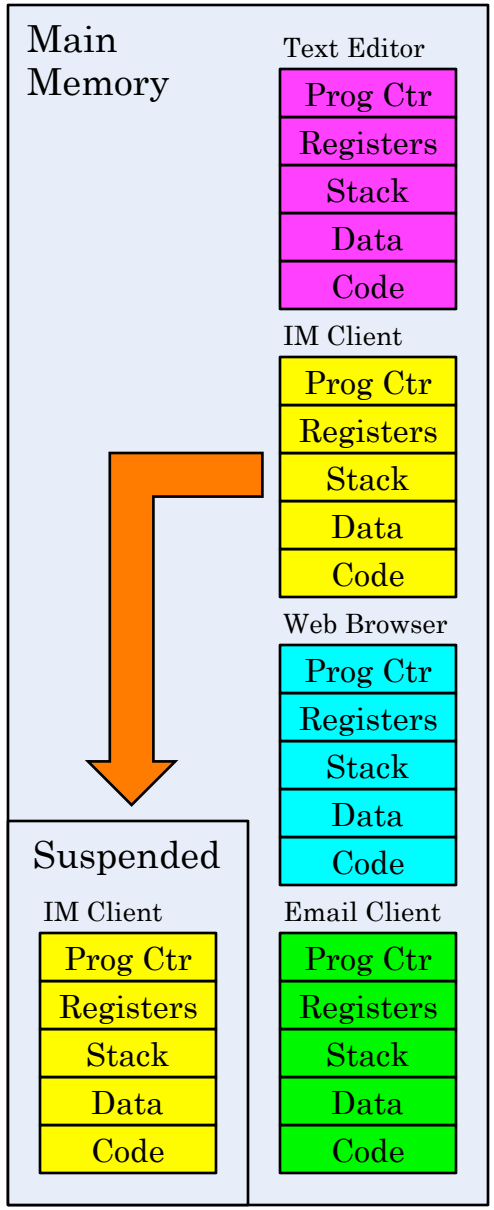
- At some point, preempt the current process, so another process can take its turn
  - Suspend the running process…
- Copy the process' context out of the "running process" area, back to the process' own context in main memory

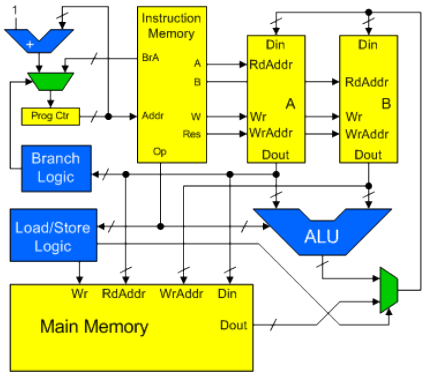




Context Switch (2)

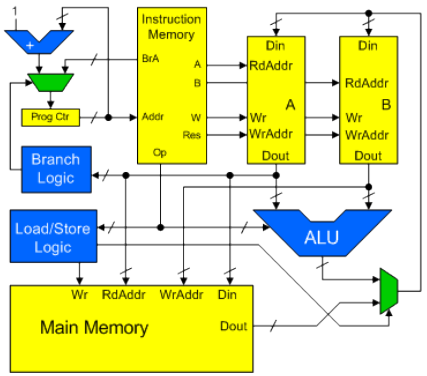
- Choose another process, and copy its state into the "running process" area
  - Copy all memory state (stack, heap, code, etc.) from context into the "running process" memory area
  - Also reload **`eip`**, **`eflags`**, **`esp`**, **`ebp`**, other registers from saved context into the processor's execution state

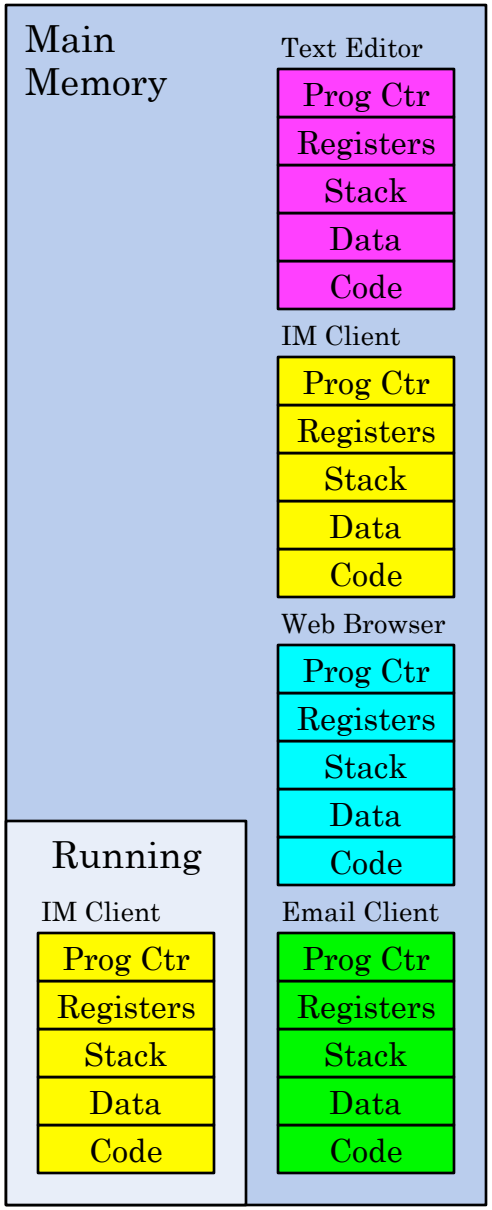




Context Switch (3)

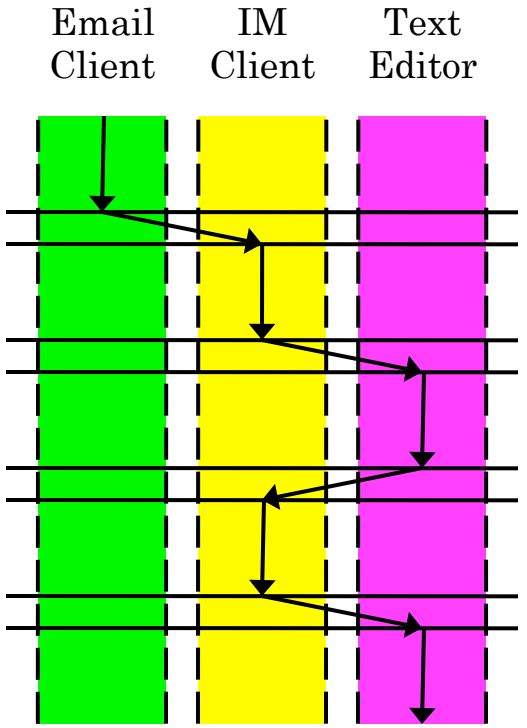
- Resume running the new process from where it previously left off
  - New process has no idea it was ever suspended…
  - Also isn't aware of any other program's state or internal memory layout





Physical and Logical Control Flow

- The physical processor is jumping back and forth between processes…
  - The *physical control flow* is jumping between the programs of multiple processes
- Within each process, execution proceeds as if it had exclusive access to the processor
  - The *logical control flow* of each process is solely through that process' code
- Concurrent processes have logical control flows that overlap



Next Time

- We have a good sketch of how we can virtualize the processor, but several big questions remain:
  - Who manages all the processes?
  - How do we ensure that processes can't see each other, but that the manager can see everything?
  - How do we interrupt a program while it's running?