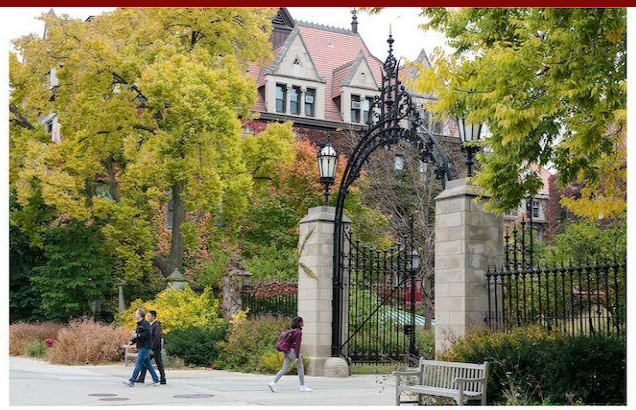
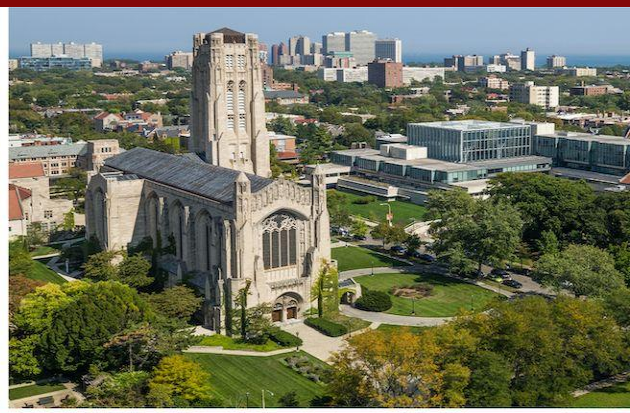
MPCS 52060 - Parallel Programming
M2: Shared-Memory Architecture

Lamont Samuels

THE UNIVERSITY OF CHICAGO | MASTERS PROGRAM IN COMPUTER SCIENCE

Shared Memory Systems

The focus of this course will be on implementing parallel programs on shared-memory systems

- A shared-memory system consists of at least one multi-core CPU that allows all processors to access memory as a single global address space (also known as global memory).
  - Many of the architectures designed by Intel, AMD, ARM, and etc.
  - Architectures in many modern personal computers and phone.
- Shared memory can be implemented as a set of a set of memory modules (e.g., caches, physical memory, etc.)

Program Execution on Shared Memory Systems

- The operating system (OS) along with the hardware is responsible for managing the execution of a program.
- An instance of a program running is known as a process and contains
  - Its own block of memory, where memory is made up of registers, caches, main memory, etc.
  - The program code translated into machine language (i.e., instructions for the processor)
  - Information about the state of the process (e.g., program counter)
  - Security Information
  - Descriptors of resources the OS has allocated to the process
  - A process cannot access the memory of another process.

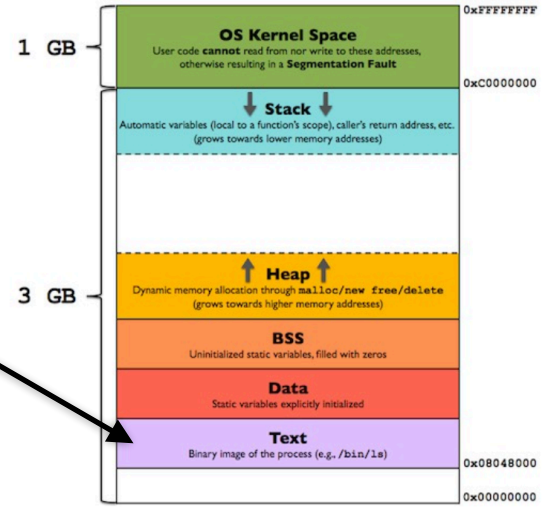
Source Code

```
int expr(int num)
{
    int value;
    value = (num + 1) * (num + 1) / 2;
    return value;
}
```

Assembly Code

```
_expr:
; %bb.0:
    sub  sp, sp, #16
    .cfi_def_cfa_offset 16
    str  w0, [sp, #12]
    ldr  w8, [sp, #12]
    add  w8, w8, #1
    ldr  w9, [sp, #12]
    add  w9, w9, #1
    mul  w8, w8, w9
    mov  w9, #2
    sdiv w8, w8, w9
    str  w8, [sp, #8]
    ldr  w0, [sp, #8]
    add  sp, sp, #16
    ret
```

The CPU executes each instruction one by one.



Cite: https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/

Processes and Processors

- The OS gives the illusion that a single processor system is running multiple programs simultaneously.
  - Each process takes turns running (i.e., time slice).
  - A processor can run a process for a while and then set it aside and run another process (i.e., context switch)
  - After its time is up, it waits (i.e., blocks) until it has a turn again.
- Processor may set aside or descheduled a process for a number of reasons:
  - A memory request that will take some time to satisfy
  - A process has run long enough (i.e., reached an end to its time slice.). Thus, it's time for another process to begin its time slice.
  - When a process is descheduled, it may resume execution on another processor.

Program Execution on Shared Memory Systems

- Each process when started will spawn a single thread, which is an entity within the process that can be scheduled for execution on a processor by the OS.
- A thread is responsible for executing the instructions of the task assigned to it.
  - Reminder a task is a unit of work (i.e., a series of instructions from the program)
  - In a sequential program there is only one thread, known as the main (also known as primary, and heavy) thread and is executing all the code in the program.
- A concurrent program can fork (also known as spawn) additional "light-weight" threads within a given process to execute tasks.
  - All threads in the process can then be executed in parallel on the various processors.

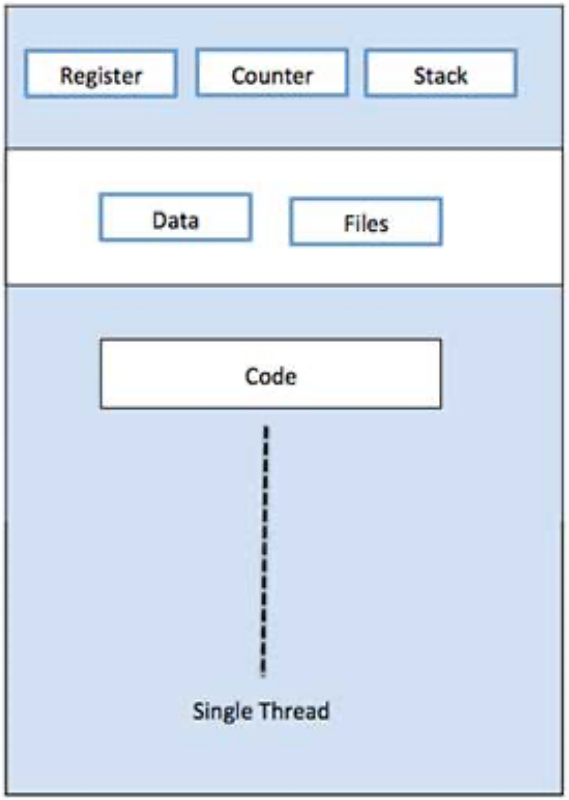
Threads

- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.

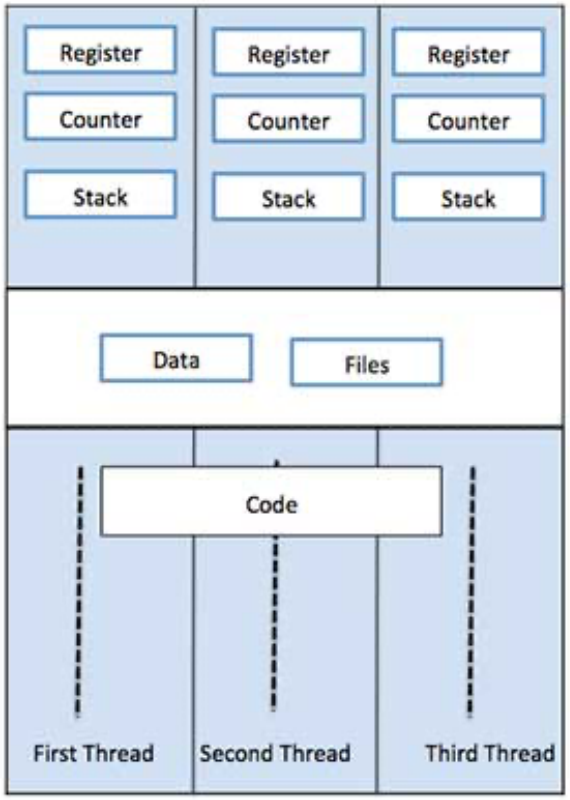
- Each have their own stack where they can store local variables, function calls, etc but share all other memory components (e.g. data and code segments) with other threads of the same process.

- Some languages and operating systems provide the notion of thread-local storage, where threads can store and retrieve values independent of other threads.

Thread vs Process



Single Process P with single thread



Single Process P with three threads

Diagram Cite: https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Aside: Simultaneous Multithreading (aka HyperThreading)

- Most modern multicore architectures have Simultaneous Multithreading(SMT):
  - Use several threads to schedule instructions from different threads in the same cycle if necessary.
  - It helps increase the usage of functional units of a processor more effectively.
  - Hardware support for SMT is based on the replication of the chip area used to store the processor state.

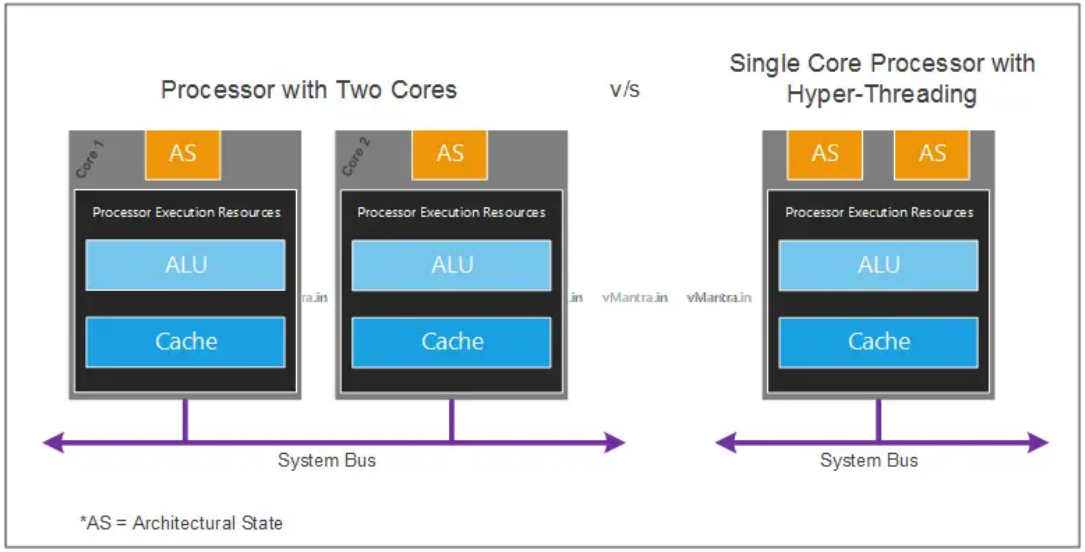


Diagram Cite: https://www.vmantra.in/hyper-threading/

Aside: Simultaneous Multithreading (aka HyperThreading)

- Processors appear to the operating system and user programs as a set of logical processors to which threads can be assigned for execution.
- Threads can come from a single or several user programs.
- Number of replications of the processor state determines the number of logical processors.
- These logical processors are also known as hardware threads.

Concurrent Programming on Shared Memory Systems

Aside: Understanding Pointers

- Before we start talking about concurrent programming in Go, you need to understand the notion of a pointer.
  - A pointer is just an object that stores a memory address.
  - It will be important to understand because this will be the way we allow “threads” in Go to communicate initially when writing parallel programs.
- Demo: m2/pointers directory

Structuring Parallel Programs

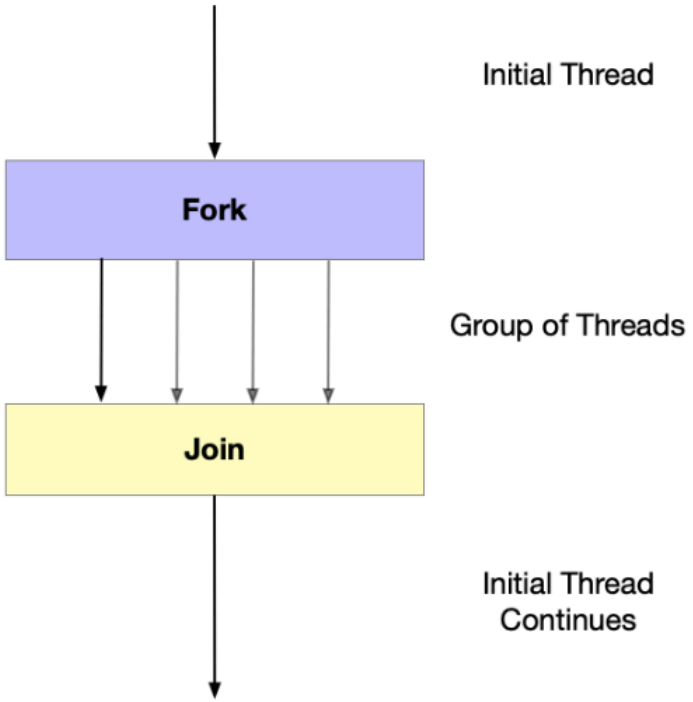
- There are many factors to consider when structuring and designing parallel programs (which we will learn throughout this course) but to start here are a few important ones to consider
  - Task Decomposition - how to divide up a problem into subproblems (i.e., tasks) to be executed concurrently.
  - Distribution of tasks - how to structure parallel programs to efficiently assign tasks to threads and how this affects the scheduling of threads by the OS.
    - We will take a look at various ways to structure parallel programs based on how tasks are generated via an algorithm.
  - Synchronization - how tasks/threads communicate and coordinate in order to obtain deterministic results
  - Data dependencies - where one task's output is required to be the input for another task.
  - Scalability - as the program as given more compute resources (e.g. cores) its performance also increases.

Task Decomposition and Granularity

- When thinking about breaking a problem down into tasks you need to consider the granularity (or grain size) of a task
  - Granularity - a measure of the amount of work (or computation) which is performed by that task
  - Also can be thought as qualitative measure of the ratio of computation (time to complete the task) to communication (time needed to exchange data between processors for a task).
  - We will talk more about granularity later in the course.
- For now, it's best on shared memory systems to have many small tasks that are evenly distributed across all processors.

Fork-join Pattern

- New parallel control flows are created by spawning (i.e., also known as forking) new threads and are terminated at a join, where the parallel control flows meet and a single flow continues onwards.
  - Basis for many of the parallel patterns we will see throughout this class.
  - Each spawned thread can then work on one or more tasks.



Basic Fork-join Pattern

- Generic pseudocode

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve(part)
        join all subtasks spawned in previous loop
        return combined results
```

Pseudocode Source: Doug Lea (2000). A Java fork/join framework. ACM Conference on Java.

Forking in Go

- In Go, each concurrently executing activity/task is called a goroutine:
  - A goroutine is similar to a thread in other languages/operating systems but it's not actually a thread.
  - At program startup, the only goroutine is the one that calls the main function, which is known as the main goroutine.
- New goroutines are created by the go statement:
  - Is an ordinary function/method call prefixed by the keyword go.
  - Causes the function being called to execute concurrently by a goroutine
  - The goroutine forking the new goroutine returns immediately after the go statement

```
f() // call f(); wait for it to return
go f() // create a new goroutine that falls f(); don't wait
```

Fork-join Pattern

- In Go, using a Waitgroup defined in the sync package allows us to implement the fork-join pattern
  - Methods:

```
type WaitGroup
func (wg *WaitGroup) Add(delta int)
func (wg *WaitGroup) Done()
func (wg *WaitGroup) Wait() // join point
```

- As stated in the documentation
  - *"A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished."*

Demo: Fork-join in Go
m2/accounts0

Race Conditions

- Shared-memory systems use shared variables/resources (i.e., memory locations) which can be accessed by all processors.
- Communication and cooperation between the processors is organized by writing and reading shared variables that are stored in memory.
- A race condition is non-deterministic behavior in a parallel program when the result of an operation depends on the interleaving of certain individual operations.
  - Code with a race condition can run deterministically sometimes and fail other times.
  - Hard to reproduce and diagnose because they can be appear in frequently.
  - Only present sometimes under heavy load or when using certain compilers, platforms, architectures
  - A specific type of race condition is a data race, which is a situation when at least two threads access a shared variable at the same time. One thread must try to modify the variable.
    - This is normally due to not using proper synchronization when accessing the shared variable.
    - Races can also happen with files and I/O (e.g., printing to the screen) too.

Race Conditions and Critical Sections

- Examples of race conditions

Thread 1	Thread 2
a := x	b := x
a += 1	b += 1
x = a	x = b

Data race to update the value of x

Thread 1	Thread 2
x = 1	y = 1
a = y	b = x

Race not explained for serial interleaving
Assume x and y are initially zero.

- To ensure determinism and to avoid race conditions, you need to determine critical sections in your code:
  - A critical section is a block of code where potentially more than one thread can access/modify a shared resource at the same time that could result in a race condition.
  - The execution of code in a critical section should, effectively, be executed as serial code.
  - Eventually, another thread should be able to access this section once one thread has completed the critical section.

Synchronization using Atomics

- Synchronization is needed when dependencies exist between parallel tasks and/or to handle race conditions
- Many of the low-level synchronization primitives (e.g., locks, monitors, etc.) are built off of specialized hardware primitives/instructions (also known as atomic operations):
  - On a shared memory system, an operation is consider atomic if it completes in a single step relative to other threads.
  - No other thread can observe the modification to that shared variable half-way through its operation.
  - Provided in Go sync package: import “sync/atomic"

Demo: Fork-join in Go with correct synchronization
m2/accounts1

Problems with Atomic Operations

- Best practices is to use atomic operations sparingly because:
  - Most atomic operations are implemented using CAS or (LL/SC), which take significant more cycles to complete than a simple load or store instruction.
  - Causes a memory fence, which forces the write-back buffer to be sent to main memory. This process can then stall other processors from reading/ writing to main memory.
  - Prevents out-of-order execution and various compiler optimizations.
  - Cost to performance varies depending on architectures, program design, etc.
  - Adds more hardware complexity.
- **In general, best practice for all synchronization is to use them only at that point where it is necessary!**
  - Too little synchronization leads to non-deterministic behavior.
  - Too much synchronization can limit scaling and/or lead to deadlock (next week).

Instruction Level Parallelism in Hardware

Processor Cycle

- For multicore architectures, the basic unit of time is a cycle:
  - The time it takes a processor to fetch and execute a single instruction
  - As technology advances, cycle times change
    - 1980: 10 million cycles/sec
    - 2005: A 3 GHz processor does 3 billion cycles/sec
    - Some instructions take one cycle, and some may take hundreds.

Instruction Level Parallelism

- Modern hardware architectures have a multi-core architecture, where both serial and parallel programs have benefited from instruction level parallelism(ILP):
  - Simultaneous execution of a sequence of instructions
- There exists various techniques to exploit ILP (here are a few)
  - Instruction Pipelining - Uses functional units like ALUS (arithmetic logical units), FPUs(floating point unit), load/store units, or branch units to execute independent instructions in parallel by different functional units.
  - Out-of-order execution - instructions execute in any order that does not violate data dependencies
  - Speculative execution & Branch prediction - Processors can execute instructions speculatively before branches or data have been computed.
- Processors that allows this instruction level parallelism are known as superscalar processors.

All of these techniques discussed are handled by compilers and/or hardware, which make sequential and parallel programs more efficient and have better performance.

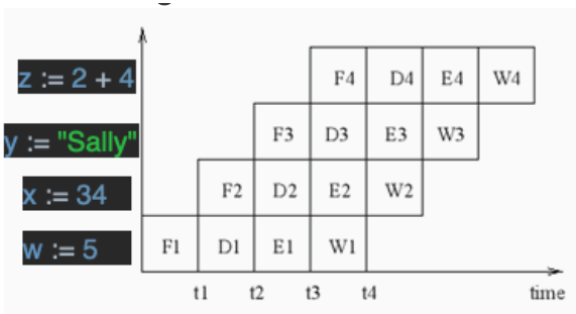
Instruction Pipelining

Simple example of pipeline stages:

- fetch: Retrieve the next instruction to be executed from memory
- decode: decode the instruction fetched in step (1).
- execute: load the operands specified and execute the instruction
- write back: write the result into the target location.

```
func main() {

    w := 5
    x := 34
    y := "Sally"
    z := 2 + 4
    //...
}
```



- Main benefit: Different pipeline stages can operate in parallel, if there are no control or data dependencies between the instructions to be executed.

Improving Data Access Performance

Processors & Memory

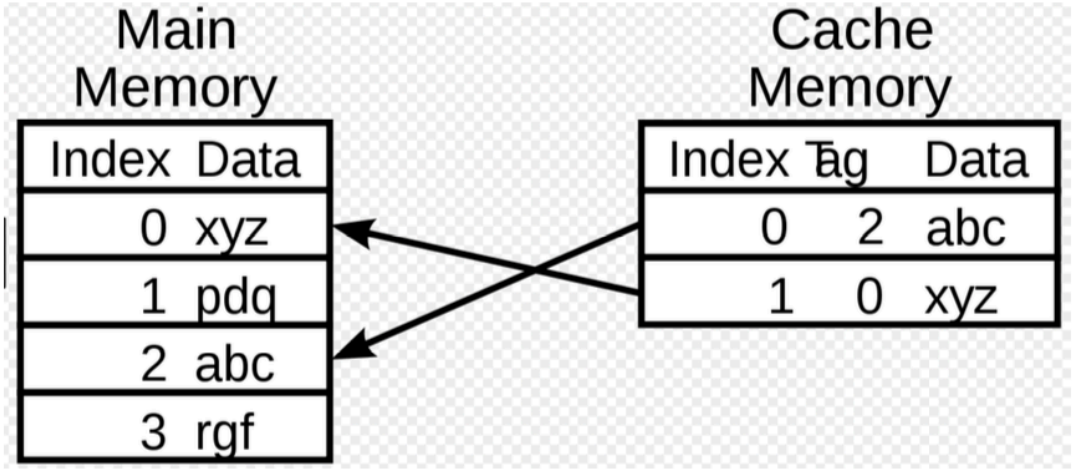
- On architectural principle drives everything else: processors and main memory are far apart.
  - Takes a long time to read a value from memory
  - Takes a long time for a processor to write a value to memory
  - Takes a longer time for the processor to verify that the value written is installed in memory.
  - The relative cost of instructions such as memory access changes slowly when expressed in terms of cycles.
  - **Analogy**: Accessing memory is more like mailing a letter than making a phone call.
- Memory access time has a large influence on program performance. The objective of architecture trends over the years have been to reduce memory access latency:
  - The total time that elapses until a memory access operation has been completely terminated.

Processors & Memory

- To help alleviate these memory latency issues, memory inside modern computers is actually a hierarchy of components that store data.
- Ranges from very few registers, to one or more levels of small, fast caches to relatively slow main memory.
- **Understanding how these levels interact is essential to understanding the actual performance of many concurrent algorithms.**

Caches

- On modern architectures, processors can waste hundreds of CPU cycles waiting to access main memory.
- We can alleviate problem by using caches:
  - A collection of memory locations that can be accessed in less time than some other memory locations.
  - A cache is typically located on the same chip as the processors
  - Cache line: fixed-size block of data that also contains metadata (e..g, tag, index)
  - Cache lines are normally 64 or 128 bytes



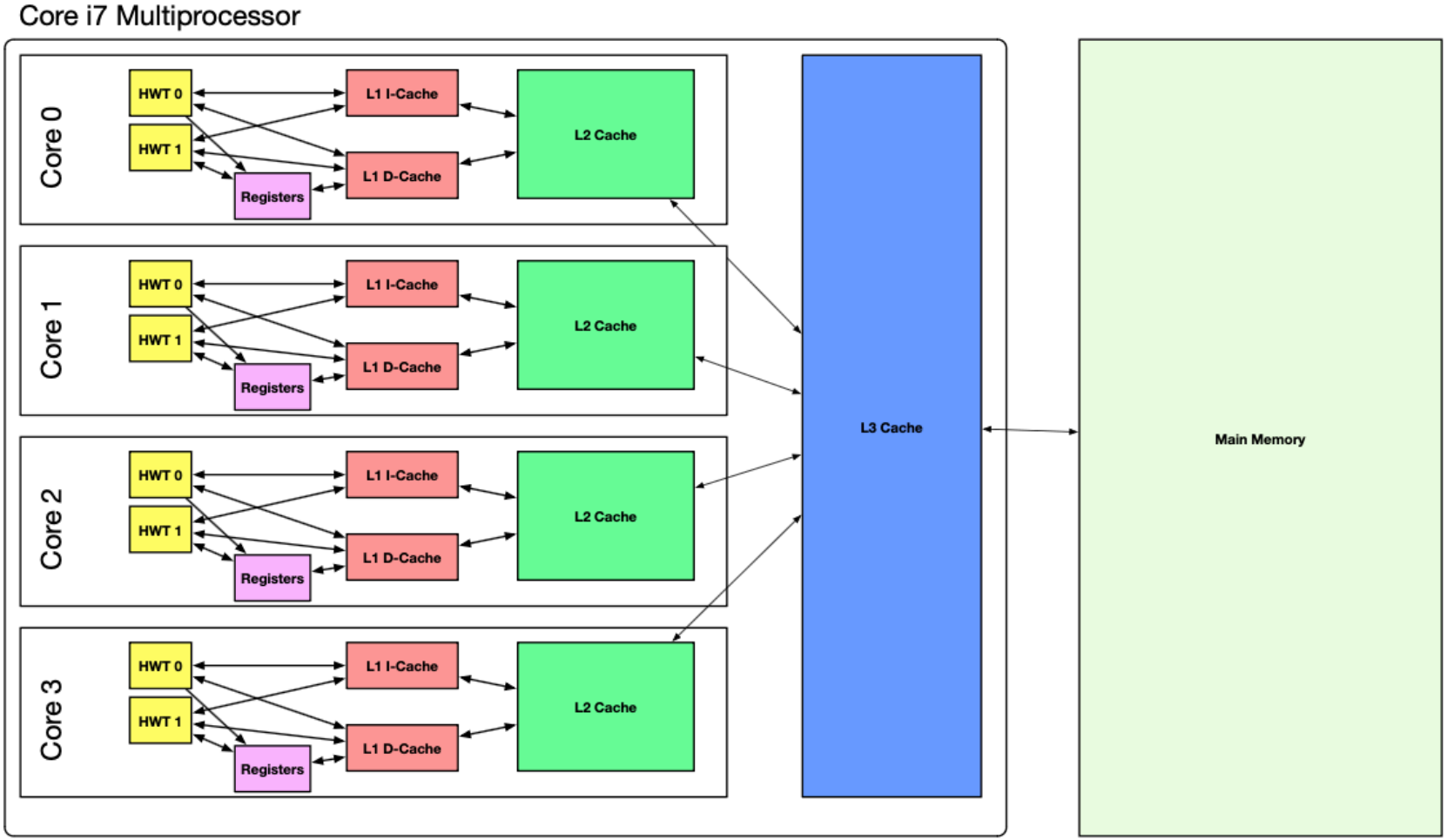
Principle of locality

- Caches are effective because most programs display a high degree of locality:
  - Accessing one location is followed by an access of a nearby location.
  - Types of locality
    - Spatial locality – accessing a nearby location.
    - Temporal locality – accessing in the near future.

Cache Levels

- Most processors have two levels of caches, called L1, L2, and L3:
  - L1 - typically resides on the same chip as the processor and takes 1-2 cycles to access
  - L2 - may reside either on or off-ship, and takes about 10 cycles to access
  - L3 - normally is off chip but is accessible within about 30 cycles
- Note: These times vary from platform to platform. All these are significantly faster to access than main memory, which can be 100s of cycles.

Core i7-9xx Cache Hierarchy



L1 i-cache and d-cache: 32KB, 8-way, Access: 4 cycles

L2 cache: 256KB, 8-way, Access: 11 cycles

L3 cache: 8MB, 16-way, Access: 30-40 cycles

Cache-Line (Block) size: 64 bytes
(All caches)

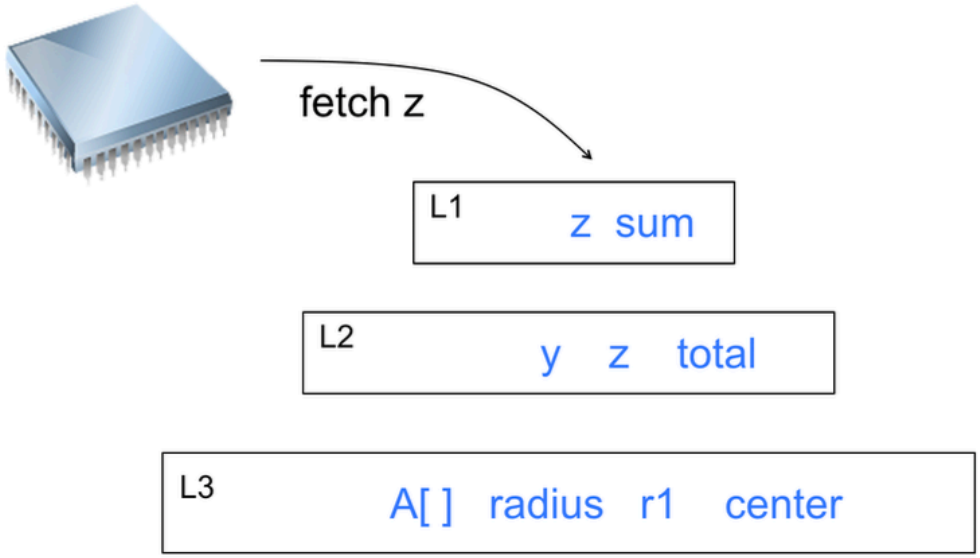
Principle of locality

- What could the caches levels look like for this program?

```
package main
import "fmt"
func main() {
  var z [1000]int
  var sum int
  for i := 0; i < 1000; i++ {
    sum += z[i]
  }
  fmt.Printf("%v\n", sum)
}
```

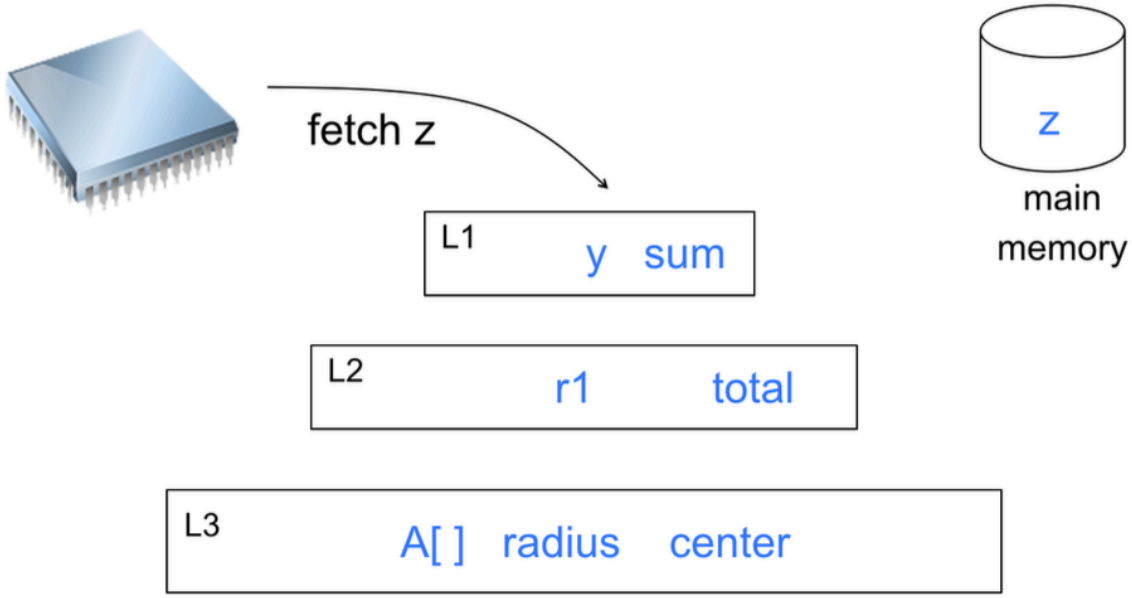
Cache hit

- When a processor attempts to read a value from a given memory address, it first checks the cache(s).
- We call it a cache hit if the value for the processor is located in one of the cache levels.



Cache Miss

- Otherwise, a cache miss is when a value is not in the cache and the process is required to go to main memory.



When a Cache Becomes Full...

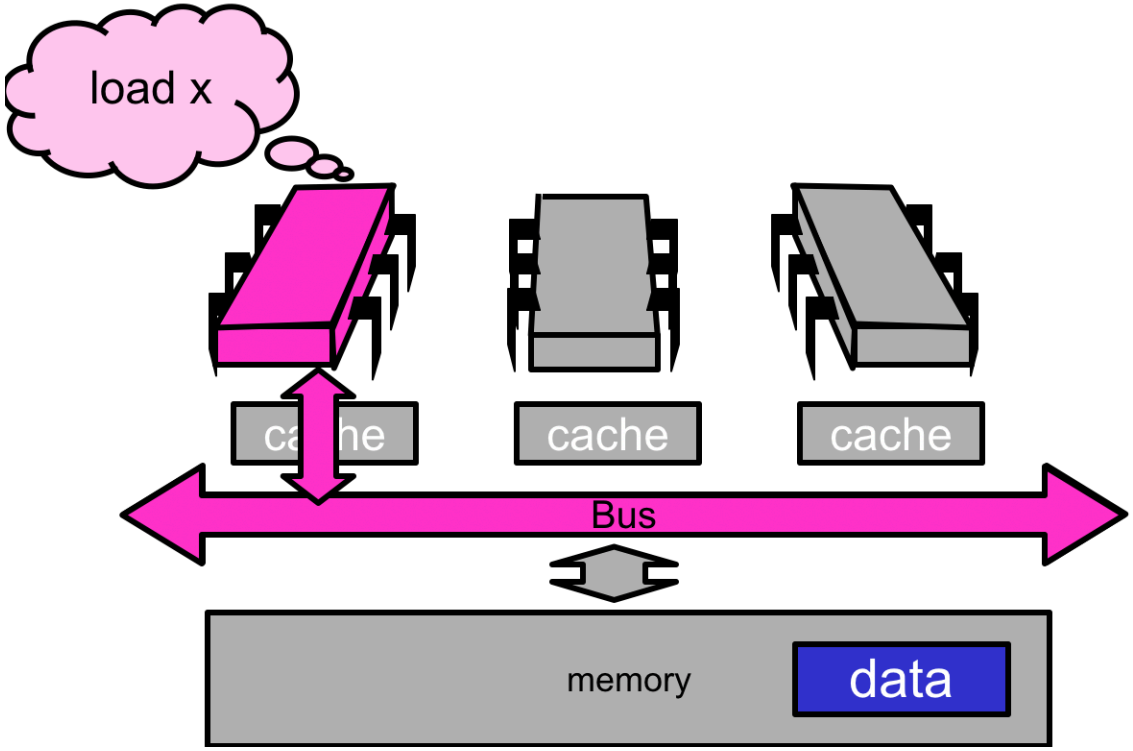
- Caches are expensive to build and therefore significantly smaller than main memory.
- Need to make room for new entry when the cache is full by evicting an existing entry:
  - Discarding a entry if it has not been modified
  - Writing it back to main memory if it has been modified
- Need a replacement policy (determines which cache line to replace to make room for a new location).
  - Usually some kind of least recently used heuristic.

Cache Coherence

- Processor **A** and Processor **B** both cache an address x
- Processor **A** writes to x
- This operation updates the cache
- How does Processor **B** find out about the update?
- A cache coherence protocols provides a specification on how to keep caches in sync with each other. Many cache coherence protocols in the literature.

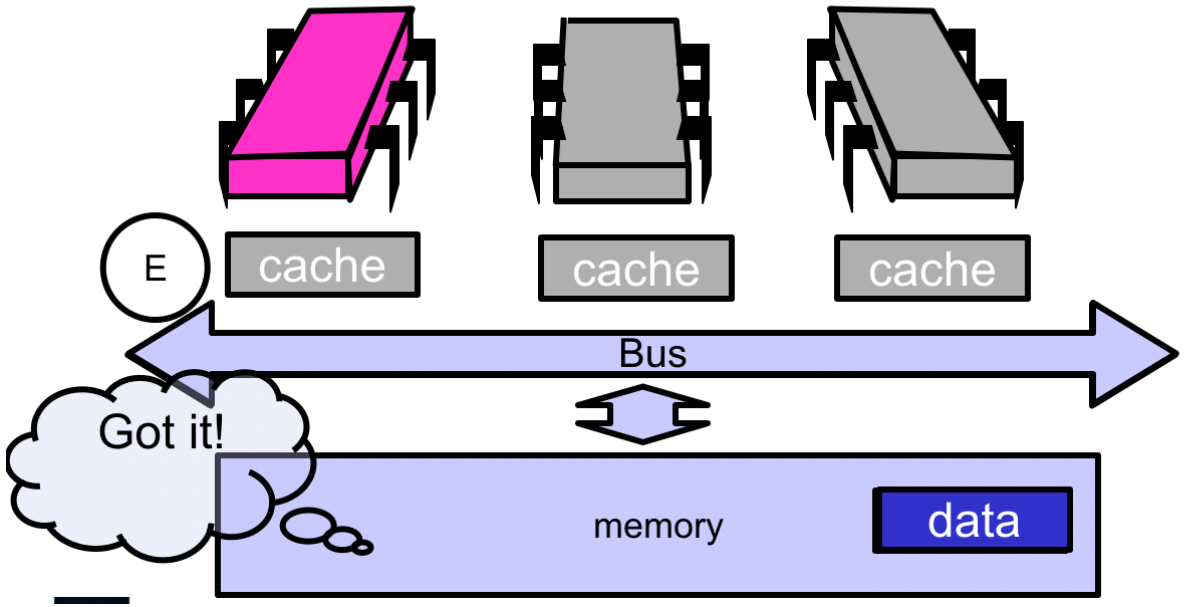
MESI Example

- A processor issues load request at address x from main memory.



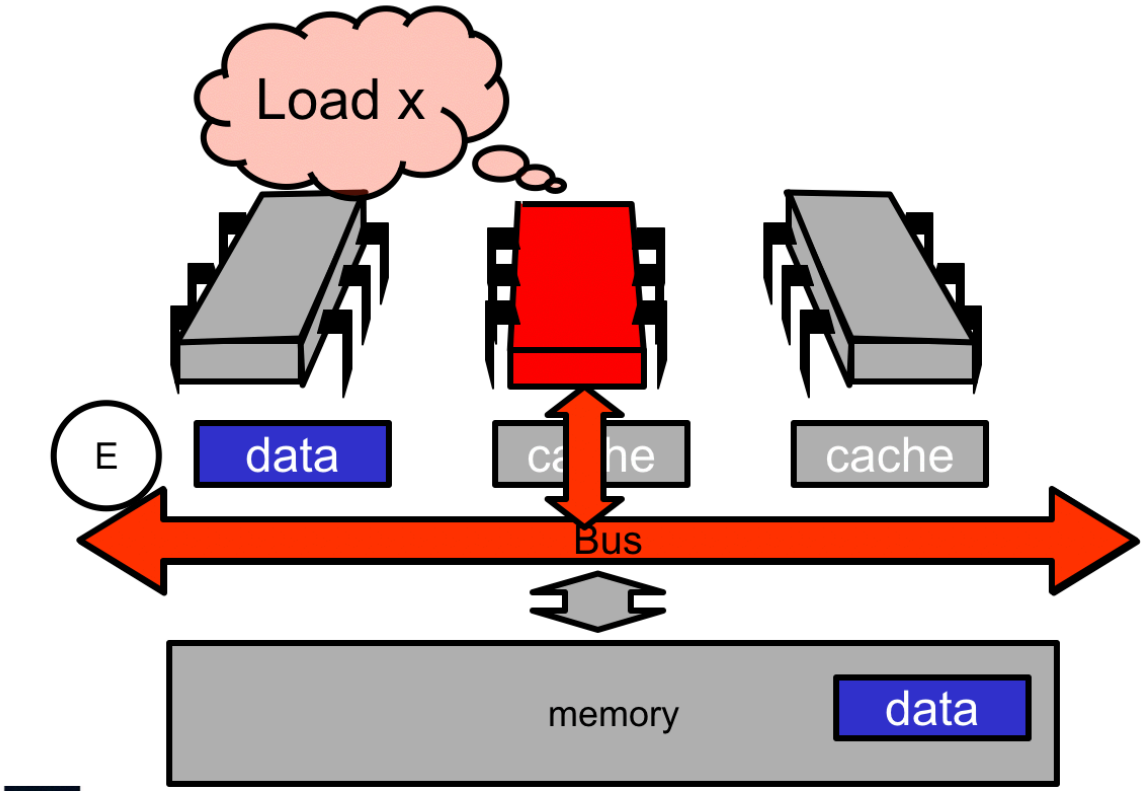
MESI Example

- Main memory responds with the requested data and the processor places the data in a cache line. The data is exclusive to the cache of the requesting processor.



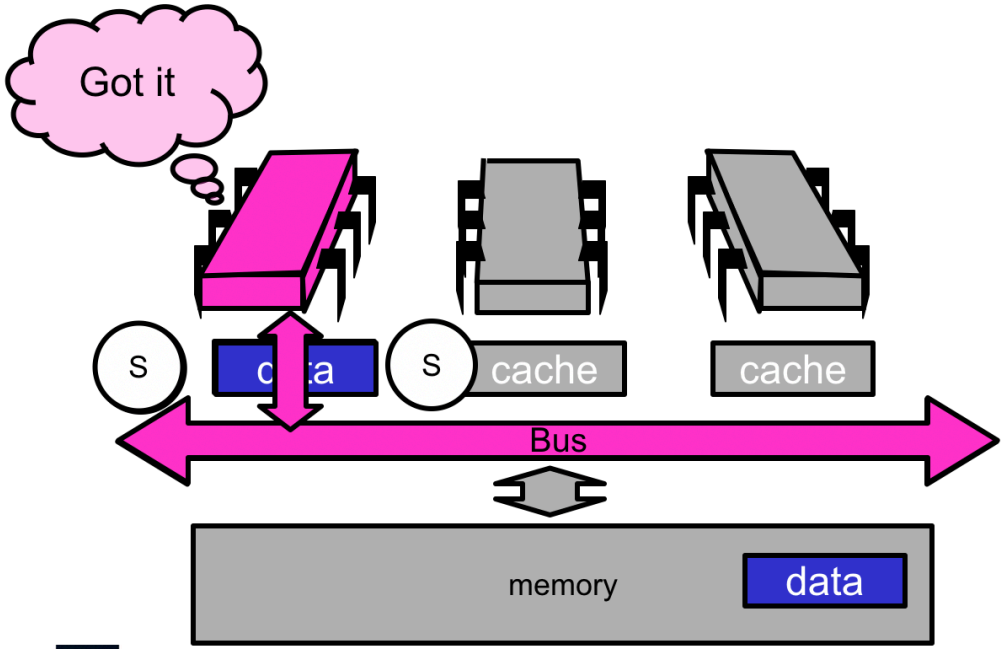
MESI Example

- Another processor requests the data from the same address x from main memory.



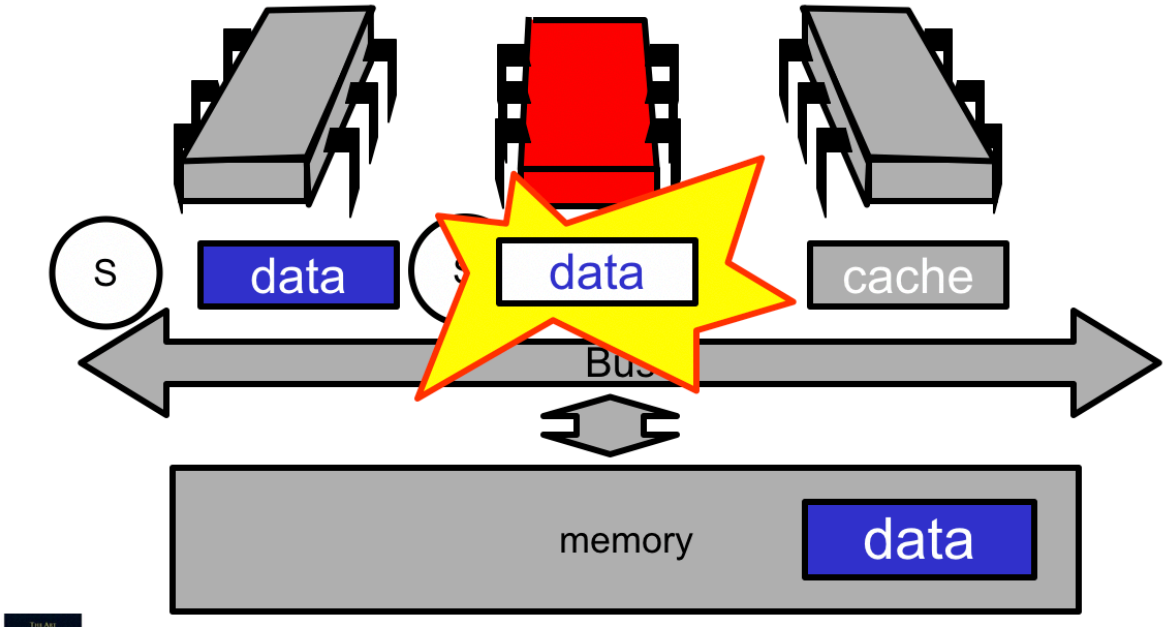
MESI Example

- Processors communicate with each other to update their states to shared since both contain data from the same memory address.



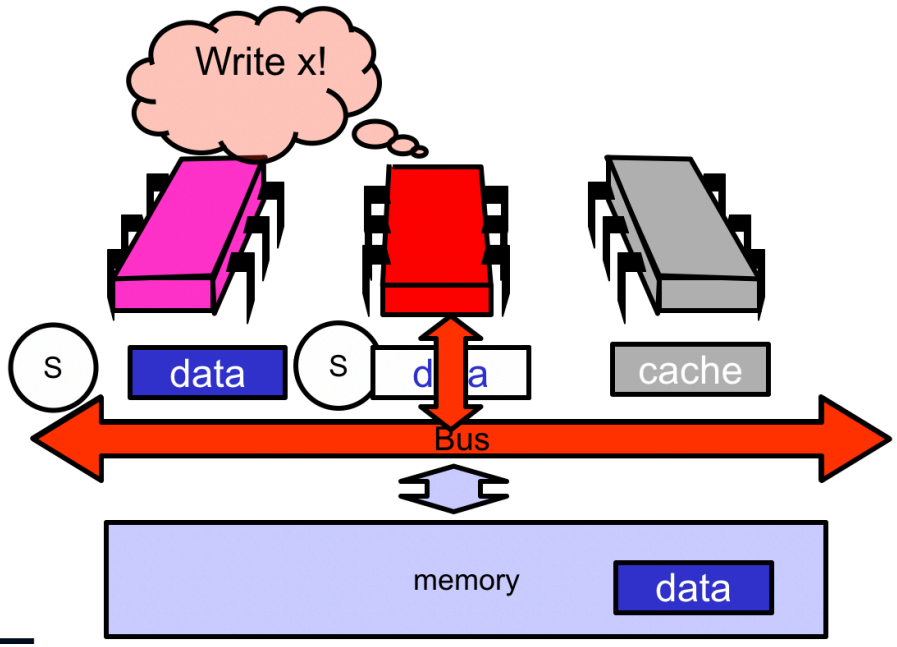
MESI Example

- A processor modifies the data for that memory address in its cache. Other processors need to be notified about this update.



MESI Example

- All other processors are given the updated data and depending on the write scheme for the cache the data is either immediately sent to main memory or held for a period of time before being sent to it.



Write-Through Caches

- Immediately broadcast changes
- Good
  - Memory, caches always agree
  - More read hits, maybe
- Bad
  - Bus traffic on all writes
  - Most writes to unshared data
    - For example, loop indices ...
- Hardly used in practice due to the bus traffic problem

Write-Back Caches

- Immediately broadcast changes
- Caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.
- The dirty data is usually held in a write-buffer that will eventually write its contents back to main memory periodically.

MESI - Invalidate