

CSC 252: Computer Organization

Spring 2026: Lecture 24

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Another Deadlock Example: Signal Handling

- Signal handlers are **concurrent with main program** and may **share the same global data structures**.

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    Signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    if (x == 5)
        y = x * 2; // You'd expect y == 10
    exit(0);
}
```

What if the following happens:

- Parent process executes and finishes `if (x == 5)`
- OS decides to take the SIGCHLD interrupt and executes the handler
- When return to parent process, **y == 20!**

Fixing the Signal Handling Bug

```
static int x = 5;
void handler(int sig)
{
    x = 10;
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    sigfillset(&mask_all);
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);

    exit(0);
}
```

- Block all signals before accessing a shared, global data structure.

How About Using a Mutex?

```
static int x = 5;
void handler(int sig)
{
    P(&mutex);
    x = 10;
    V(&mutex);
}

int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    signal(SIGCHLD, handler);

    if ((pid = Fork()) == 0) { /* Child */
        Execve("/bin/date", argv, NULL);
    }

    P(&mutex);
    if (x == 5)
        y = x * 2; // You'd expect y == 10
    V(&mutex);

    exit(0);
}
```

- This implementation will get into a deadlock.
- Signal handler wants the mutex, which is acquired by the main program.
- **Key:** signal handler is in the same process/thread as the main program. The kernel forces the handler to finish before returning to the main program.

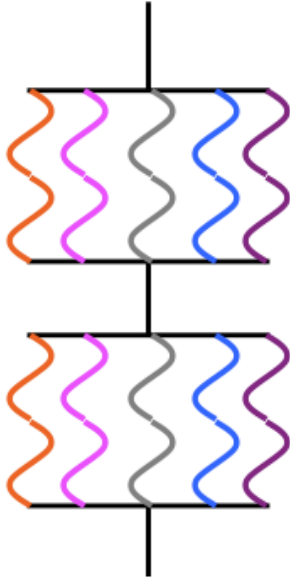
Summary of Multi-threading Programming

- Concurrent/parallel threads access shared variables
- Need to protect concurrent accesses to guarantee correctness
- Semaphores (e.g., mutex) provide a simple solution
- Can lead to deadlock if not careful
- Take CSC 254/258 to know more about avoiding deadlocks (and parallel programming in general)

Thread-level Parallelism (TLP)

- Thread-Level Parallelism
 - Splitting a task into independent sub-tasks
 - Each thread is responsible for a sub-task
- Example: Parallel summation of N number
 - Partition values $1, \dots, n-1$ into t ranges, $\lfloor n/t \rfloor$ values each range
 - Each of t threads processes one range (sub-task)
 - Sum all sub-sums in the end
- Question: if you parallel you work N ways, do you always an N times speedup?

Why the Sequential Bottleneck?



- Maximum speedup limited by the sequential portion
- Main cause: **Non-parallelizable operations on data**
- Parallel portion is usually not perfectly parallel as well
 - e.g., Synchronization overhead

Each thread:

```
loop {
```

```
  Compute
```

```
  N
```

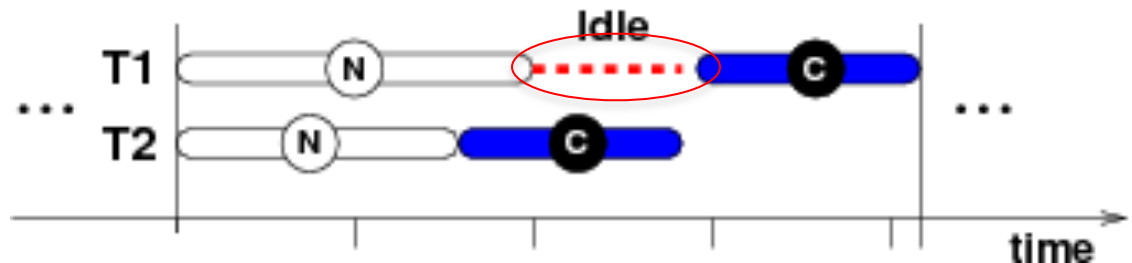
```
  P(A)
```

```
    Update shared data
```

```
  V(A)
```

```
  C
```

```
}
```



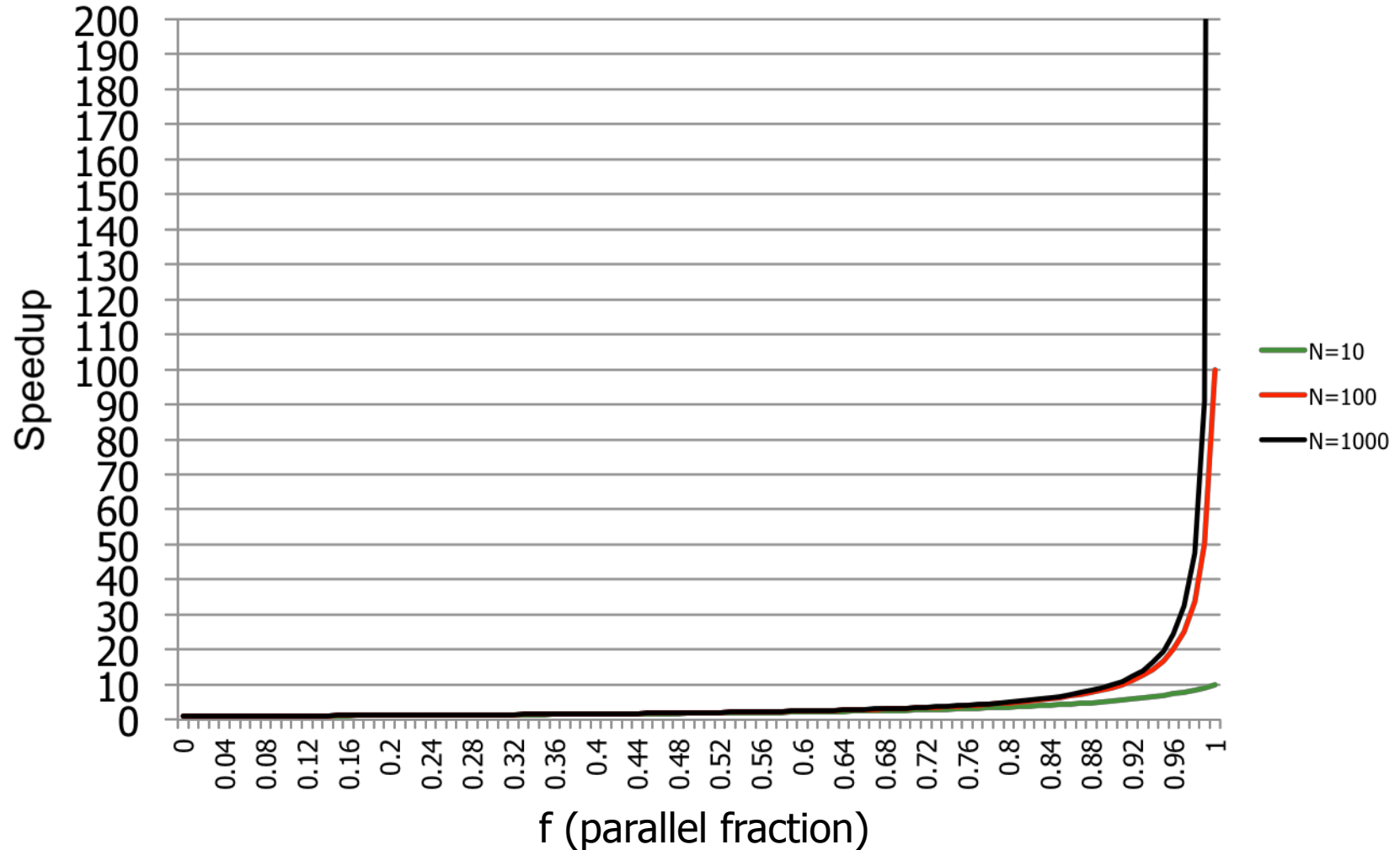
Amdahl's Law

- Gene Amdahl (1922 – 2015). Giant in computer architecture
- Captures the difficulty of using parallelism to speed things up
- Amdahl's Law
 - f : Parallelizable fraction of a program
 - N : Number of processors (i.e., maximal achievable speedup)

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}$$

- Completely parallelizable ($f = 1$): Speedup = N
- Completely sequential ($f = 0$): Speedup = 1
- Mostly parallelizable ($f = 0.9$, **$N = 1000$**): **Speedup = 9.9**

Sequential Bottleneck



Today

- From process to threads
 - Basic thread execution model
- Multi-threading programming
- **Hardware support of threads**
 - Single core
 - Multi-core
 - Cache coherence

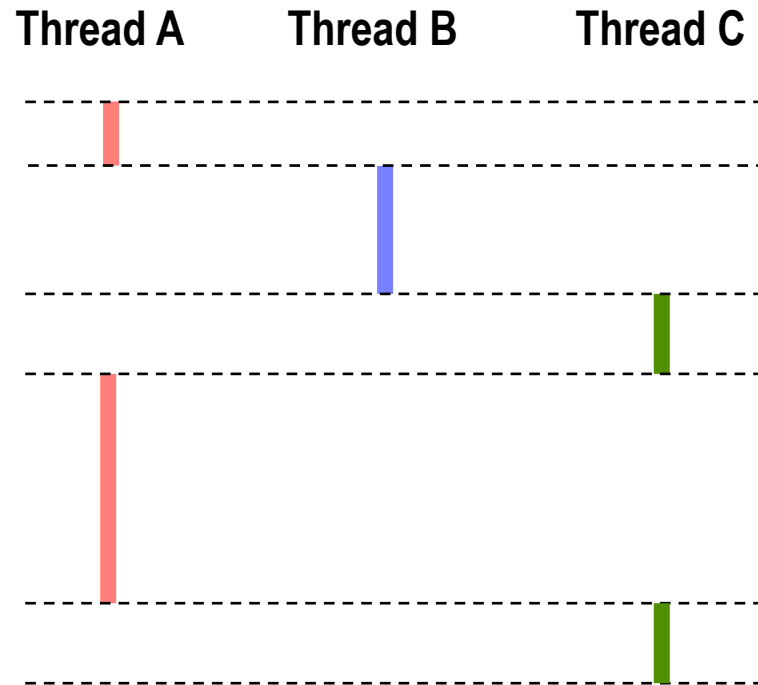
Can A Single Core Support Multi-threading?

- Need to multiplex between different threads (time slicing)

Sequential

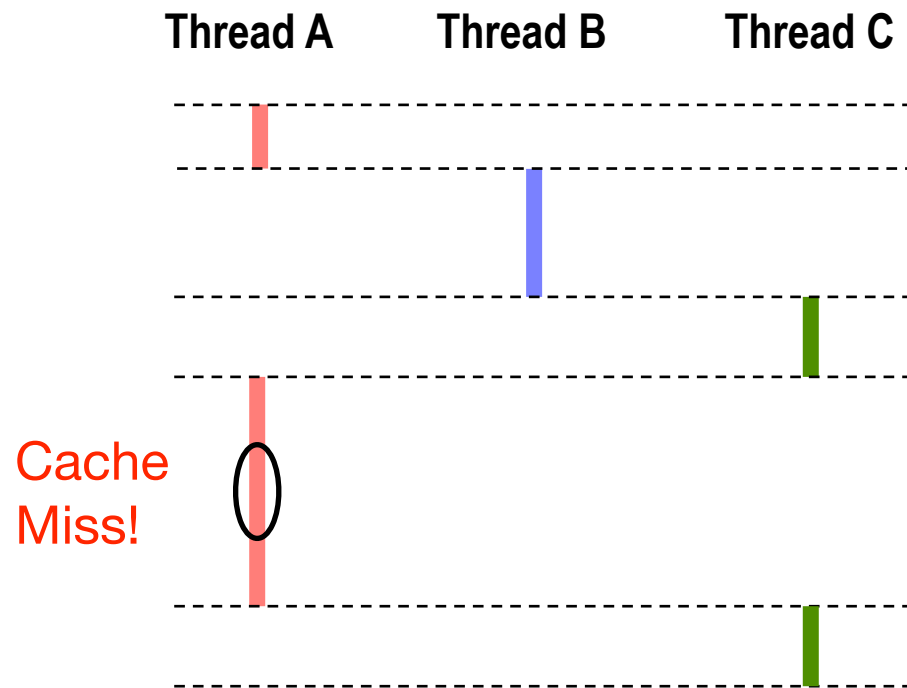


Multi-threaded



Any benefits?

- Can single-core multi-threading provide any performance gains?
- If Thread A has a cache miss and the pipeline gets stalled, switch to Thread C. Improves the overall performance.



When to Switch?

- Coarse grained
 - Event based, e.g., switch on L3 cache miss
 - Quantum based (every thousands of cycles)
- Fine grained
 - Cycle by cycle
 - Thornton, “[CDC 6600: Design of a Computer](#),” 1970.
 - Burton Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978. The HEP machine. A seminal paper that shows that using multi-threading can avoid branch prediction.
- Either way, need to save/restore thread context upon switching.

Fine-Grained Switching

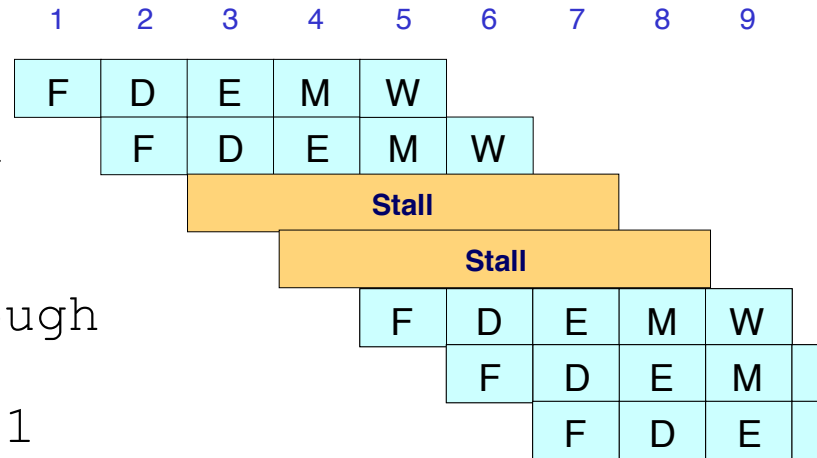
- One big bonus of fine-grained switching: no need for branch predictor!!

The stalling approach

```

    xorg %rax, %rax
    jne L1          # Not taken
Stall
Stall
    irmovq $1, %rax # Fall Through
L1  irmovq $4, %rcx  # Target
    irmovq $3, %rax # Target + 1

```



Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!

The branch prediction approach

demo-j.ys

0x000: xorq %rax,%rax

0x002: jne target # Not taken

0x016: **irmovq \$2,%rdx** # Target

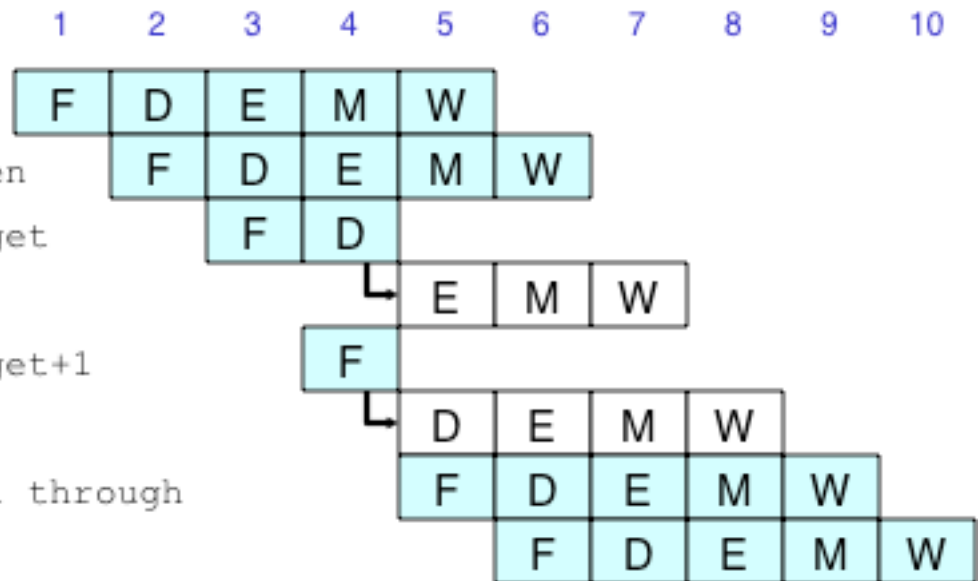
bubble

0x020: irmovq \$3,%rbx # Target+1

bubble

0x00b: **irmovq \$1,%rax** # Fall through

0x015: halt

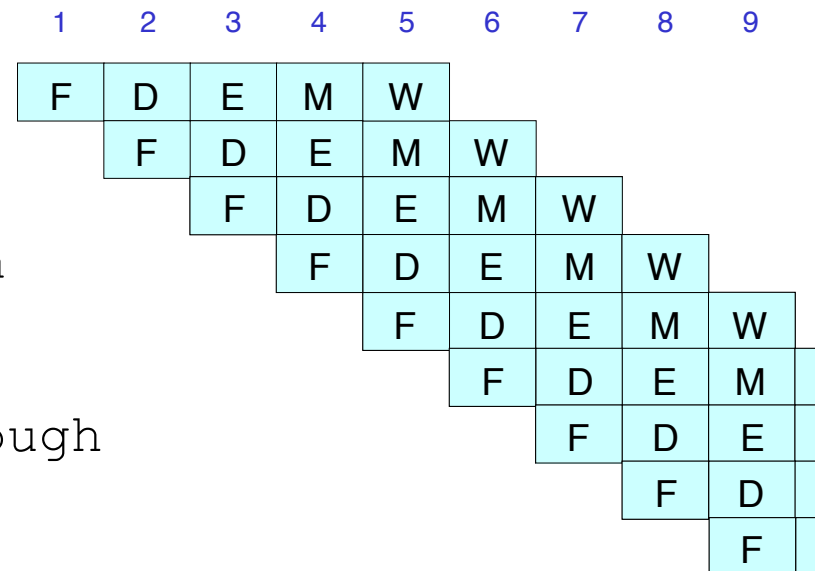


Fine-Grained Switching

- One big bonus of fine-grained switching: no need for branch predictor!!
 - Context switching overhead would be very high! Use separate hardware contexts for each thread (e.g., separate register files).
 - GPUs do this (among other things). More later.

The fine-grained multi-threading approach

```
xorg %rax, %rax
Inst x from TID=1
Inst y from TID=2
jne L1          # Not taken
Inst x+1 from TID=1
Inst y+1 from TID=2
irmovq $1, %rax # Fall Through
Inst x+2 from TID=1
Inst y+2 from TID=2
... ..
```



Thread Switching

- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?

Thread 1 (main thread) **Thread 2 (peer thread)**

stack 1

Thread 1 context:
Data registers
Condition codes
SP1
PC1

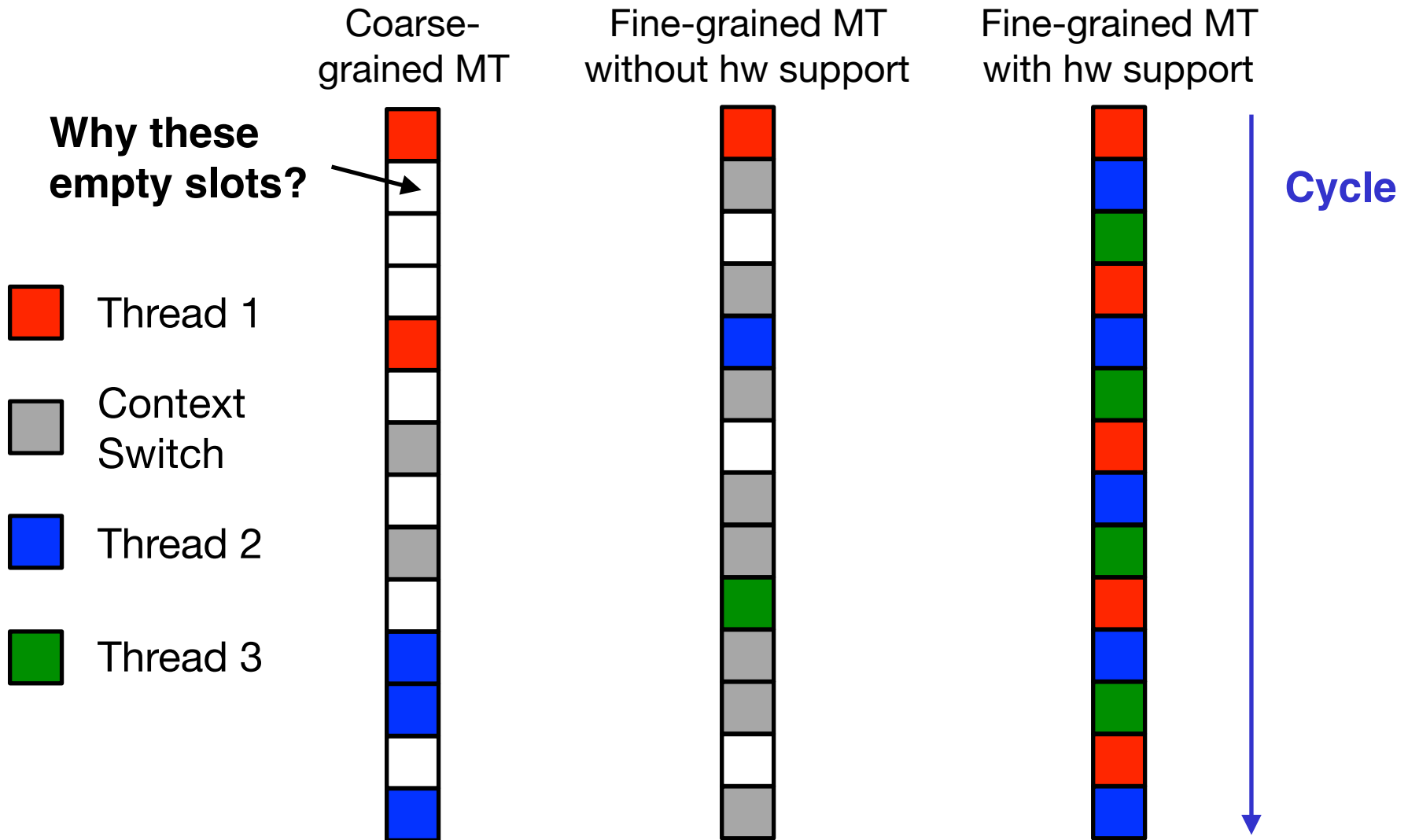
stack 2

Thread 2 context:
Data registers
Condition codes
SP2
PC2

Thread Switching

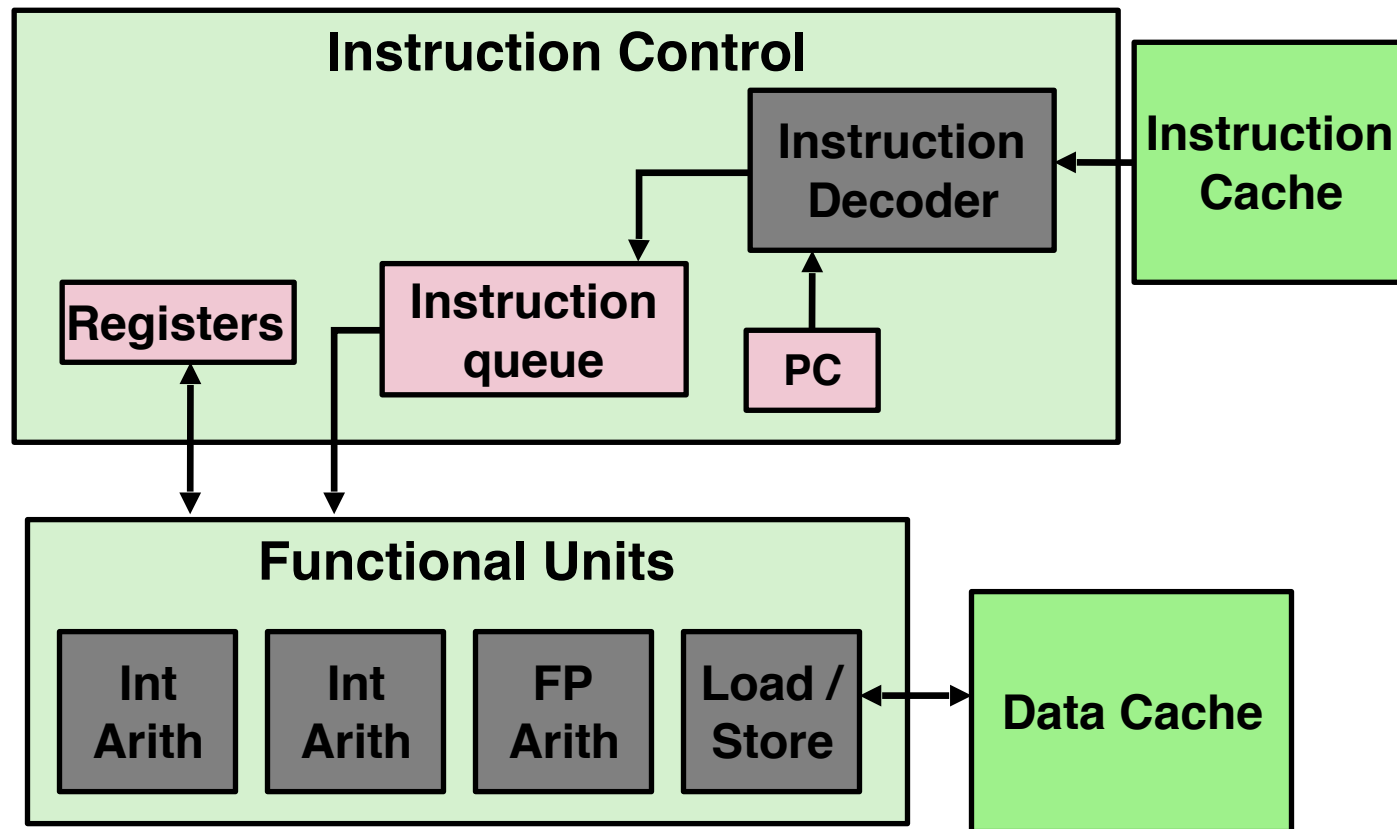
- A perhaps not great analogy: yellow traffic light
- Context switching is pure overhead, but you have to have it in order to support many threads with limited resources
- What does thread switching do?
 - Save the context for the old thread, and restore the context of the new thread
- How do you implement thread context switching?
 - Save/restore thread contexts in memory
 - Have dedicated context for each thread (e.g., each thread has a dedicated register file)
 - Lots of hardware resources, but is a must if we want to support a lot of threads.
 - GPU does this (later).
 - CPU does this for a limited number of threads (hyper-threading, later).

Multi-threading Illustration (so far...)



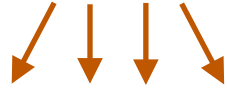
Modern Single-Core: Superscalar

- Typically has multiple function units to allow for decoding and issuing multiple instructions at the same time
- Called “Superscalar”

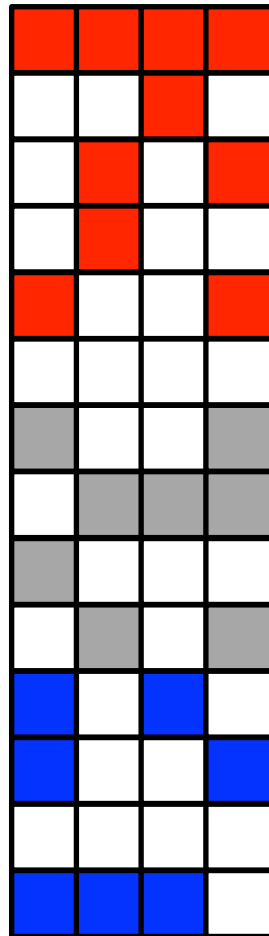


From Scalar to Multi-Scalar Multi-threading

Functional Units

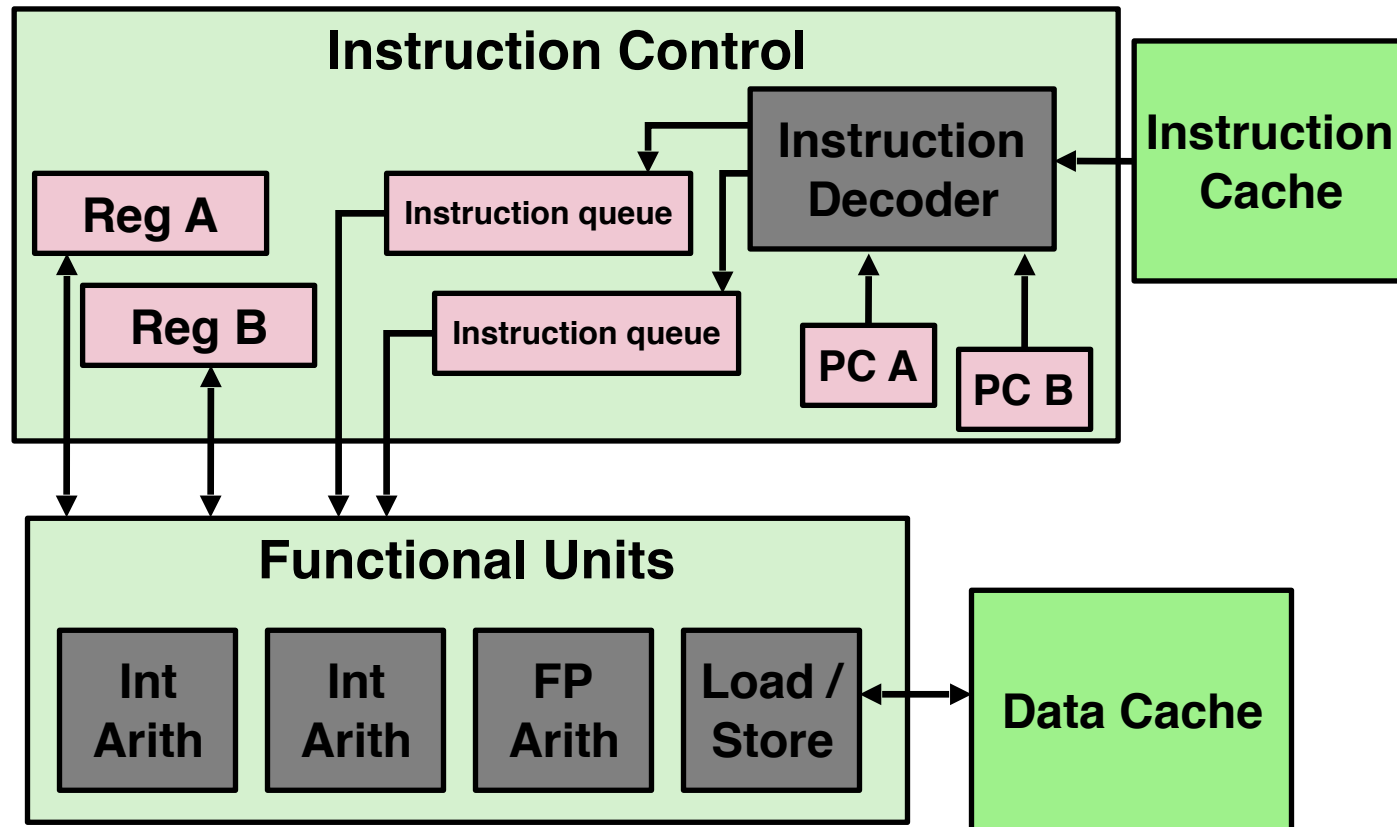


-  Thread 1
-  Context Switch
-  Thread 2



Simultaneous Multi-Threading (SMT)

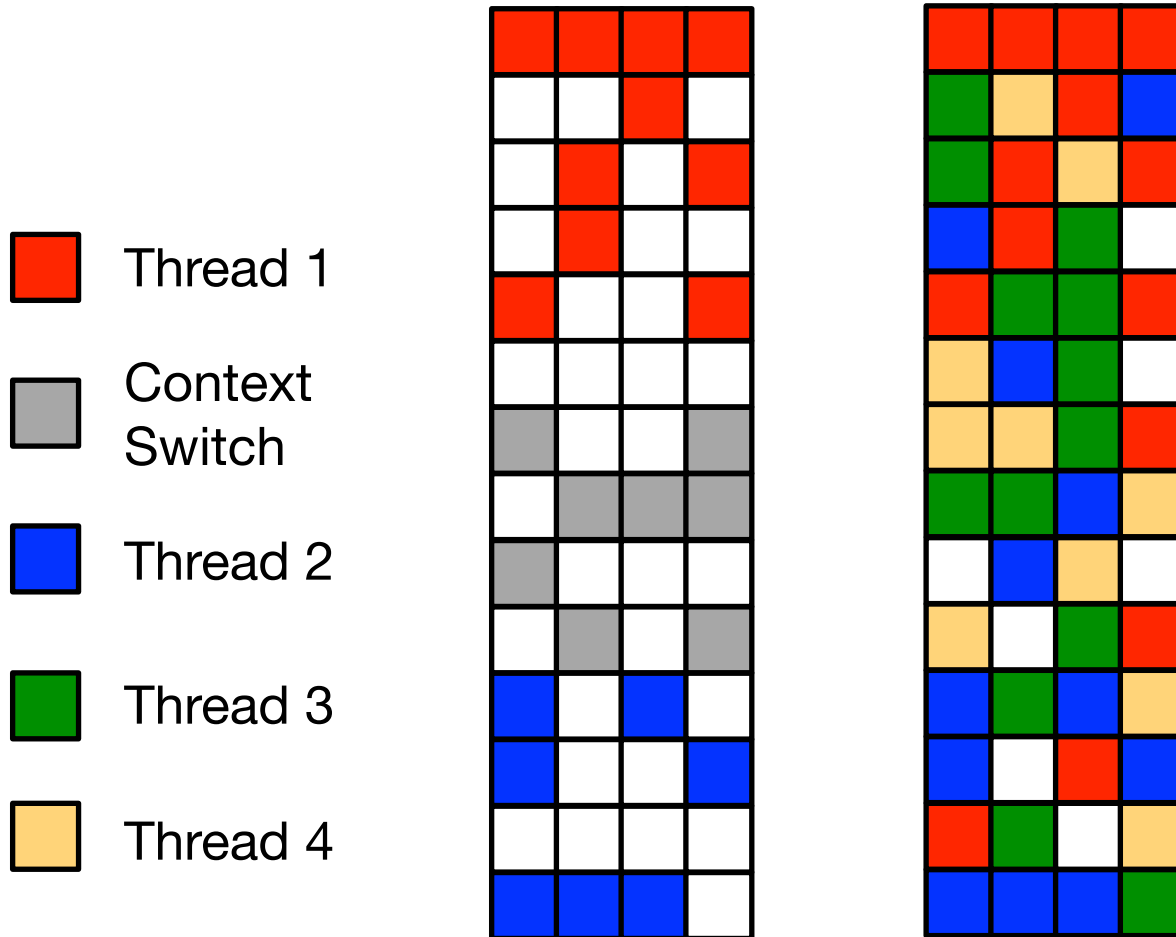
- Intel call it hyper-threading.
- Replicate enough hardware structures to process K instruction streams, i.e., threads. K copies of all registers. Share functional units.
- SMT = Superscalar + Multi-threading



Conventional Multi-threading vs. Hyper-threading

Coarse-grained MT on a superscalar core

SMT



Can now make use of idle issue slots in conventional MT cores.

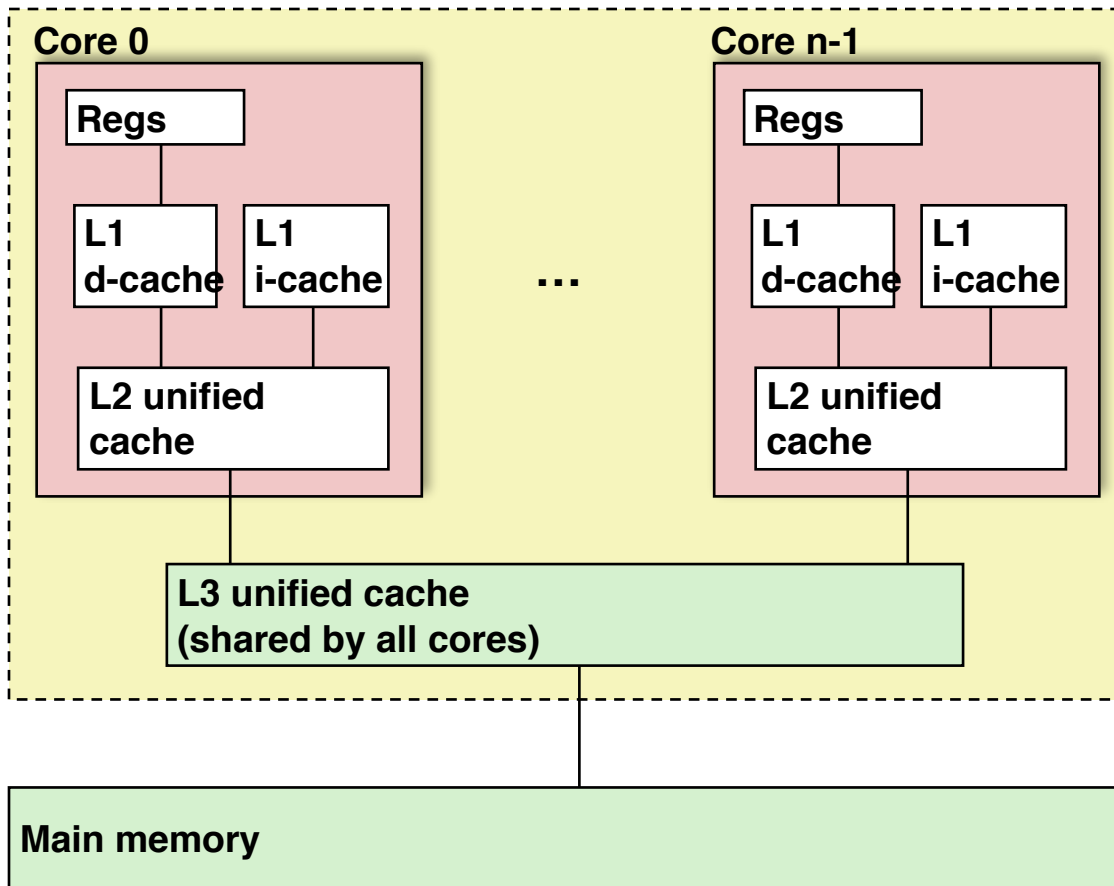
Multiple threads actually execute in parallel (even with one single core)

No/little context switch overhead

Today

- From process to threads
 - Basic thread execution model
- Multi-threading programming
- **Hardware support of threads**
 - Single core
 - **Multi-core**
 - Cache coherence

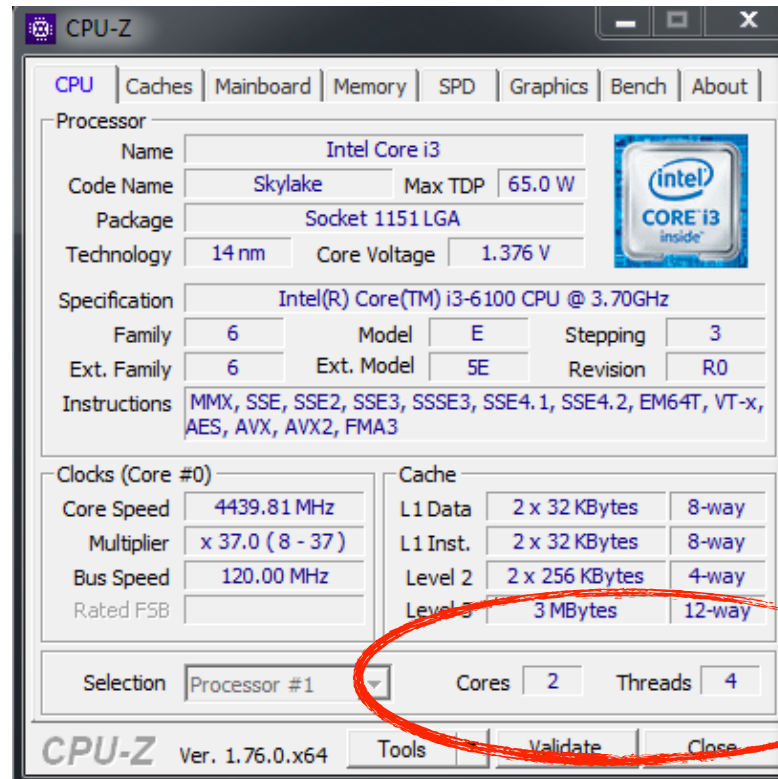
Multi-Threading on a Multi-core Processor



- Each core can run multiple threads, mostly through coarse-grained switching.
- Fine-grained switching on conventional multi-core CPU is too costly.

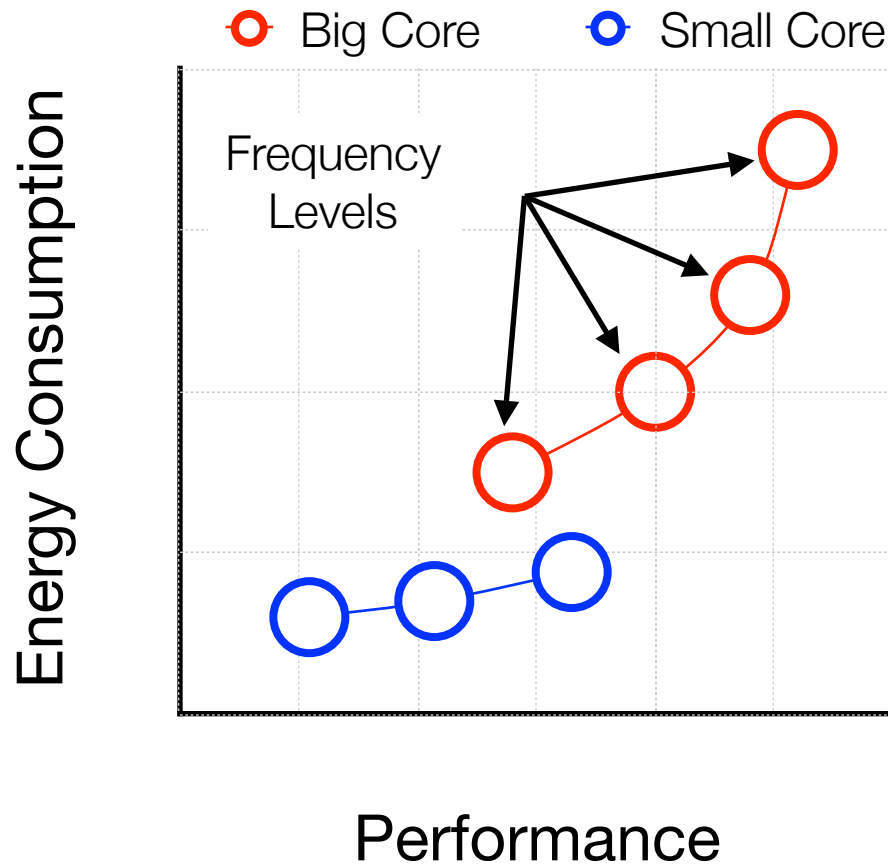
Combine Multi-core with SMT

- Common for laptop/desktop/server machine. E.g., 2 physical cores, each core has 2 hyper-threads => 4 virtual cores.
- Not for mobile processors (Hyper-threading costly to implement)



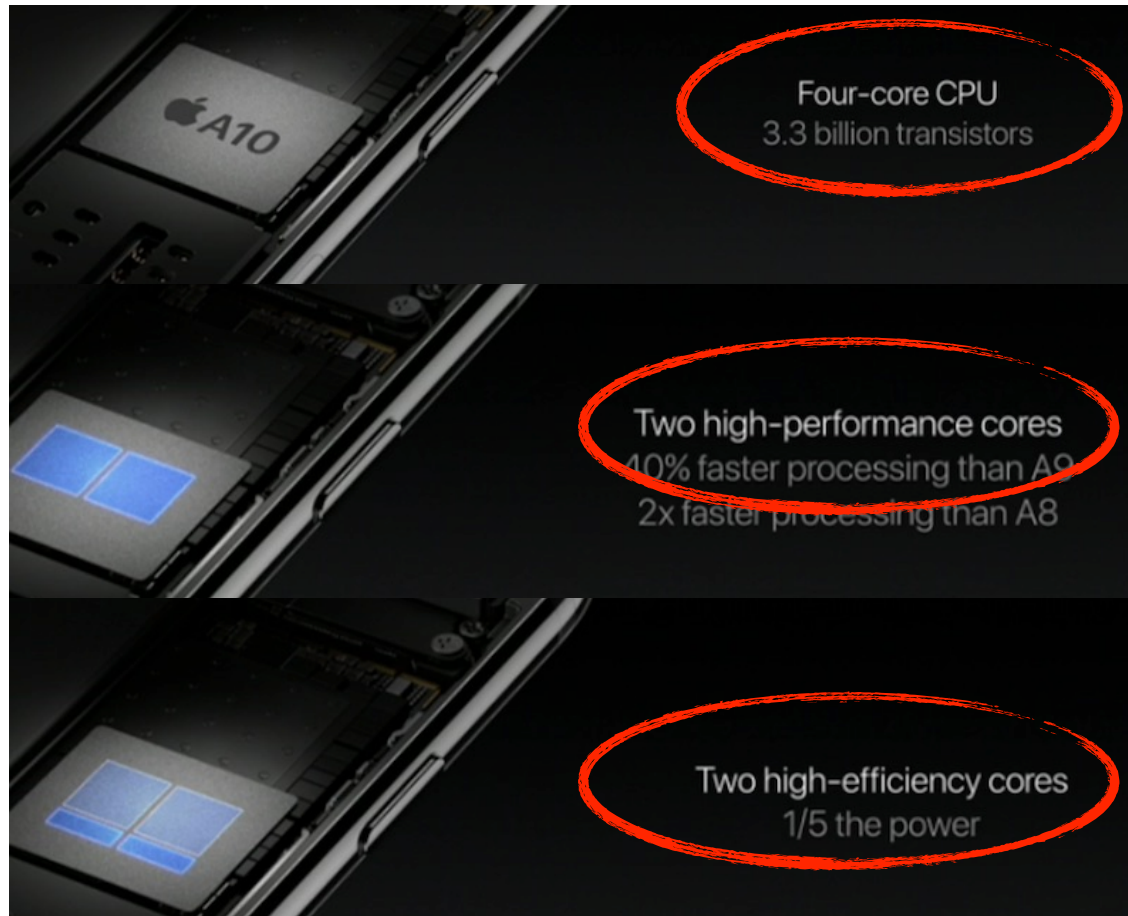
Asymmetric Multiprocessor (AMP)

- Offer a large performance-energy trade-off space



Asymmetric Chip-Multiprocessor (ACMP)

- Already used in commodity devices (e.g., Samsung Galaxy S6, iPhone 7)



Today

- From process to threads
 - Basic thread execution model
- Multi-threading programming
- **Hardware support of threads**
 - Single core
 - Multi-core
 - Cache coherence

The Issue

- Assume that we have a multi-core processor. Thread 0 runs on Core 0, and Thread 1 runs on Core 1.
- Threads share variables: e.g., Thread 0 writes to an address, followed by Thread 1 reading.
- Each read should receive the value last written by anyone
- **Basic question:** If multiple cores access the same data, how do they ensure they all see a consistent state?

Thread 0

Mem[A] = 1

Thread 1

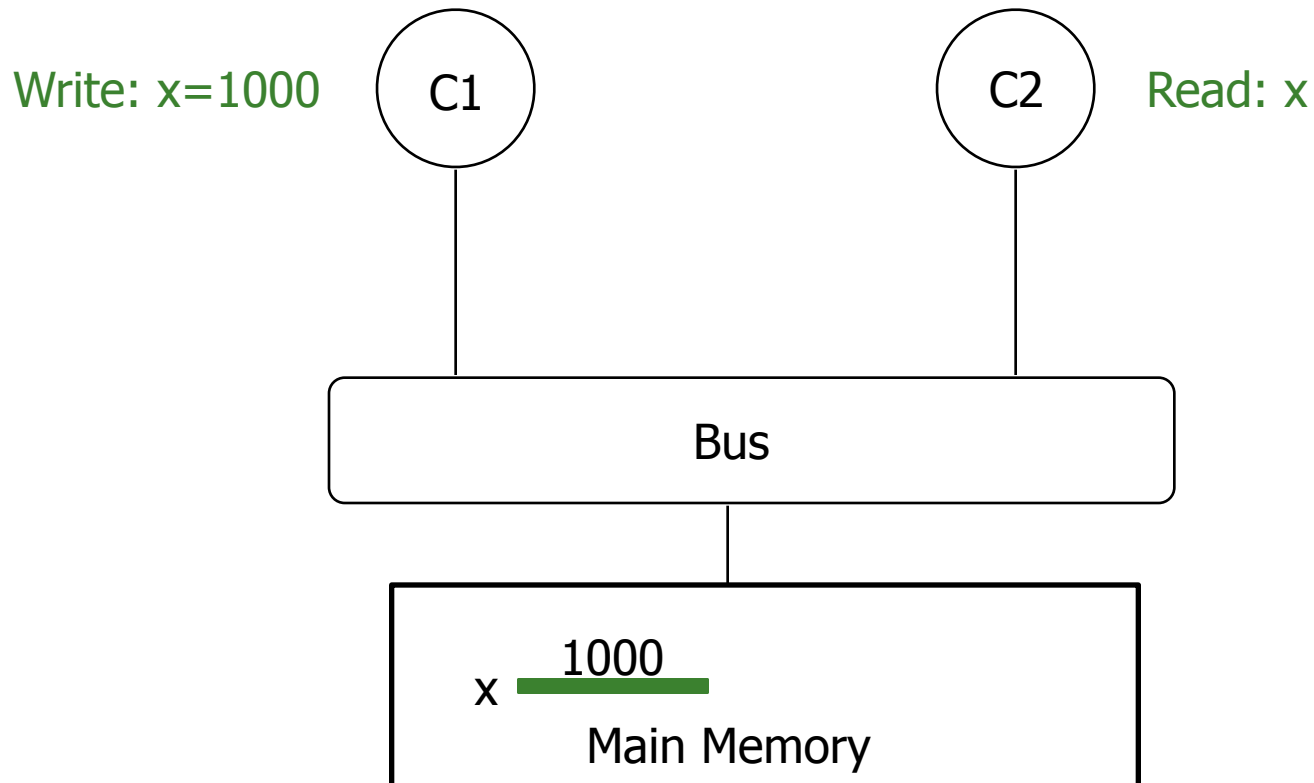
...

Print Mem[A]

The diagram illustrates a race condition in a multi-core system. On the left, under the heading Thread 0, the code `Mem[A] = 1` is shown. On the right, under the heading Thread 1, the code `...` is shown above a dashed horizontal line, followed by `Print Mem[A]`. An arrow points from the assignment `Mem[A] = 1` in Thread 0 to the `Print Mem[A]` statement in Thread 1, crossing the dashed line. This represents Thread 1 reading the memory after Thread 0's write, but the dashed line suggests a potential ordering or visibility issue.

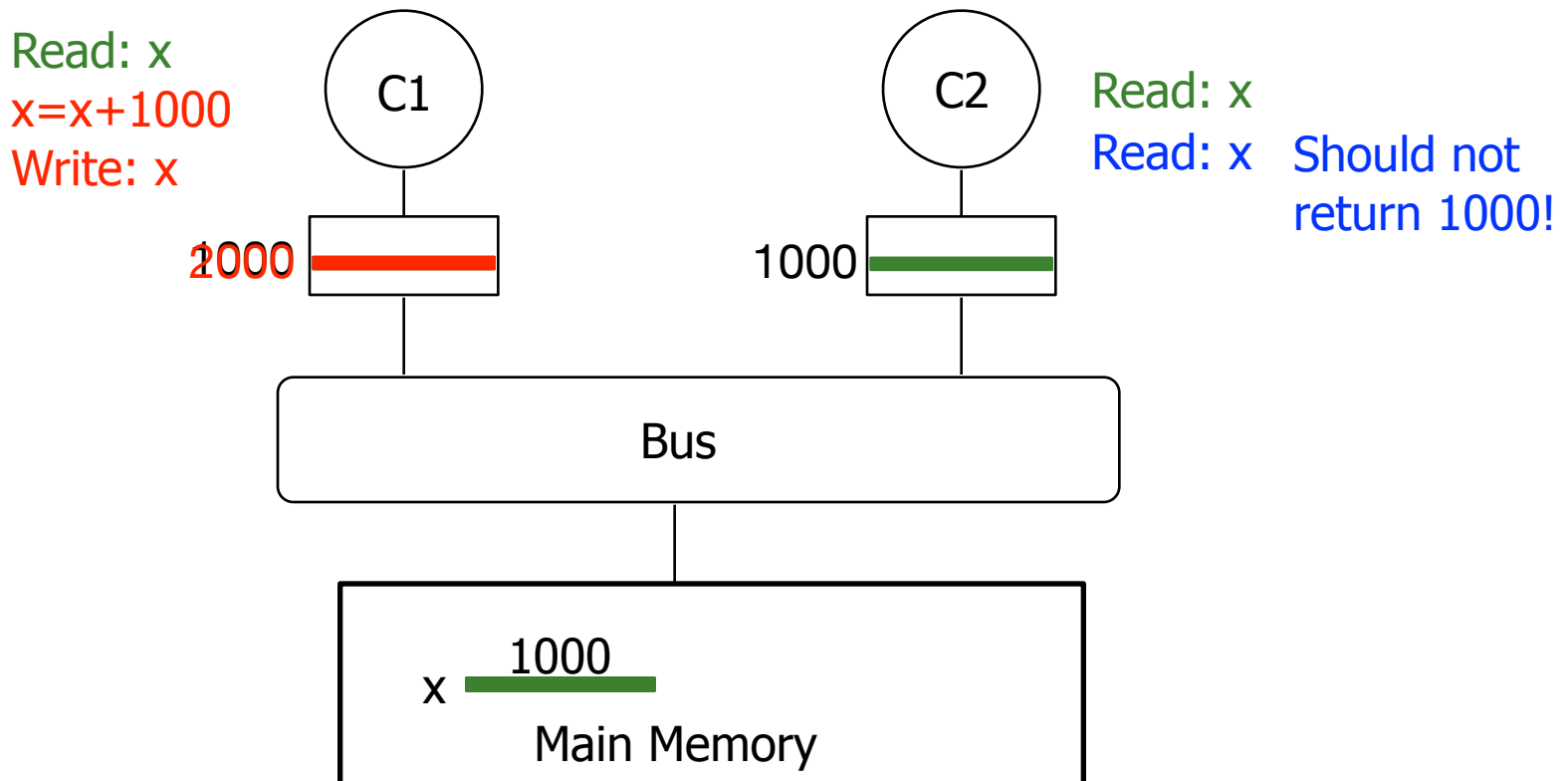
The Issue

- Without cache, the issue is (theoretically) solvable by using mutex.
- ...because there is only one copy of x in the entire system. Accesses to x in memory are serialized by mutex.



The Issue

- What if each core **cache** the same data, how do they ensure they all see a consistent state? (assuming a write-back cache)



Cache Coherence: The Idea

- **Issue:** there are multiple copies of the same data in the system, and they could have different values at the same time.
- **Idea:** ensure multiple copies have same value, i.e., *coherent*
- **How?** Two options:
 - **Update:** push new value to all copies (in other caches)
 - **Invalidate:** invalidate other copies (in other caches)

Readings: Cache Coherence

- Most helpful

- Culler and Singh, Parallel Computer Architecture
 - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
- Patterson&Hennessy, Computer Organization and Design
 - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)
- Papamarcos and Patel, “[A low-overhead coherence solution for multiprocessors with private cache memories](#),” ISCA 1984.

- Also very useful

- Censier and Feautrier, “[A new solution to coherence problems in multicache systems](#),” IEEE Trans. Computers, 1978.
- Goodman, “[Using cache memory to reduce processor-memory traffic](#),” ISCA 1983.
- Laudon and Lenoski, “[The SGI Origin: a ccNUMA highly scalable server](#),” ISCA 1997.
- Martin et al, “[Token coherence: decoupling performance and correctness](#),” ISCA 2003.
- Baer and Wang, “[On the inclusion properties for multi-level cache hierarchies](#),” ISCA 1988.

Does Hardware Have to Keep Cache Coherent?

- Hardware-guaranteed cache coherence is complex to implement.
- Can the programmers ensure cache coherence themselves?
- Key: ISA must provide cache flush/invalidate instructions
 - FLUSH-LOCAL A: Flushes/invalidates the cache block containing address A from a processor's local cache.
 - FLUSH-GLOBAL A: Flushes/invalidates the cache block containing address A from all other processors' caches.
 - FLUSH-CACHE X: Flushes/invalidates all blocks in cache X.
- Classic example: TLB
 - Hardware does not guarantee that TLBs of different core are coherent
 - ISA provides instructions for OS to flush PTEs
 - Called "TLB shutdown"