*To know if I should extend the HW3 deadline, please take a moment to complete the linked form: https://forms.cloud.microsoft/r/UeFtH7vcTc

More Multicore Architectures

Last lab tonight!
HW3 due Friday*

Outline

- Multicore architecture design decisions
- Examining synchronization primitives in hardware
- Designing software for thread-level parallelism

(From Monday) The Speedup Pitfall with Amdahl's Law

Takeaway: programs require very high percentages of parallelizable regions to fully benefit from multicore!

- To understand speedup due to thread-level parallelism, we need to understand how much of the program is parallelizable versus being sequential
- Amdahl's Law: speedup = ((1 - % parallel) + (% parallel / speedup parallel))$^{-1}$
- Example: suppose a program that is 85% parallelizable is written for 100 cores with 75 threads

Speedup = ((1 - .85) + (.85 / 75))$^{-1}$
Speedup = 1 / (.15 + .0113)
Speedup = 6.2 times speedup

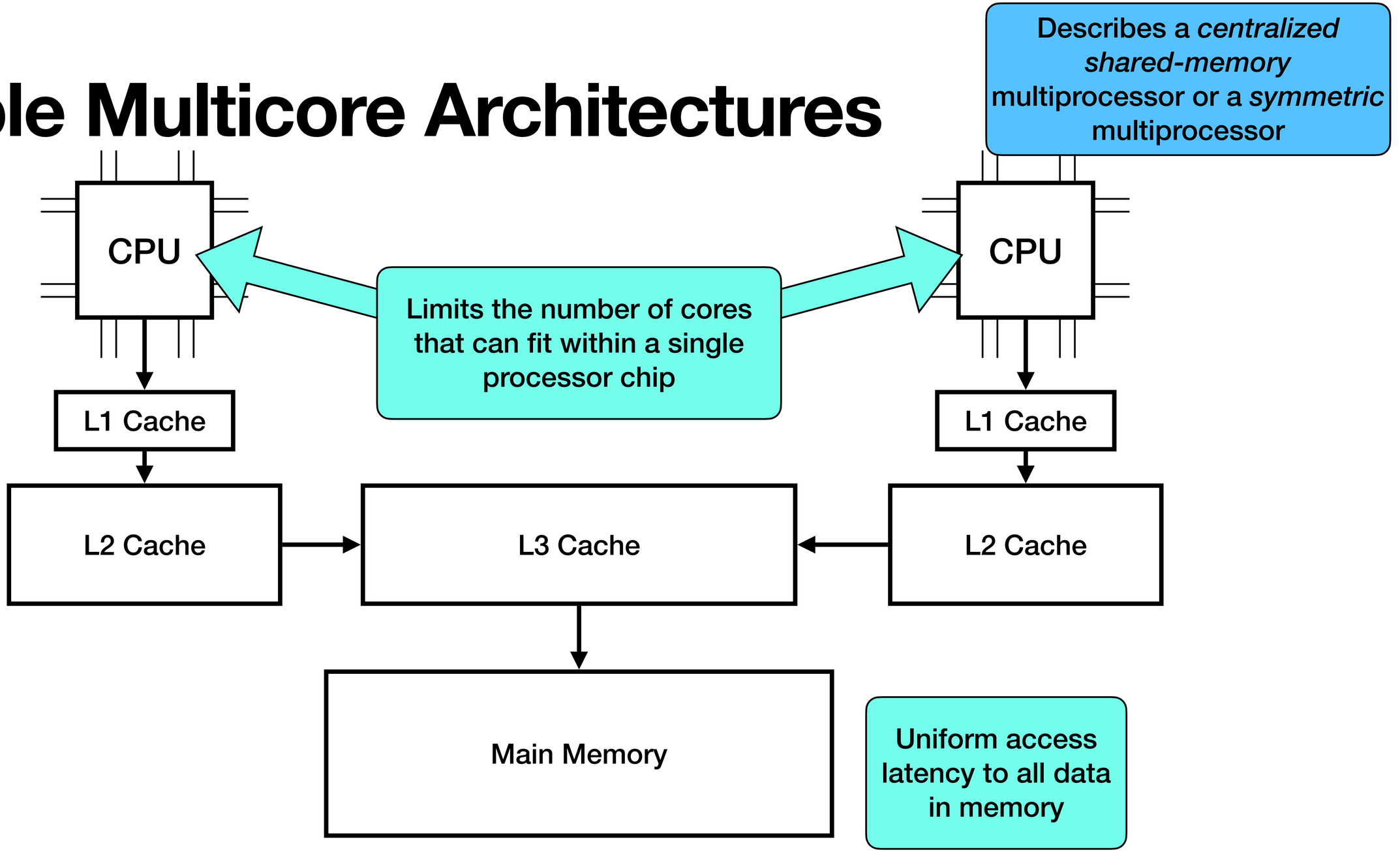
100 threads?

Speedup = ((1 - .85) + (.85 / 100))$^{-1}$
Speedup = 1 / (.15 + .0085)
Speedup = 6.3 times speedup

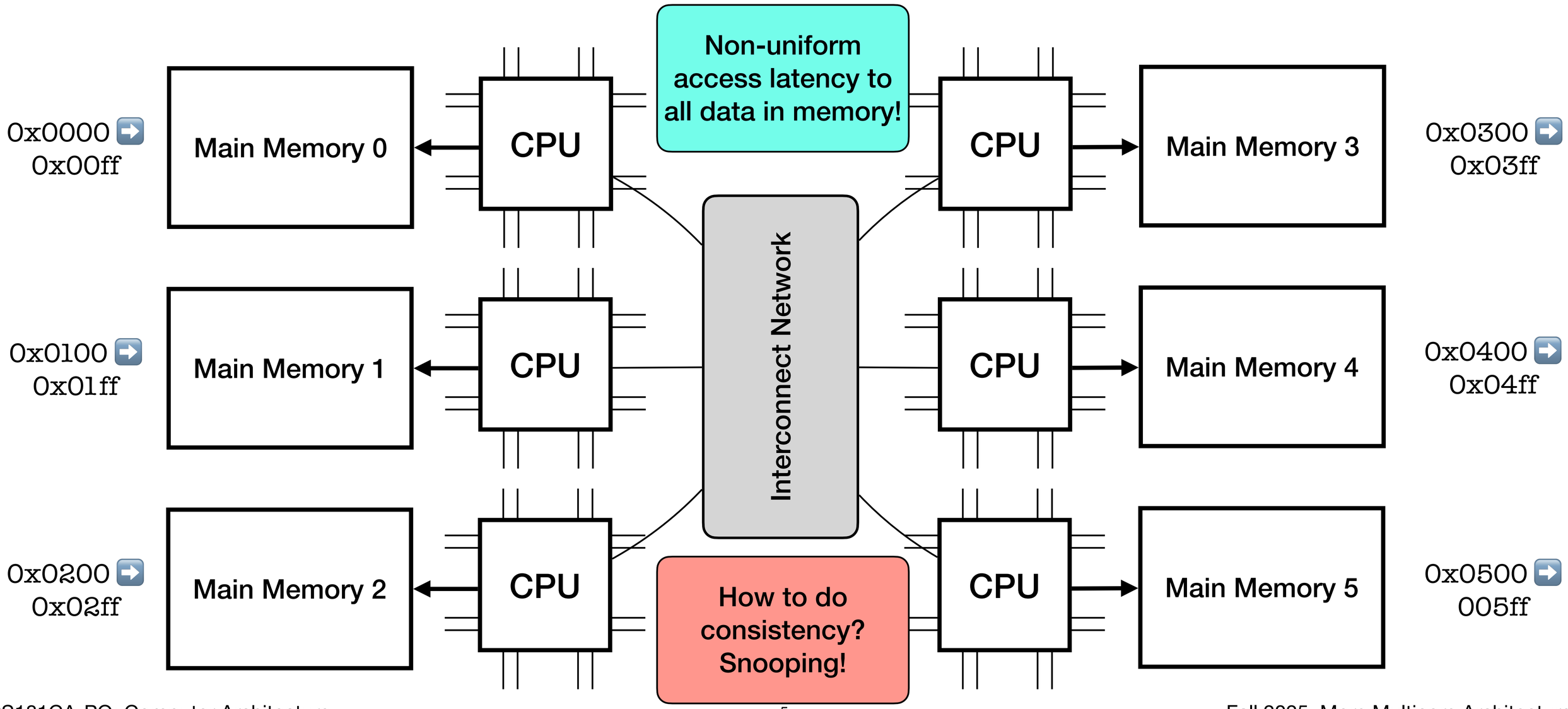
How parallel to get 80x?

80 = ((1 - p) + (p / 100))$^{-1}$
80((1 - p) + (p / 100)) = 1
80(1 - p) + 80(p / 100) = 1
80 - 80p + .8p = 1
-79.2p = -79
p = .9974

Simple Multicore Architectures



Distributed Memory Multiprocessor Architectures



Multiprocessor Architectures

- A multiprocessor may implement a simple *symmetric* multiprocessor architecture or a *distributed shared memory* multiprocessor
- Symmetric multiprocessors are limited in the number of processors that can fit within a single chip due to limitations of shared bus widths, etc…
- Processors in a distributed shared memory multiprocessor architecture maintain a small subset of the address space close to their core ➡️ for that processor, this is referred to as *local memory*
- Fetching or storing data in local memory is performed at a lower latency than performing the equivalent accesses in a *remote memory* (e.g., a subset of the address space not associated with that processor) due to traversing the interconnect network

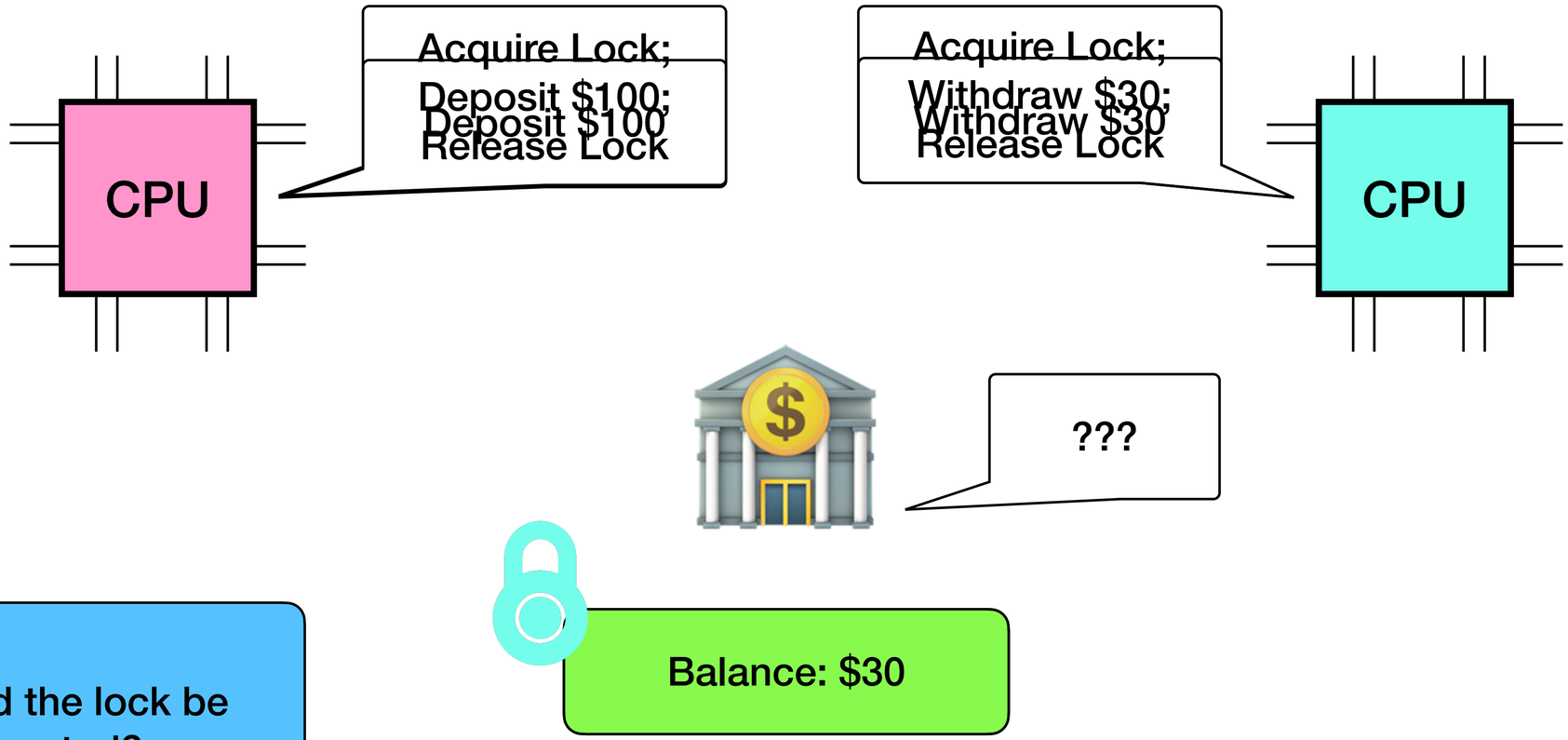
Chat with your neighbor(s)!

Consider the distributed shared memory multiprocessor architecture. How might you implement a shared last-level cache (e.g., L3 cache)? Think about where the cache would need to reside and how each processor would access it.

Just like how memory may be distributed into non-uniform memory access, caches can have non-uniform access as well!

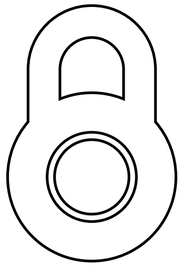
If we want to avoid a NUCA architecture in a DSM, then we need to give up the semantics of shared caches!

Implementing a Multi-Threaded Program



How should the lock be implemented?

Implementing a Mutex (mutual exclusion)



What is meant by "atomic"?

```
class Lock {
    atomic<bool> held = false;

    void acquire() {
        while (!compare_exchange_weak(false, true)) {  };
    }

    void release() {
        held = false;
    }
};
```

cppreference.com

Search

Page | Discussion

C++ | Concurrency support library | std::atomic

std::atomic

Defined in header <atomic>

```
template< class T >
struct atomic;                                  (1)  (since C++11)

template< class U >
struct atomic<U*>;                              (2)  (since C++11)
```

Defined in header <memory>

```
template< class U >
struct atomic<std::shared_ptr<U>>;              (3)  (since C++20)

template< class U >
struct atomic<std::weak_ptr<U>>;                (4)  (since C++20)
```

Defined in header <stdatomic.h>

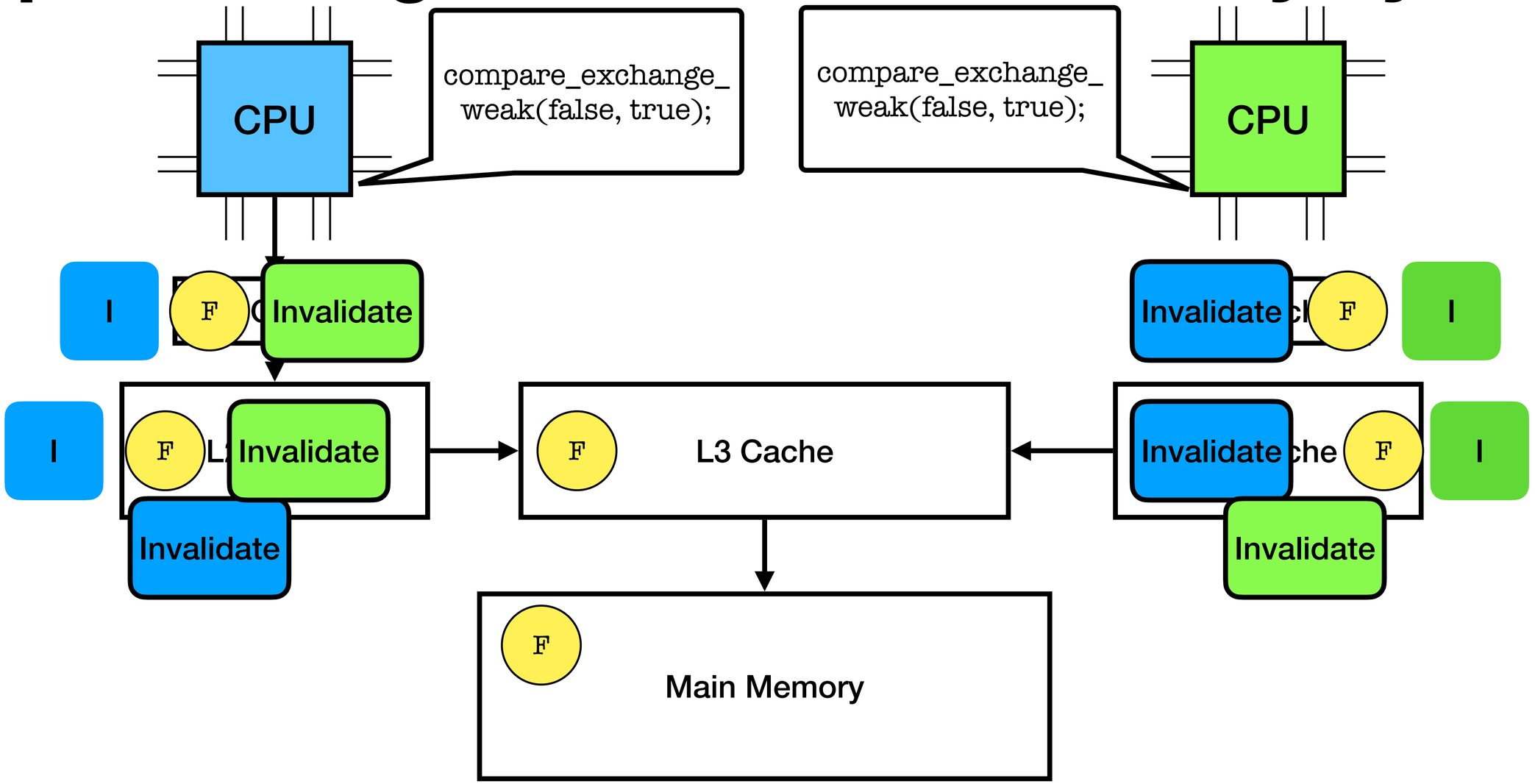
```
#define _Atomic(T) /* see below */               (5)  (since C++23)
```

Each instantiation and full specialization of the `std::atomic` template defines an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined (see memory model for details on data races).

In addition, accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by `std::memory_order`.

`std::atomic` is neither copyable nor movable.

Implementing a Mutex in the Memory System



Implementing a Better Mutex

- To call "compare_exchange_weak" means that many coherence messages will be sent throughout the memory system to modify the shared variable!
- To reduce the coherence traffic in the memory system, we can implement a *test-and-set lock* (TAS) to minimize the data race
- Reducing coherence traffic will benefit the thread that currently holds the lock!

```
void acquire() {
    while (true) {
        while (held) {  };
        if (compare_exchange_weak(false, true)) {
            return;
        }
    }
};
```

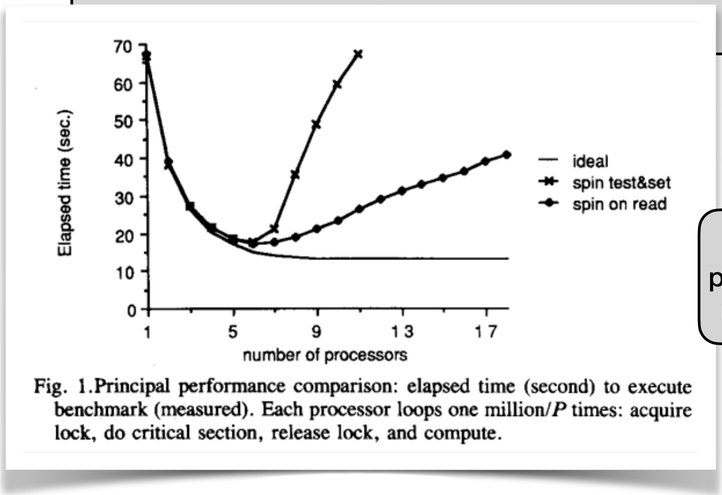


Fig. 1.Principal performance comparison: elapsed time (second) to execute benchmark (measured). Each processor loops one million/P times: acquire lock, do critical section, release lock, and compute.

Image Credit: https://pdos.csail.mit.edu/6.828/2010/readings/anderson-locks.pdf

Chat with your neighbor(s)!

Suppose we are implementing a multithreaded program with a mutex. What coherence traffic would you expect in the memory system to update the bank account balance?

When using a lock, the coherence of data in the “critical section” should be more trivial for the memory system to handle!

Takeaways

- Multiprocessor architectures can vary in their implementation to reach large scale deployments
- We can implement software constructs (e.g., a mutex) using special types that dictate the behavior in the memory system
- If we understand how coherence is implemented in the memory system, we can write better software the outperforms