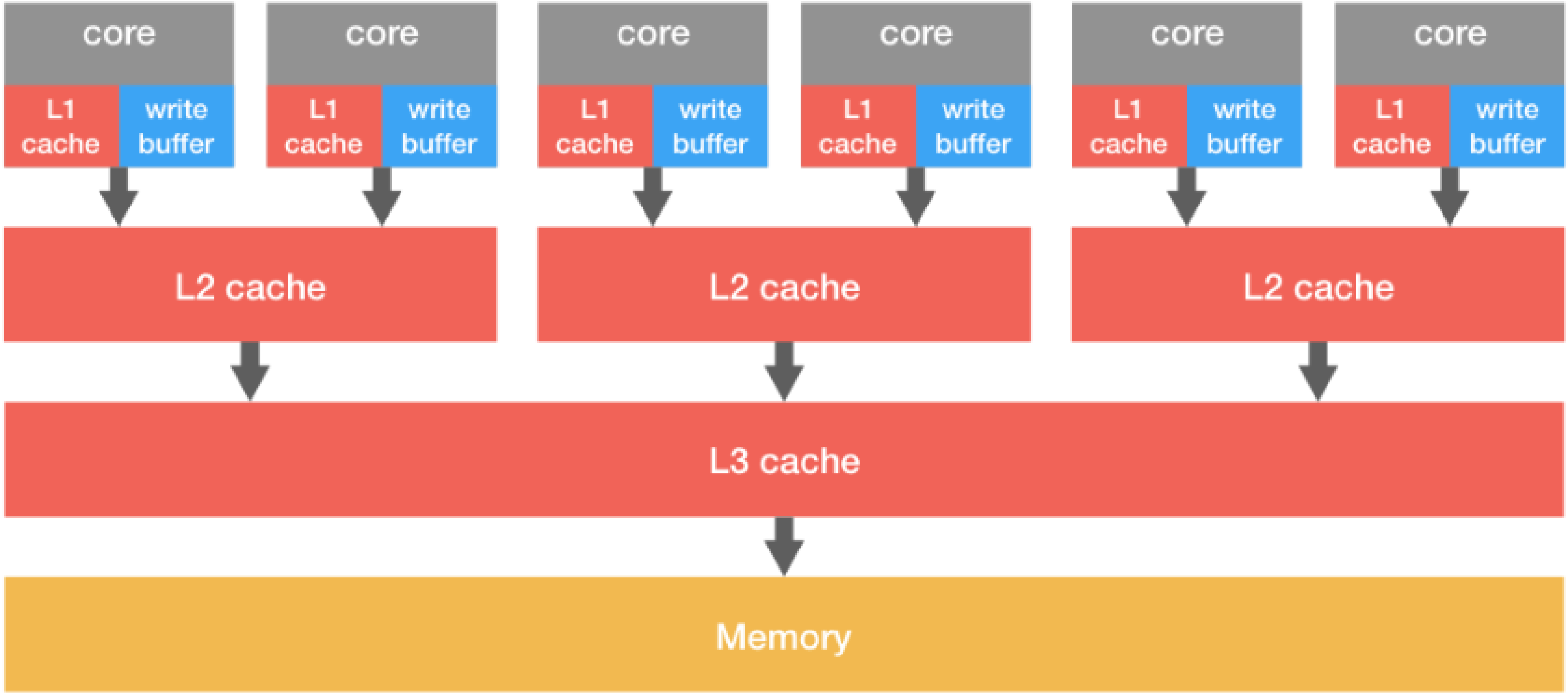
ECE 421

Lecture 11

Concurrency

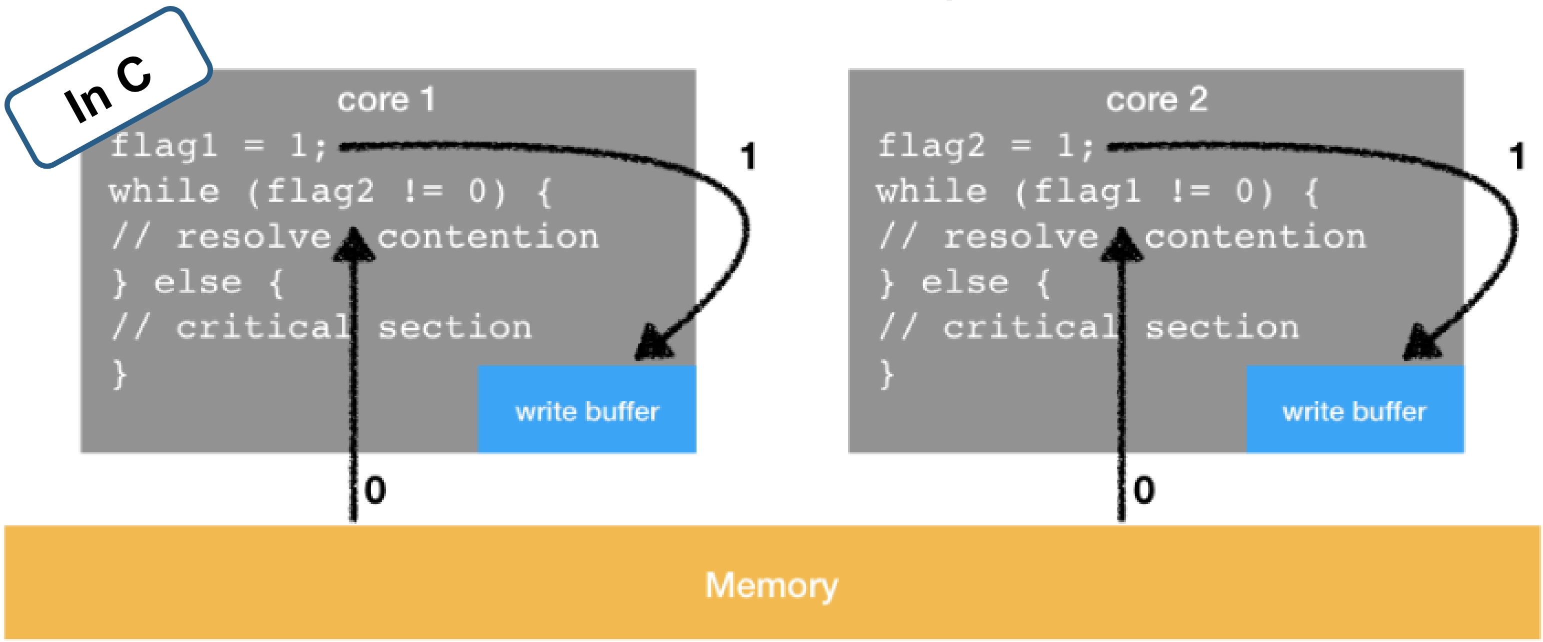
Why parallelism is hard?

- The hardware, the OS, and the compilers are too complex.
- Since 1970 processor cores do not work with the memory directly and instead use a complex hierarchy of caches and write buffers.



Why parallelism is hard?

- Let us remove all the cache and consider the write buffers only.
- Write buffers are absolutely essential for the performance of the processor since writes to the memory are expensive and we want to batch them as much as possible.



Why parallelism is hard?

- Another way of thinking about the problem is to think that the write buffer has reordered the order of operations by executing

  flag2 != 0 before flag1 = 1.
- Similarly, we can think that caches also reorder the operations.
- Operations are reordered by the compiler that does optimizations.
- As a result, the order of operations in the source code will be different from the order executed by a specific core.
- The same code can have a different order of operations when executed in parallel on two separate cores.

Why parallelism is hard?

- The order of operations would not be a problem if we were not using threads running on different cores to collaborate with each other.
- Collaborating threads require us to argue that operation X happened on thread A before operation Y on thread B
- Multithreading requires us to be able to talk about causality between operations across threads.

Without special tools that wouldn't be possible!

Threads

Splitting a program into threads.

```
use std::thread::spawn;
fn main(){
     spawn(||{
        for i in 0..10{
            println!("{}",i);
         }
     });
     for i in 10..20{
        println!("{}",i);
     }
     println!("Done");
}
```

- Spawn creates the new thread.
- It is a function which takes a function as a parameter.
- We can write normal functions or use closures.

Splitting a program into threads.

```
//output:
10
11
12
13
14
15
16
17
18
19
Done
//possible to see some output from the other thread, but
completely random
```

- The program has only run the latter part of that. We've not seen the print from the threads because when the program reaches its end, all other threads are killed.

Splitting a program into threads.

- If we don't want that to happen then we can call **h.join**, and that will wait for the first thread to finish before completing.

```
use std::thread::spawn;
fn main(){
    let h=spawn(||{
        for i in 0..10{
            println!("{}",i);
         }
     });
     for i in 10..20{
        println!("{}",i);
     }
     h.join().expect("Error");
     println!("Done");
}
```

Splitting a program into threads.

```
//output:
10.. 19
0..9
Done
```

Splitting a program into threads.

- Because **h.join** returns a result, we can pull information out of that result. What if our function for exam was to return 5?

```
use std::thread::spawn;
fn main(){
    let h=spawn(||{
        for i in 0..10{
            println!("{}",i);
         }
         return 5;
     });
     for i in 10..20{
        println!("{}",i);
     }
     let r = h.join().unwrap();
     println!("Done R= {}",r);
}
```

Splitting a program into threads.

```
//output:
10..19
0..9
Done R= 5
```

Splitting a program into threads.

- Now, every time we've run this so far, we've always finished the one thread before the other really got to get started.
- So, let's perhaps make the one run a little bit slower:

```
use std::thread::{spawn,sleep};
use std::time::Duration;
fn main(){
    let h=spawn(||{
        for i in 0..10{
            sleep(Duration::from_millis(10));
            println!("{}",i);
         }return 5;
     });
     for i in 10..20{
        sleep(Duration::from_millis(10));
        println!("{}",i);      }
     let r = h.join().unwrap();
     println!("Done R= {}",r);
}
```

Splitting a program into threads.

- When we run the program, hopefully we'll find that the two sets of numbers interlace as they both wait and take it in turns.

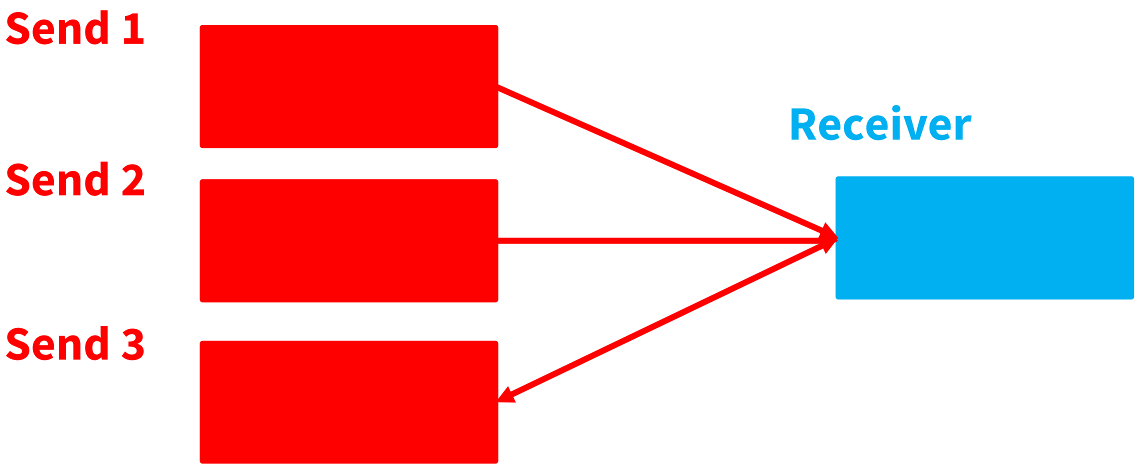
```
//output:
0
10
1
11
12
2
3
13
4
14
```

```
15
5
16
6
17
7
18
8
19
9
Done R= 5
```

Channels

Channels

- multiple-producer, single-consumer channel (mpsc).



Channels

- Channels allow us to pass data between threads safely.

```
use std::sync::mpsc::{Sender, Receiver};
use std::thread;
static NTHREADS: i32 = 3;
fn main() {
    let (tx, rx): (Sender<_>, Receiver<_>) = mpsc::channel();
    let mut children = Vec::new();
    for id in 0..NTHREADS {
        let thread_tx = tx.clone();
        let child = thread::spawn(move || {
               thread_tx.send(id).unwrap();
               println!("thread {} finished", id);
        });
        children.push(child);     }
    let mut ids = Vec::with_capacity(NTHREADS as usize);
    for _ in 0..NTHREADS {ids.push(rx.recv());}
    for child in children {child.join().expect("oops! the
child thread panicked");}
    println!("{:?}", ids);
}
```

Channels

```
   Compiling
    Finished
     Running
thread 0 finished
thread 2 finished
thread 1 finished
[Ok(0), Ok(2), Ok(1)]
```

Channels

```
fn main() {

    // Channels have two endpoints: the `Sender<T>` and the
`Receiver<T>`,
    // where `T` is the type of the message to be transferred
    // (type annotation is superfluous)

    let (tx, rx): (Sender<_>, Receiver<_>) = mpsc::channel();

    let mut children = Vec::new();
```

Channels

```
for id in 0..NTHREADS {

        // The sender endpoint can be copied
        let thread_tx = tx.clone();

        // Each thread will send its id via the channel

        let child = thread::spawn(move || {

            // The thread takes ownership over `thread_tx`
            // Each thread queues a message in the channel

            thread_tx.send(id).unwrap();

            // Sending is a non-blocking operation, the thread
will continue
            // immediately after sending its message

            println!("thread {} finished", id);
        });

        children.push(child);
    }
```

Channels

```
// Back in the main thread

// Here, all the messages are collected

    let mut ids = Vec::with_capacity(NTHREADS as usize);

    for _ in 0..NTHREADS {

        // The `recv` method picks a message from the channel
        // `recv` will block the current thread if there are
no messages available

        ids.push(rx.recv());
    }
```

Channels

```
 // Wait for the send threads to complete any remaining work

    for child in children {
      child.join().expect("oops! the child thread panicked");
    }

    // Show the order in which the messages were sent,
remember, receiving is in one thread.

    println!("{:?}", ids);
}
```