National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Concurrency

IOC5226 Operating System Capstone

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - MIT 6.828 Operating system engineering class, 2018
  - MIT 6.004 Operating system, 2018
  - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC

Outline

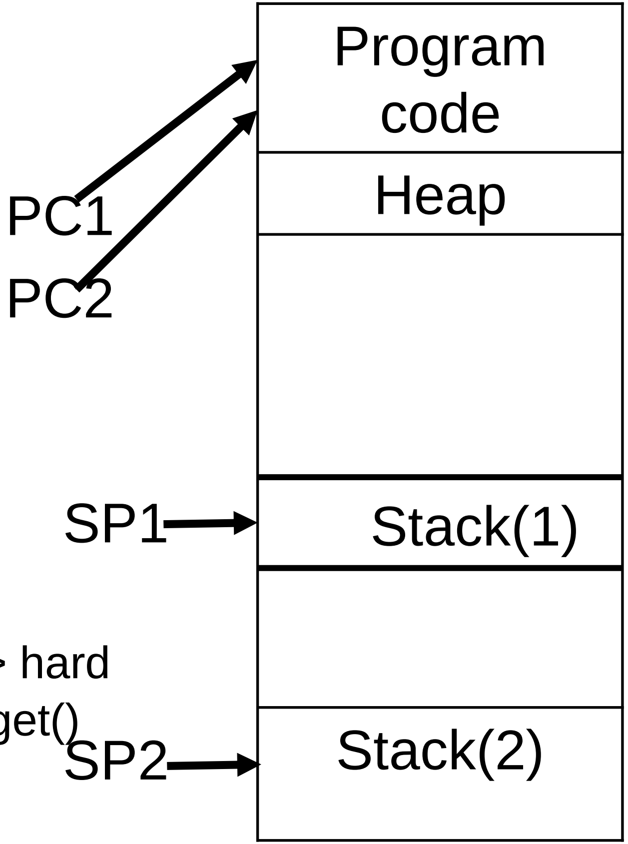
- Threads
- Race condition
- Peterson’s algorithm
- Bakery algorithm
- Memory consistency models

Introducing threads

- **Processes are “heavyweight”**
  - Memory mappings may be expensive to swap
  - Cache/TLB state: flushing expensive
  - Lots of kernel state
  - Context switching between process is expensive
- **Threads are “lightweight”**
  - Multiple threads share process state
    - Same address space
    - Same open file/socket tables
  - Making context switching between threads cheap

Recap: threads

- **Address space shared by threads**
  - Code
  - Data and heap
- **Thread private state**
  - Registers (pc, sp, psw)
  - Stack
- **Key issue**
  - How to safely access “shared” state ?
    - Read-only (e.g. code)-> easy , writable-> hard
    - Shared memory mapping, mmap(), shmget()



Why use threads ?

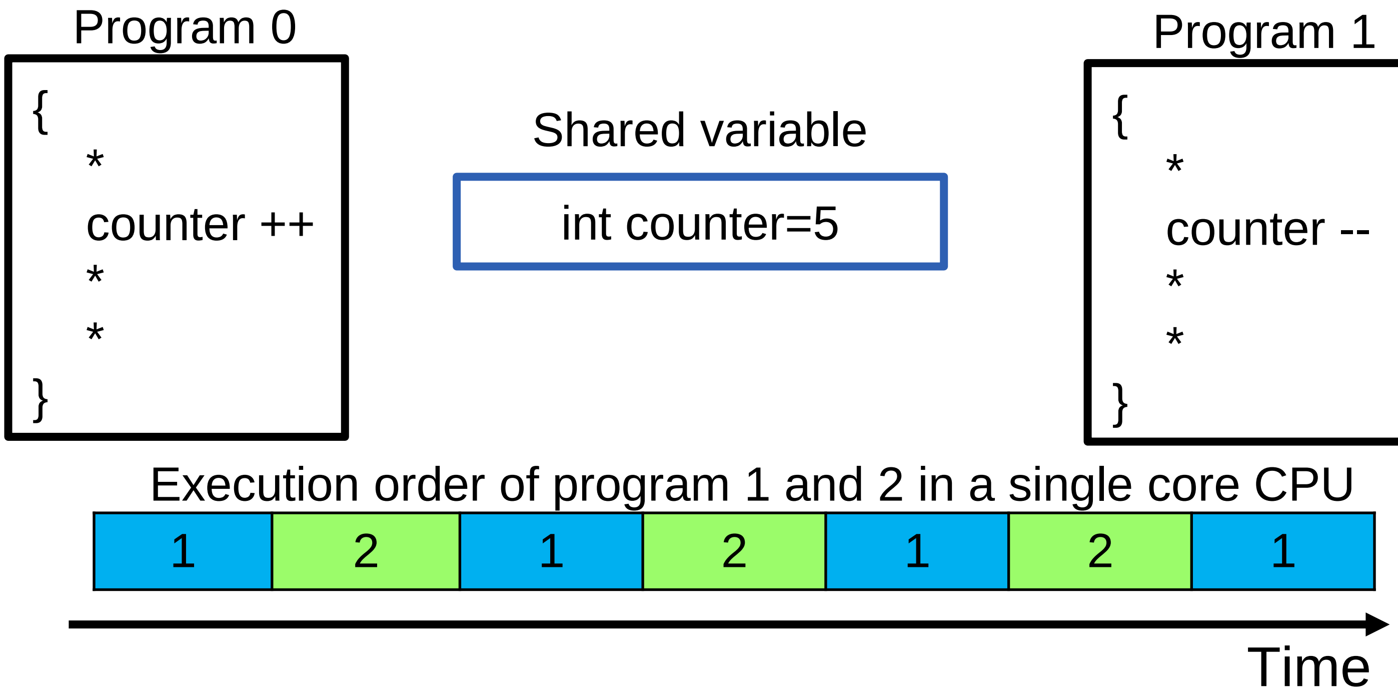
- **Multi-threading can provide benefits**
  - Improved performance by overlapping activities
- **Problems arise**
  - New failure modes introduced -> concurrency control
  - Errors often are hard to debug, or even to reproduce
- **Multiprogramming**
  - Higher overheads but great isolation
- **Multithreading**
  - Cooperation via shared memory
  - Faster context switches (why?)

Shared memory synchronization

- **Threads share memory**
- **Preemptive thread scheduling is a major problem**
  - Context switch can happen at any time, even in the middle of a line of code
    - Unit of atomicity -> machine instruction
  - Individual processes have little control over the order in which processes run
  - Preemptive scheduling introduces **non-determinism**

Sequential Scenario

- What is the expected value of counter in single core CPU ?



Indeterministic Scheduling

Program 0

```
{
    counter ++
}
```

Shared variable

int counter=5

Program 1

```
{
    counter --
}
```

Context switch

```
R1 <- counter
R1 <- R1 + 1
counter <- R1
- - - - - - - - - -
R2 <- counter
R2 <- R2 – 1
counter <- R2
```

counter = 5

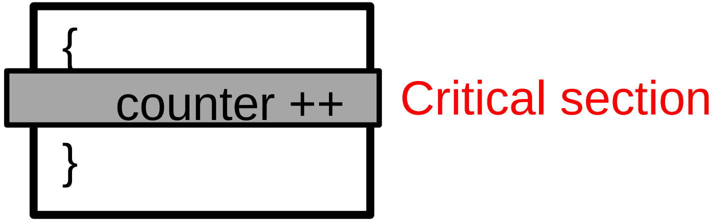
```
R1 <- counter
- - - - - - - - - -
R2 <- counter
R2 <- R2 - 1
counter <- R2
- - - - - - - - - -
R1 <- R1 + 1
counter <- R1
```

counter = 6

```
R2 <- counter
- - - - - - - - - -
R1 <- counter
R1 <- R1 + 1
counter <- R1
- - - - - - - - - -
R2 <- R2 – 1
counter <- R2
```

counter = 4

Race condition



- The results depend on the timing execution of the code
- **Critical section**
  - A piece of code that accesses a **shared** resource that is a variable or data structure
  - No more than one process should execute in critical section at a time
- **Race condition**
  - Multiple threads enter the critical section at roughly the same time
  - Both attempt to update the shared data structure
  - Leading to a undesired outcome

Critical section

- To avoid race condition, requirements on the critical section
  - **Mutual exclusion**
    - Guarantees that only a single thread ever enters a critical section
  - **Progress**
    - Any process that requires entry into the critical section must be permitted without any delay
  - **No starvation**
    - An upper bound on the number of times a process enters the critical section, while another is waiting

Synchronization

- **The race condition happened because**
  - There were conflicting accesses to a resource
- **Basic idea behind most synchronization**
  - When threads and processes have conflicting accesses
  - Force one of them to wait until it is safe to proceed
- **Difficult in practice (why?)**
  - The problem is that we need to protect all possible locations where two (or more) threads or processes might conflict

Atomic operations

- **Atomic: series of operations that cannot be interrupted**
  - This context means "as a unit" and we take as "all or none"
  - It could not be interrupt mid-instruction, no in-between state
  - Single instructions by themselves are atomic
    - e.g. add %eax, %ebx
  - Multiple instructions can be explicitly made atomic
  - Each piece of code in the OS must be checked if they need to be atomic

Busy waiting : Attempt 1

- Achieve mutual exclusion
- Busy waiting– waste power and time
- Static execution order in the critical section (How to resolve?)
  - Process 1 -> process 2 -> process 1 -> process 2

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2; // unlock
}
```

Shared variable

```
int turn = 1;
```

Process 2

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
}
```

Using two turn flags: Attempt 2

- Break the static execution in the critical section
- Don't guarantee mutual exclusion
  - The flag (p1_inside, p2_inside) is set after breaking from the while loop

Process 1

```
while(1){
    while(p2_inside == True); // lock
    p1_inside = True;
    critical section
    p1_inside = False; // unlock
}
```

Shared variable

P2_inside = False
P1_inside = False

Process 2

```
while(1){
    while(p1_inside == True); // lock
    p2_inside = True;
    critical section
    p2_inside = False; // unlock
}
```

Why attempt2 no mutual exclusion

Time ↓

CPU	p1_inside	p2_inside
while (p2_inside == True);	False	False
Context switch		
while (p1_inside == True);	False	False
p2_inside = True;	False	True
Context switch		
p1_inside = True;	True	True

Process 1

```
while(1){
  while(p2_inside == True); // lock
  p1_inside = True;
  critical section
  p1_inside = False; // unlock }
```

Shared variable

P2_inside = False
P1_inside = False

Process 2

```
while(1){
  while(p1_inside == True); // lock
  p2_inside = True;
  critical section
  p2_inside = False; // unlock}
```

Attempt 3: switching while and flag

- Achieve mutual exclusion
- What's problem of the implementation below ?

Process 1

```
while(1){
   p1_wants_to_enter = True;
   while (p1_wants_to_enter =
True);
   critical section
   p1_wants_to_enter = False;
}
```

Process 2

```
while(1){
  p2_wants_to_enter = True;
  while (p2_wants_to_enter =
True);
    critical section
   p2_wants_to_enter = False;
}
```

P2_wants_to_enter, P1_wants_to_enter

Globally defined

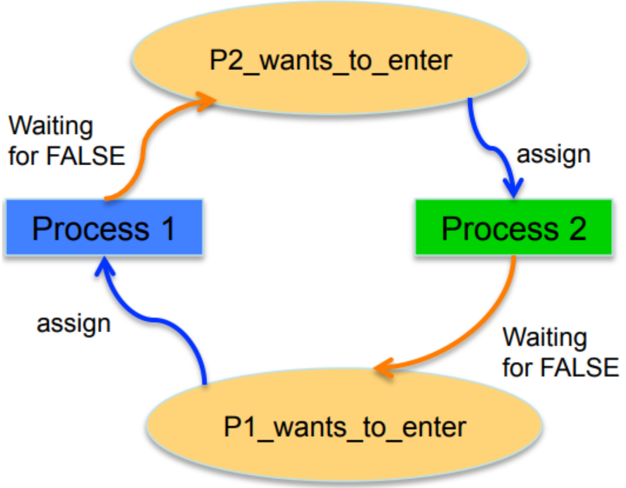
Attempt 3: No progress (Deadlock)

CPU	p1_inside	p2_inside
p1_wants_to_enter = True	False	False
Context switch		
P2_wants_to_enter = True	False	False

```
while(1){
  p2_wants_to_enter = True;
  while (p2_wants_to_enter =
True);
    critical section
   p2_wants_to_enter = False;
}
```

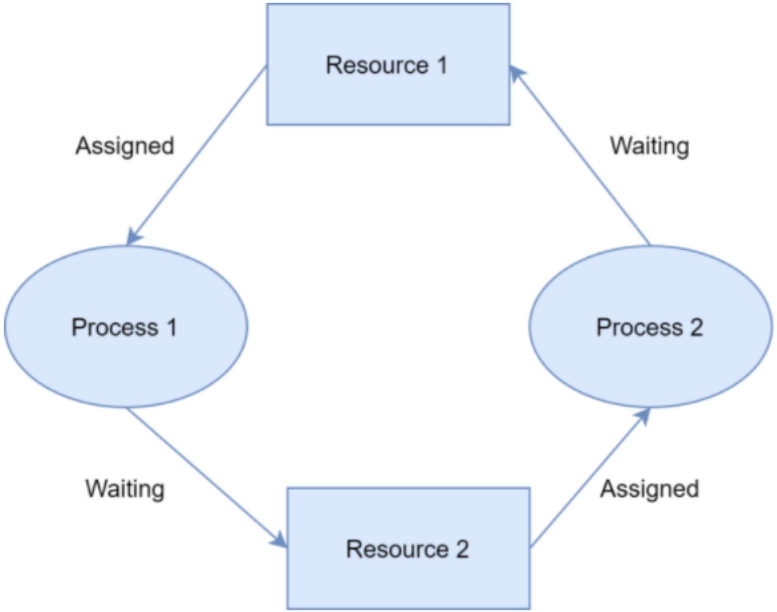
Deadlock

CPU	p1_inside	p2_inside
p1_wants_to_enter = True	False	False
Context switch		
P2_wants_to_enter = True	False	False



Deadlock

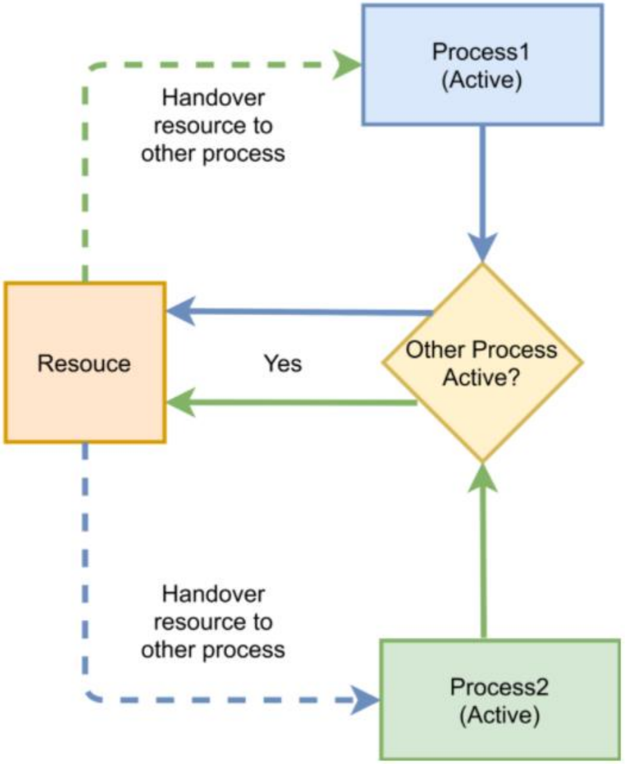
- **Deadlock**
  - Two or more threads are waiting or events that only those threads can generate
  - **Both processes are holding one resource** and **waiting for other resource** held by the other process
  - Thus, both processes cannot make progress until one of them gives up its resource



https://www.baeldung.com/cs/deadlock-livelock-starvation

Livelock

- **Livelock**
  - Thread blocked indefinitely by other thread(s) using a resource
  - Livelock naturally goes away when system load decreases
  - Both processes need a shared resource
  - Each one checks whether the other one is an active state
  - If so, it hands over the resource to the other process
  - **Both kept on handing over the resource to each other indefinitely**



https://www.baeldung.com/cs/deadlock-livelock-starvation

Conditions for deadlock

- **Mutual exclusion**
  - Resource cannot be shared
- **Hold and wait**
  - A thread is both holding a resource and waiting on another resource to become free
- **No preemption**
  - A cycle in the graph

Deadlock free condition

- Given a system has
  - R identical resources, P processes compete for them, and N is the maximum need of each process
  - What is the minimum number of resources R require to reach deadlock free condition ?
- Example
  - Input: P = 7, N = 2
  - Output: R >= 8

R >= P * (N – 1) + 1

Breaking Deadlock – Peterson’s solution

Globally defined

P2_wants_to_enter, P1_wants_to_enter, favored

Process 1

```
while(1){
   p1_wants_to_enter = True;
   favored = 2;
   while (p2_wants_to_enter =
True  && favored = 2);
   critical section
   p1_wants_to_enter = False;
}
```

Break the deadlock with a ‘favored’ flag

If the second process wants to enter, favor it.

favored is used to break the tie when both p1 and p2 want to enter the critical section

favored can take only two values: 1 or 2

Peterson's solution

- Deadlock broken because favored can only be 1 or 2
  - Only one process will enter the critical section

Process 1

```
while(1){
   p1_wants_to_enter = True;
   favored = 2;
   while (p2_wants_to_enter =
True && favored = 2);
   critical section
   p1_wants_to_enter = False;
}
```

Process 2

```
while(1){
   p2_wants_to_enter = True;
   favored = 1;
   while (p1_wants_to_enter =
True  && favored = 1);
   critical section
   p2_wants_to_enter = False;
}
```

2 process solution: Peterson's algorithm

- Ensure **two** threads never enter a critical section at the same time
  - Using 'flag' and 'turn' variables

```
void init() {
        // indicate intend to hold the lock with 'flag'
        flag[0] = flag[1] = 0;
        // whose turn is it ? (thread 0 or 1)
        turn = 0;
}
```

2 process solution: Peterson's algorithm

```
void lock () {
        // 'self' is the thread ID of caller
        flag[self] = 1;
        // make it other thread's turn
        turn = 1 - self;
        while ((flag [1 - self]) == 1 && (turn == 1 - self));
        // spin-wait while it's not your own
}
```

```
void unlock () {
        flag[self] = 0;
}
```

Bakery algorithm

- Synchronization between N > 2 processes
  - Processes numbered 0 to N – 1
  - num is an array N integers (initially 0)
  - Each entry corresponds to a process

```
lock (i) {
    num[i] = MAX(num[0], num[1], …, num[N - 1] + 1);
    for(p = 0; p < N; ++p)
      while(num[p] != 0 && num[p] < num[i])
}
```

Critical section

```
unlock (i) {num[i] = 0; }
```

1. num[i] = 0 means inactive
2. P means the priority

Should be atomic to ensure two processes don't get the same token.

Bakery algorithm

- How does Bakery algorithm work?

```
lock (i) {
  num[i] = MAX(num[0], num[1], …, num[N - 1] + 1);
        for(p = 0; p < N; ++p)
          while(num[p] != 0 && num[p] < num[i]) }
```

Critical section

```
unlock (i) {num[i] = 0; }
```

1. num[i] = 0 means inactive
2. P means the priority

T = 0

	P0	**P1**	**P2**	**P3**	**P4**
num	0	0	0	0	0

T = 1

	P0	**P1**	**P2**	**P3**	**P4**
num	0	1	2	3	4

Bakery algorithm

- What is the problem of the bellow implementation ?
- What about this situation ?
  - When P1 and P2 get the same number

```
lock (i) {
num[i] = MAX(num[0], num[1], …, num[N - 1] + 1);
          for(p = 0; p < N; ++p)
            while(num[p] != 0 && num[p] < num[i])}
```

Critical section

```
unlock (i) {num[i] = 0; }
```

1. num[i] = 0 means inactive
2. P means the priority

T = 0

	P0	P1	P2	P3	P4
num	0	0	0	0	0

T = 1

	P0	P1	P2	P3	P4
num	0	1	1	3	4

Bakery algorithm

- P1 and P2 will get into the critical section at the same time
  - That breaks the rule or mutual execution

1. num[i] = 0 means inactive
2. P means the priority

```
lock (i) {
num[i] = MAX(num[0], num[1], …, num[N - 1] + 1);
          for(p = 0; p < N; ++p)
            while(num[p] != 0 && num[p] < num[i])}
```

Critical section

```
unlock (i) {num[i] = 0; }
```

T = 0

	P0	**P1**	**P2**	**P3**	**P4**
num	0	0	0	0	0

T = 1

	P0	**P1**	**P2**	**P3**	**P4**
num	0	1	1	3	4

How to fix this problem ?

Bakery algorithm

(a, b) < (c, d) is equivalent to (a<c) or ((a == c) and (b < d))

- Adding choosing[i] to make MAX atomic
  - Initially all values of choosing[i] are false

```
lock (i) {
    choosing[i] = True;
    num[i] = MAX(num[0], num[1], …, num[N - 1] + 1);
    choosing[i] = False;
    for(p = 0; p < N; ++p) {
    // wait until process p receives its number
    while(choosing[p]);
    while(num[p] != 0 && (num[p],p) < (num[i],i)) }
```

If there are two processes with the same num value, favor the process with the smaller id (i)

Critical section

```
unlock (i) {num[i] = 0; }
```

Bakery algorithm

(a, b) < (c, d) is equivalent to (a<c) or ((a == c) and (b < d))

- How does Bakery algorithm work ?

```
lock (i) {
    choosing[i] = True;
    num[i] = MAX(num[0], num[1], …, num[N - 1] + 1);
    choosing[i] = False;
    for(p = 0; p < N; ++p) {
     // wait until process p receives its number
    while(choosing[p]);
    while(num[p] != 0 && (num[p],p) < (num[i],i)) }
```

Critical section

```
unlock (i) {num[i] = 0; }
```

T = 0

	P0	**P1**	**P2**	**P3**	**P4**
num	0	0	0	0	0

T = 1

	P0	**P1**	**P2**	**P3**	**P4**
num	0	1	2	3	4

Multiprocessor memory models

- Uniprocessor memory is simple
  - Every load from a location retrieves the last value stored to that location
  - All processes / threads see the same view of memory
- The straightforward multiprocessor memory model – sequential consistency
  - All operations executed in some sequential order
  - Each thread's operations happen in program order

Memory consistency models

- Why memory consistency models matter ?
  - **Multiprocessors reorder memory** operations in unintuitive ways
  - This behavior affects the **performance of programs**
  - These models are hidden by programmers – hard to debug
  - But kernel developers see it all the time
- What is consistency model ?
  - Consistency models deal with how multiple threads see the world
  - Define the allowed orderings of multiple threads on a multiprocessor

Multithread programs

What is our expected output print ?

Initially, A = 0, B = 0

Thread 1

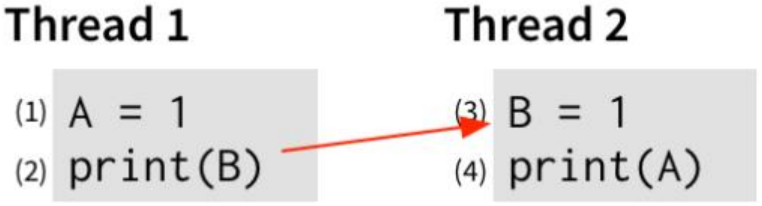
(1) A = 1
(2) print (B)

Thread 2

(3) B = 1
(4) print (A)

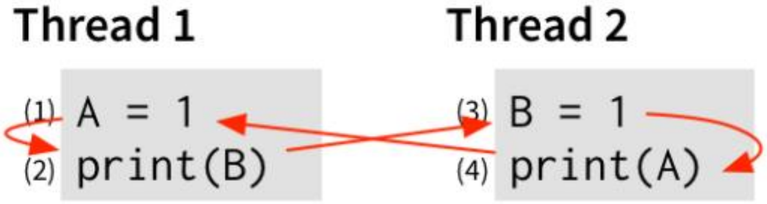
- What is the value of A and B ?
  - The order the events decides outputs
  - (1) -> (2) -> (3) -> (4):
    - The first thread runs event (1) (2) before the second thread -> print B = 0, A = 1
  - (3) -> (4) -> (1) -> (2):
    - The second thread runs event (3) (4) before the first thread -> print B = 1, A = 0
  - (1) -> (3) -> (2) -> (4)
    - The first instruction in each thread runs before the second inst. -> print B = 1, A = 1
  - (1) -> (3) -> (4) -> (2)
    - The second threads runs the second instructions before the first thread -> print B = 1, A = 1

Multithread programs



- This program should print ’11’
  - Each thread’s events should happen in order
  - (1) before (2), (3) before (4)
- The “happens-before” graph
  - Shows the order where events must execute to get a desired outcome
  - If there’s a cycle in the graph, an outcome is impossible – an event must happen before itself

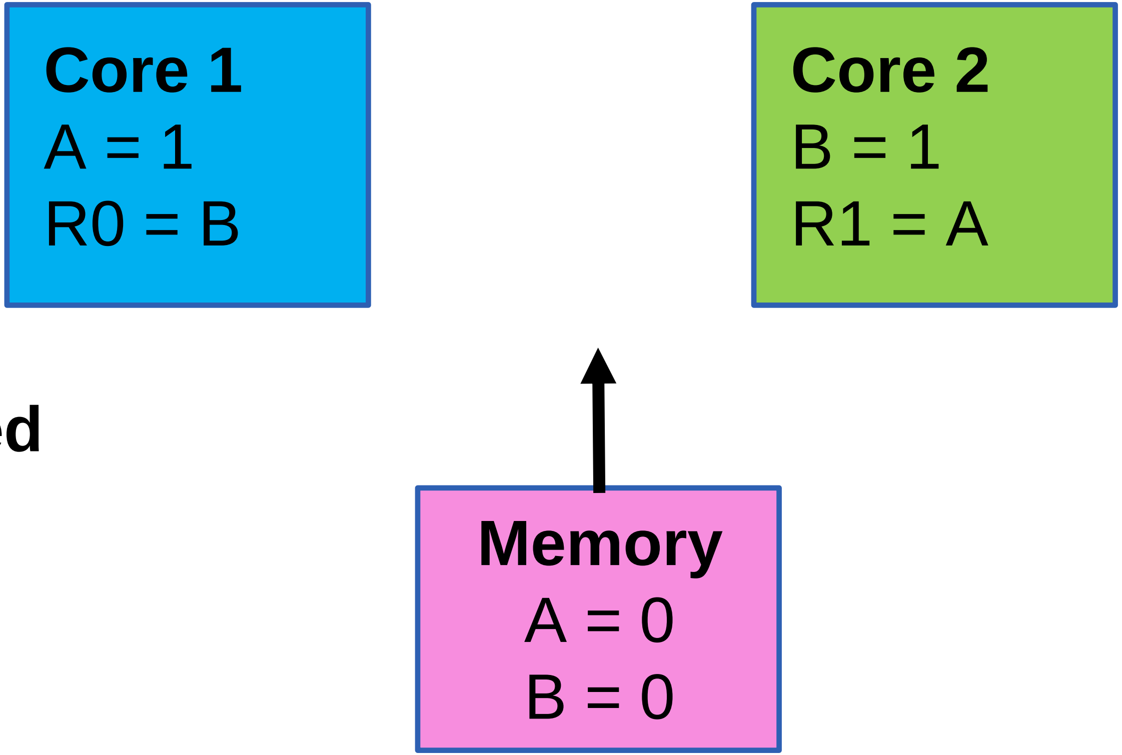
An edge from operation x to operation y says that x must happen before y



Sequential consistency

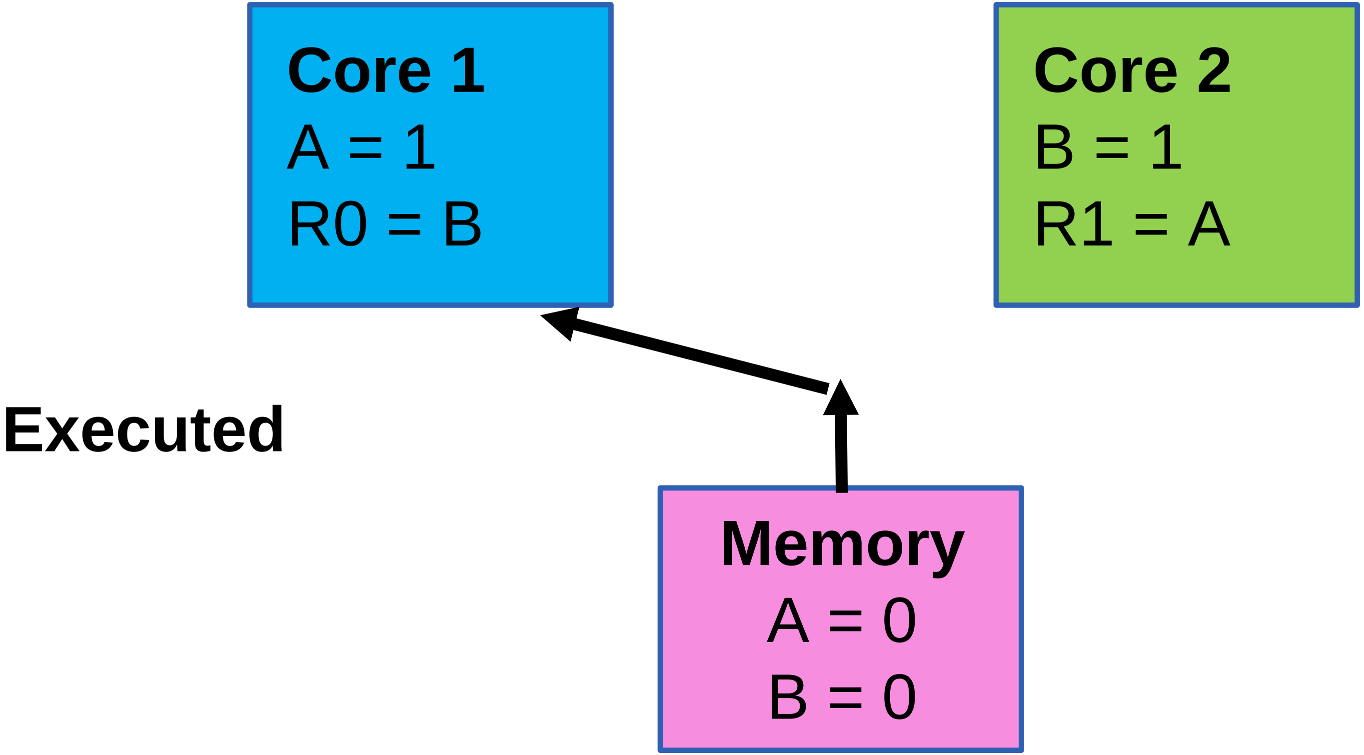
- **The scenario**
  - **Multiple threads** running in parallel are manipulating **a single main memory**, so everything must happen in order
- **Two invariants**
  - All operations executed in **some** sequential order
  - Each thread’s operations happen in program order
- Says nothing about which order all operations happen in
  - Any interleaving of threads is allowed
- From Leslie Lamport in 1979

Sequential consistency



Executed

Sequential consistency



Sequential consistency

Core 1
A = 1
R0 = B

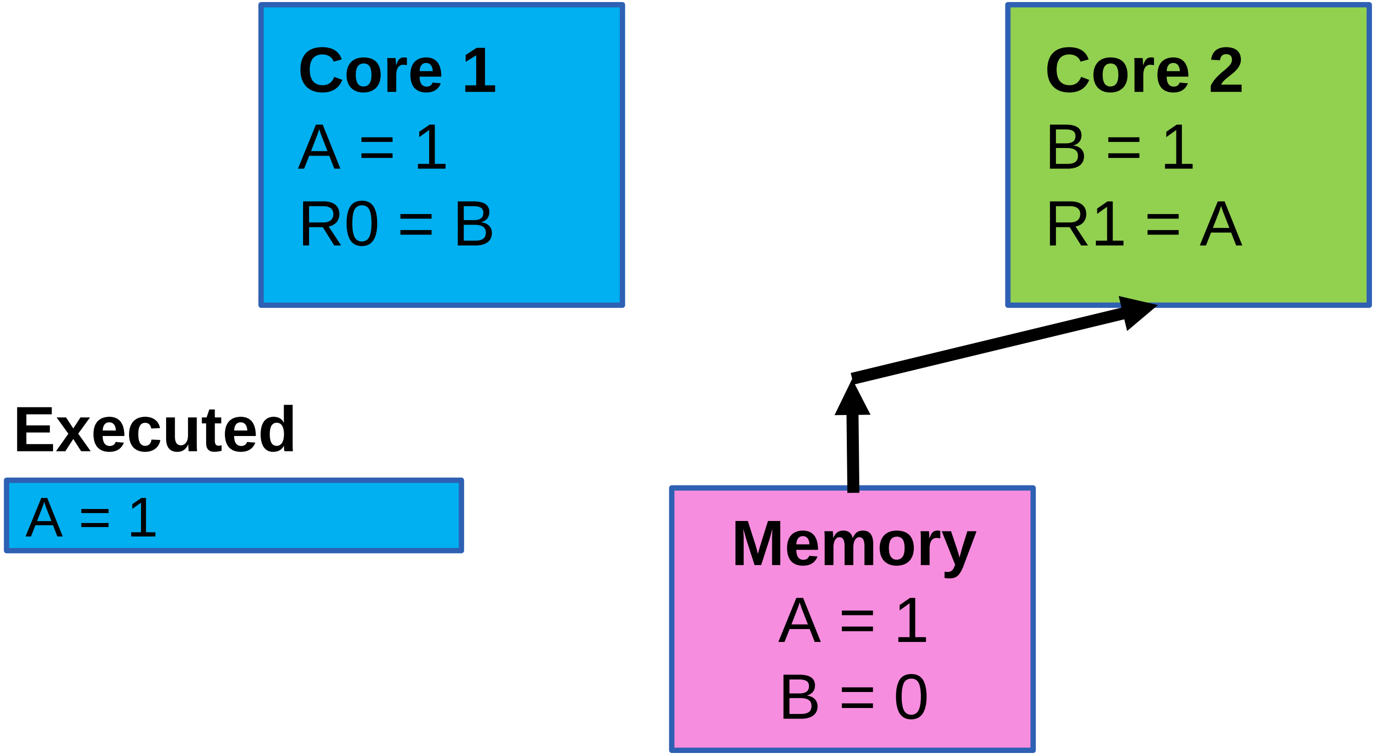
Core 2
B = 1
R1 = A

Executed

A = 1

Memory
A = 1
B = 0

Sequential consistency



Sequential consistency

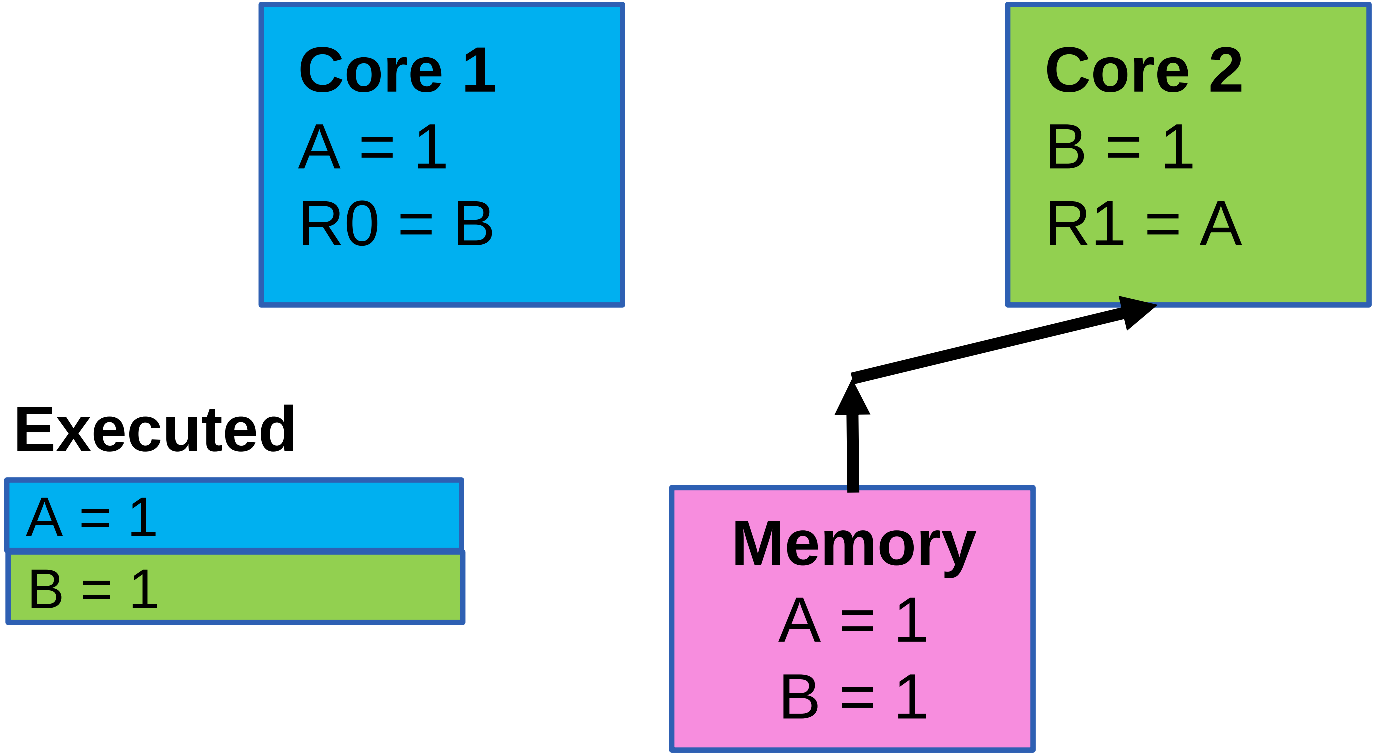
Core 1
A = 1
R0 = B

Core 2
B = 1
R1 = A

Executed
A = 1
B = 1

Memory
A = 1
B = 1

Sequential consistency



Sequential consistency

Core 1
A = 1
R0 = B

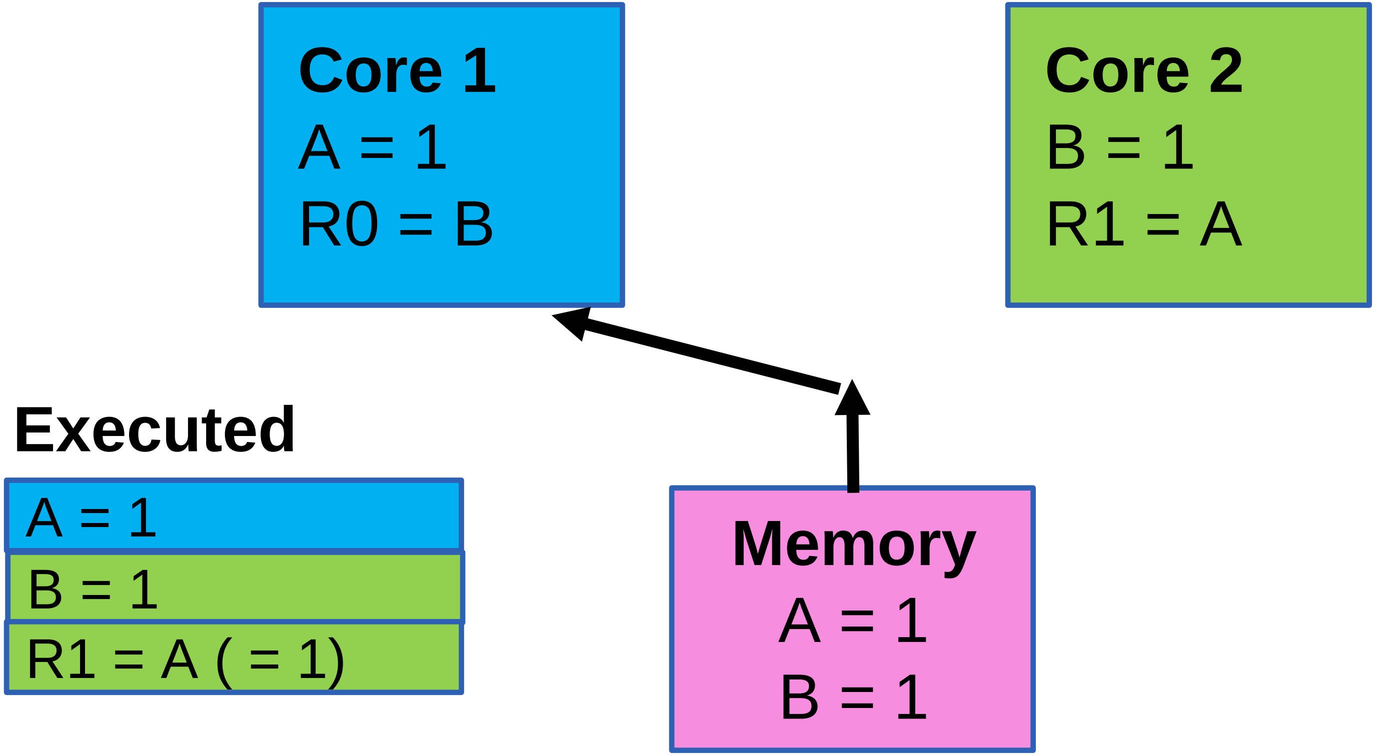
Core 2
B = 1
R1 = A

Executed

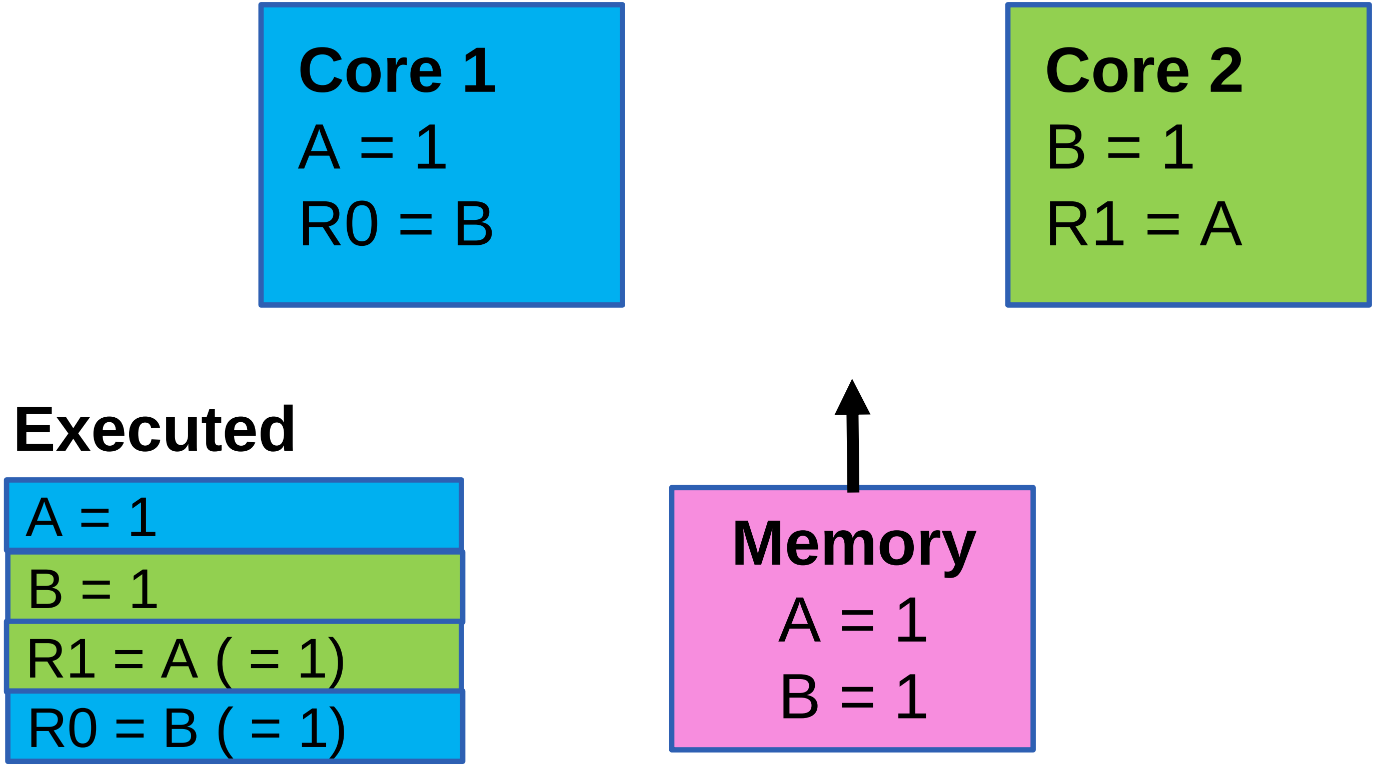
A = 1
B = 1
R1 = A (= 1)

Memory
A = 1
B = 1

Sequential consistency



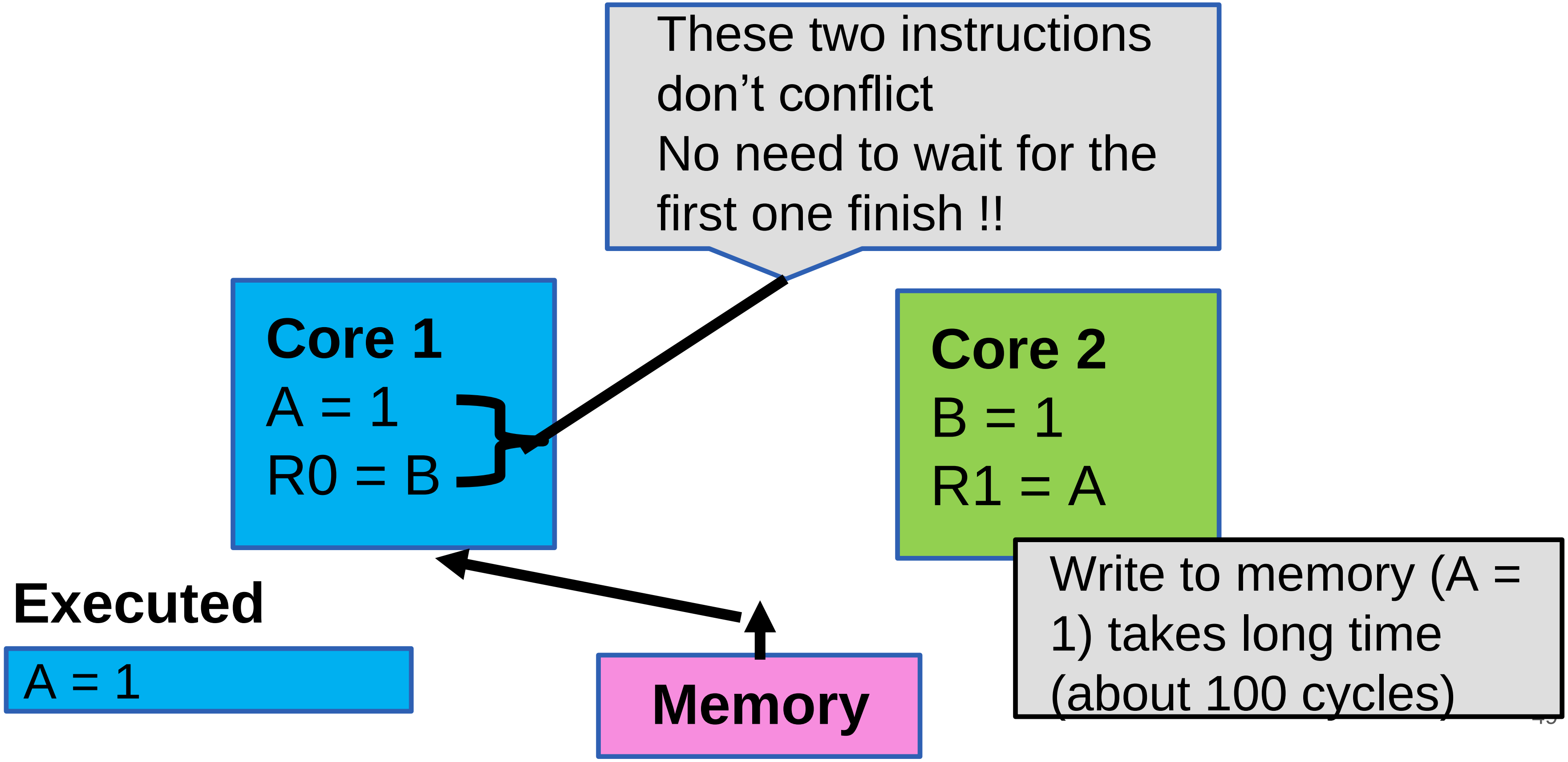
Sequential consistency



Sequential consistency

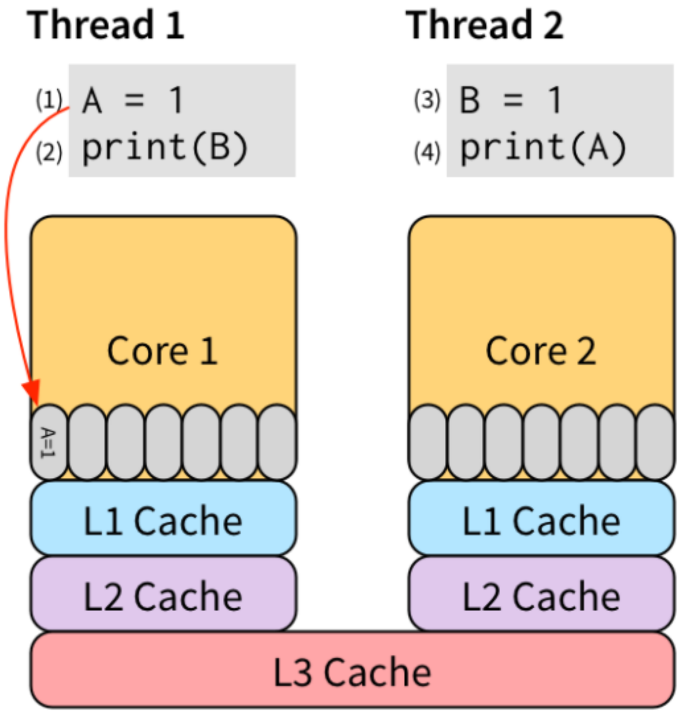
- Why sequential consistency (SC) ?
  - Agrees with programmer intuition
- Why **not** sequential consistency ?
  - Horribly slow to guarantee in hardware
  - The “switch” model is overly conservative

The problem of sequential consistency



Relaxed memory models – Total store ordering

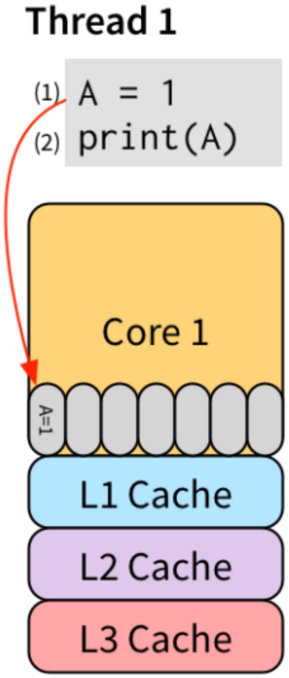
- **Total store ordering (TSO)**
  - **Store writes in a local buffer** and then proceed to next instruction immediately
  - The cache will pull writes out of the write buffer when it's ready
  - The (2) starts immediately after putting (1) into the store buffer, rather than waiting for it to reach the L2 cache
  - The store buffer hides the **write latency**



https://www.cs.utexas.edu/~bornholt/post/memory-models.html

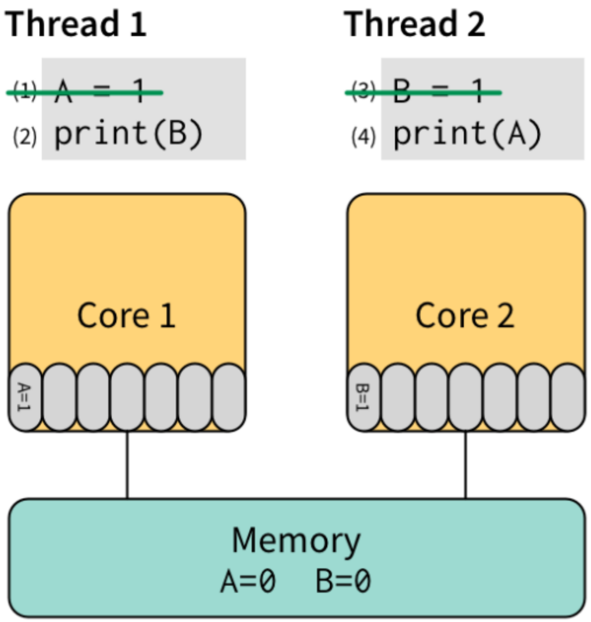
Total store ordering (TSO)

- Store buffering is nice because it preserves single-threaded behavior
  - Read (2) inspect the store buffer directly
  - If the store buffer contains a write to the location it's reading, and use that value instead
  - Otherwise, check the memory



https://www.cs.utexas.edu/~bornholt/post/memory-models.html https://www.cs.utexas.edu/~bornholt/post/memory-models.htm

More total store ordering



- First, executing (1) then (3)
  - Both of them place their data in the store buffer rather than sending back to memory
- Next, executing (2) on core 1
  - no value of B in the store buffer
  - So, it reads B from memory and get value 0
- Finally, executing (4) on core 2
  - No value in core 2's store buffer, it reads from memory and gets the value 0
- Under TSO, this program print 00 -> No !

Memory fences

- The x86 "mfence" instruction
  - Used to against programs broken by TSO
  - Loads and stores cannot be moved before or after the mfence instruction
  - It is like to flush the store buffer and prevent the pipeline from reordering around the fence
- mfence is not cheap
  - See "sfence" and "lfence" which are weaker (and faster) than mfence

Thread 1

(1) A = 1
(2) mfence
(3) print (B)

Thread 2

(1) B = 1
(2) mfence
(3) print (A)

Conclusion

- **Threads**
  - Share the process's states such as address space
- **Race condition**
  - Multiple threads attempt to change the shared data concurrently
- **Peterson's algorithm**
  - Ensure two processes not enter the critical section at the same time
- **Bakery algorithm**
  - Synchronization over N > 2 processes
- **Memory consistency**