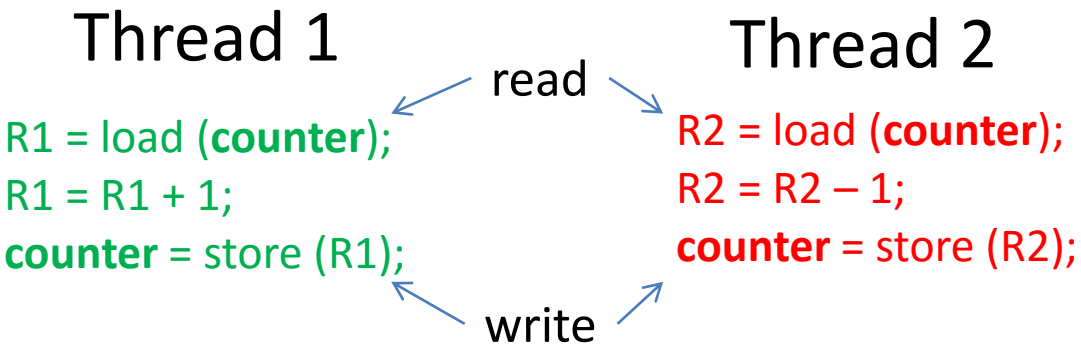
Recap

- Race condition
- Critical section

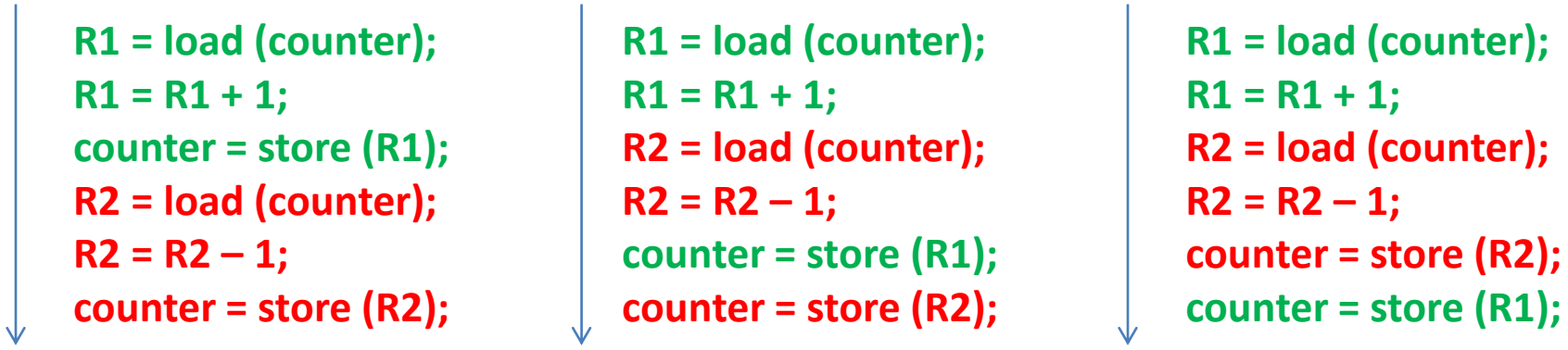
Recap: Race Condition

- A situation when two or more threads **read and write** shared data at the same time
- Correctness depends on the execution order



Recap: Race Condition

Initial condition: *counter = 5*



counter = 5

counter = 4

counter = 6

Recap: Critical Section

- Code sections of potential race conditions

Thread 1	Thread 2
Do something	Do something
..	..
R1 = load (**counter**);	R2 = load (**counter**);
R1 = R1 + 1;	R2 = R2 – 1;
counter = store (R1);	**counter** = store (R2);
...	..
Do something	Do something

Critical sections

Recap: Flag Doesn't Work

```
// wait
while (in_cs)
    ;
// enter critical section
in_cs = true;

Do something

// exit critical section
in_cs = false;
```

T1

```
while(in_cs);


in_cs = true;
//enter
```

T2

```
while(in_cs);
in_cs = true;

//enter
```

- Mutual exclusion is not guaranteed

Roadmap

- Solutions for mutual exclusion
  - Peterson's algorithm (Software)
  - Synchronization instructions (Hardware)

Peterson's Solution

- Software solution (no h/w support)
- Two process solution
  - Multi-process extension exists
- The two processes share two variables:
  - int turn;
    - The variable turn indicates whose turn it is to enter the critical section
  - Boolean flag[2]
    - The flag array is used to indicate if a process is ready to enter the critical section.

Peterson's Solution

Thread 1

```
do {
   flag[0] = TRUE;
   turn = 1;
   while (flag[1]  &&  turn==1)
      ;
   // critical section

   flag[0] = FALSE;

   // remainder section
} while (TRUE)
```

Thread 2

```
do {
   flag[1] = TRUE;
   turn = 0;
   while (flag[0]  &&  turn==0)
      ;
   // critical section

   flag[1] = FALSE;

   // remainder section
} while (TRUE)
```

- Solution meets all three requirements
  - Mutual exclusion: P0 and P1 cannot be in the critical section at the same time
  - Progress: if P0 does not want to enter critical region, P1 does no waiting
  - Bounded waiting: process waits for at most one turn

Peterson's Solution

- Only supports two processes
  - generalizing for more than two processes has been achieved, but not very efficient
- Assumes that the LOAD and STORE instructions are atomic
- Assumes that memory accesses are not reordered
  - your compiler re-orders instructions (gcc –O2, -O3, …)
  - your processor re-orders instructions (memory consistency models)

Reordering by the CPU

Initially X = Y = 0

Thread 0	Thread 1
X = 1	Y = 1
R1 = Y	R2 = X

Thread 0	Thread 1
R1 = Y	
	R2 = X
X = 1	
	Y = 1

- Possible values of R1 and R2?
  - 0,1
  - 1,0
  - 1,1
  - **0,0 ← possible on PC**

Lock

- General solution
  - Protect critical section via a lock
  - Acquire on enter, release on exit

```
do {
    acquire lock;

        critical section

    release lock;

        remainder section

} while(TRUE);
```

How to Implement a Lock?

- Unicore processor
  - No true concurrency

    one thread at a time
  - Threads are *interrupted* by the OS
    - scheduling events: timer interrupt, device interrupts
- Disabling interrupt
  - Threads can't be interrupted

```
do {

    disable interrupts;
        critical section
    enable interrupts;

        remainder section
} while(TRUE);
```

How to Implement a Lock?

- Multicore processor
  - True concurrency
    - More than one active threads sharing memory
  - Disabling interrupts don't solve the problem
    - More than one threads are executing at a time
- Hardware support
  - Synchronization instructions
    - **Atomic** *test&set* instruction
    - **Atomic** *compare&swap* instruction
- What do we mean by **atomic**?
  - All or nothing

TestAndSet Instruction

- Pseudo code

```
boolean TestAndSet (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

Mutual Exclusion using *TestAndSet*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex)
{
    *mutex = 0;
}

void lock (int *mutex)
{
    while(TestAndSet(mutex))
        ;
}

void unlock (int *mutex)
{
    *mutex = 0;
}
```

CAS (Compare & Swap) Instruction

- Pseudo code

```
int CAS(int *value, int oldval, int newval)
{
        int temp = *value;
        if (*value == oldval)
           *value = newval;
        return temp;
 }
```

Mutual Exclusion using *CAS*

```
int mutex;
init_lock (&mutex);

do {

    lock (&mutex);
        critical section
    unlock (&mutex);

        remainder section
} while(TRUE);
```

```
void init_lock (int *mutex) {
    *mutex = 0;
}

void lock (int *mutex) {
    while(CAS(&mutex, 0, 1) != 0);
}

void unlock (int *mutex) {
    *mutex = 0;
}
```