Memory Reordering

zhuyie
zhuyie@gmail.com

Agenda

- Background
- Example 1: Read two 0s
- Example 2: Peterson lock
- Conclusion

Background

What is memory reordering?

- Memory reordering refers to the reordering of instructions when the **execution order** of the instructions is **inconsistent** with the order written in the code (**source code order**).
- It usually occurs in the following two procedures:
  - Compiler reordering (Compile time)
  - CPU reordering (Run time)
- Motivation: to improve the **speed** of running code.

Basic principle

- Memory reordering **shall not modify the behavior of a single-threaded program**.
- Therefore, memory reordering does not affect single-threaded programs.
- However, if it is **multi-threaded**, these optimizations can bring problems to the semantics of the program.

Different kinds of orderings

Source code order	The order in which the memory operations are specified in the source code.
Program order	The order in which the memory operations are specified in the machine code. May differ from the source code order due to compiler optimization.
Execution order	The order in which the individual memory-reference instructions are executed on a given CPU. May differ from the program order due to optimizations based on the specific CPU' hardware memory model.
Perceived order	The order in which a CPU perceives its and other CPUs' memory operations. May differ from the execution order due to caching, interconnect, and memory-system optimizations defined by the hardware memory model.

Compiler reordering

- The compiler may adopt a series of compilation **optimizations** during the compilation process.
    - register allocation
    - loop-invariant code motion
    - dead store elimination
    - ...
- The **program order** (specified in the machine code) may differ from the **source code order**.

Compiler reordering

- What is the output?

P1	P2
`A = 100;`	`B = true;`
`B = true;`	`A = 100;`
`If (B)`	`If (B)`
`    Print A;`	`    Print A;`

- From the single-threaded perspective, the store-to-A and store-to-B is **irrelevant**, so the compiler is free to change the ordering.
- How about multi-threaded execution?

T1	T2
`void Init()`	`void Use()`
`{`	`{`
`    B = true;`	`    If (B)`
`    A = 100;`	`        Print A;`
`}`	`}`

Compiler reordering

- What is the output?

Original	After compiler optimization
`X = 0`	~~`X = 0`~~ `// dead store elimination`
`for i in range(100):`	`X = 1 // loop-invariant code motion`
`    X = 1`	`for i in range(100):`
`    print X`	`    print X`

- But now suppose there’s another thread running in **parallel** with our loop, and it performs a single write to X (X = 0).
  - The first program can print strings like 11101111…
  - And the second program will print strings like 11100000…

Compiler barrier

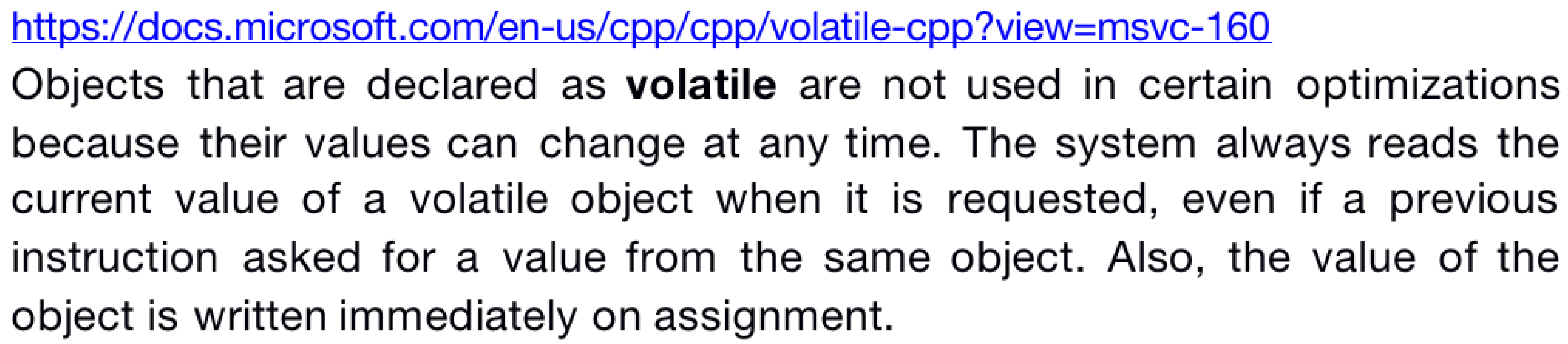
- The minimalist approach to preventing compiler reordering is by using a special directive known as a **compiler barrier**.

MSVC	**gcc/clang**
`#include <intrin.h>`	
`void Init()`	`void Init()`
`{`	`{`
`    A = 100;`	`    A = 100;`
`    _ReadWriteBarrier();`	`    asm volatile("" ::: "memory");`
`    B = true;`	`    B = true;`
`}`	`}`

- Just a directive, no machine code will be generated.
- Every function containing a compiler barrier must act as a compiler barrier itself, even when the function is inlined.

The volatile keyword

- Another way to implement the compiler barrier is by using the **volatile** keyword in the C/C++ language.

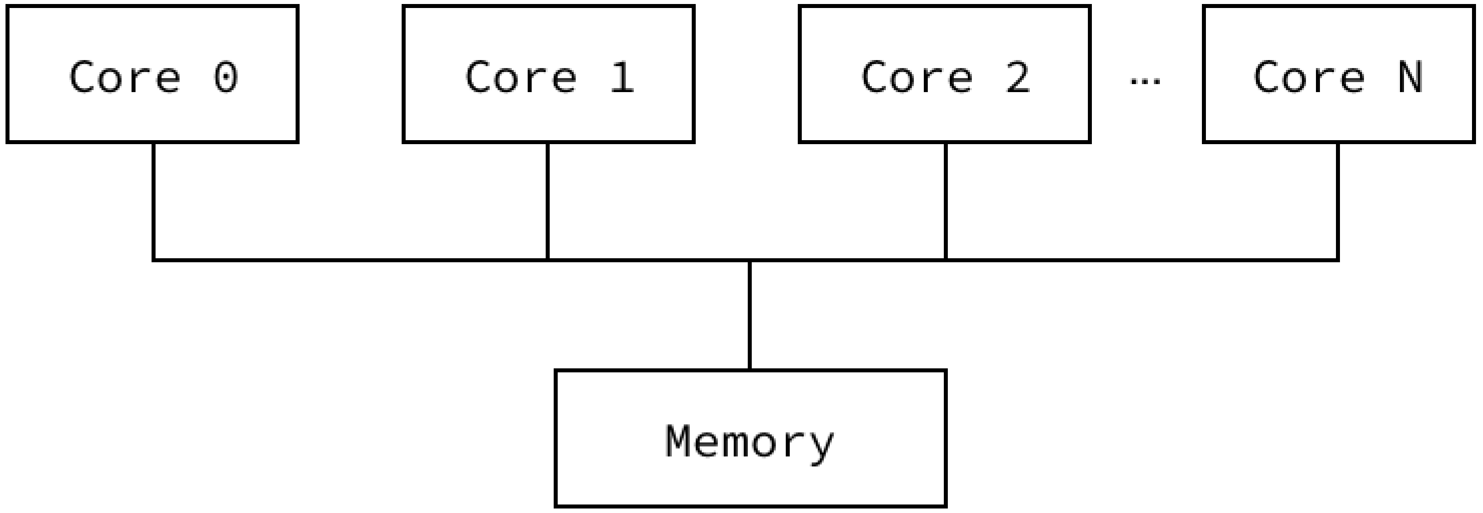


https://docs.microsoft.com/en-us/cpp/cpp/volatile-cpp?view=msvc-160

Objects that are declared as **volatile** are not used in certain optimizations because their values can change at any time. The system always reads the current value of a volatile object when it is requested, even if a previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.

- A volatile variable can no longer be cached by registers.
- Not recommended for Linux kernels.

A multi-core shared memory computer



Coherence

- Consider this example: if both (1) and (2) starts to run at the same time, what is the output of (3) and (4)?

Thread 1	Thread 2	Thread 3	Thread 4
(1) `A = 1`	(2) `A = 2`		
		(3) `Print(A)`	(4) `Print(A)`

- The single main memory guarantees that there will always be a **"winner"**: a single last write to each variable.
- We call this guarantee **coherence**, and it says that all writes to the **same location** are seen in the **same order** by every thread.
- It doesn't prescribe the actual order.

Consistency

- How about the ordering of operations to **multiple locations**?
- A **consistency model** defines the **allowed** behavior of loads and stores to different addresses in a parallel system.
- For example, a consistency model can define that a process is not allowed to issue an operation until all previously issued operations are completed. Different consistency models enforce different conditions.

Memory operation ordering

- A program defines a sequence of loads and stores.
    - the **"program order"**
- Four types of memory operation orderings:
    - W→R: write to X must **commit** before subsequent read from Y
    - R→R: read from X must commit before subsequent read from Y
    - R→W: read to X must commit before subsequent write to Y
    - W→W: write to X must commit before subsequent write to Y

Multi-threaded execution

- Consider this example: (Initially A=B=0)

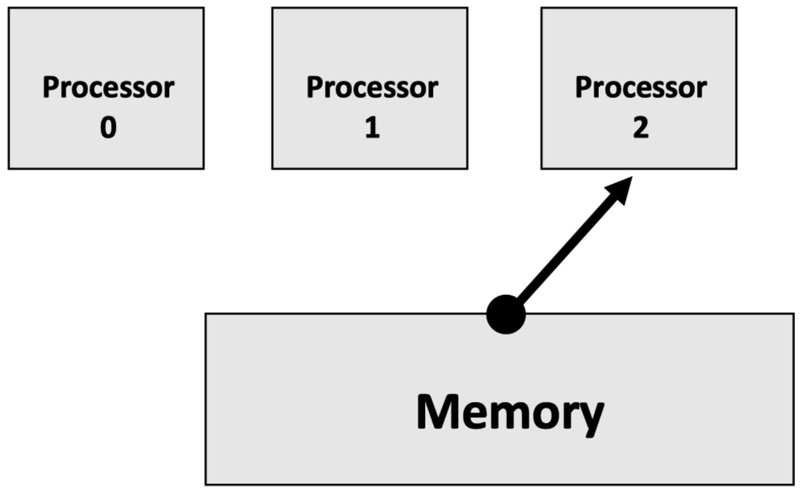
Thread 1	**Thread 2**
(1) `A = 1`	(3) `B = 1`
(2) `Print(B)`	(4) `Print(A)`

- Possible orders:
  - 1→2→3→4: prints "01"
  - 3→4→1→2: prints "01"
  - 1→3→2→4: prints "11"
  - and a few others that also prints "11"

What should programmers expect

- **Sequential Consistency**
  - Lamport 1976 (Turing Award 2013)
  - Each thread's operations happen in **program order**.
  - All operations executed in **some** sequential order.
- A sequentially consistent memory system maintains all four memory operation orderings (W→R, R→R, R→W, W→W)

Sequential consistency (switch metaphor)



- All processors issue loads and stores in program order.
- Memory chooses a processor, performs a memory operation to completion, then chooses another processor, …
- The problem with this model is that it’s terribly **slow**.

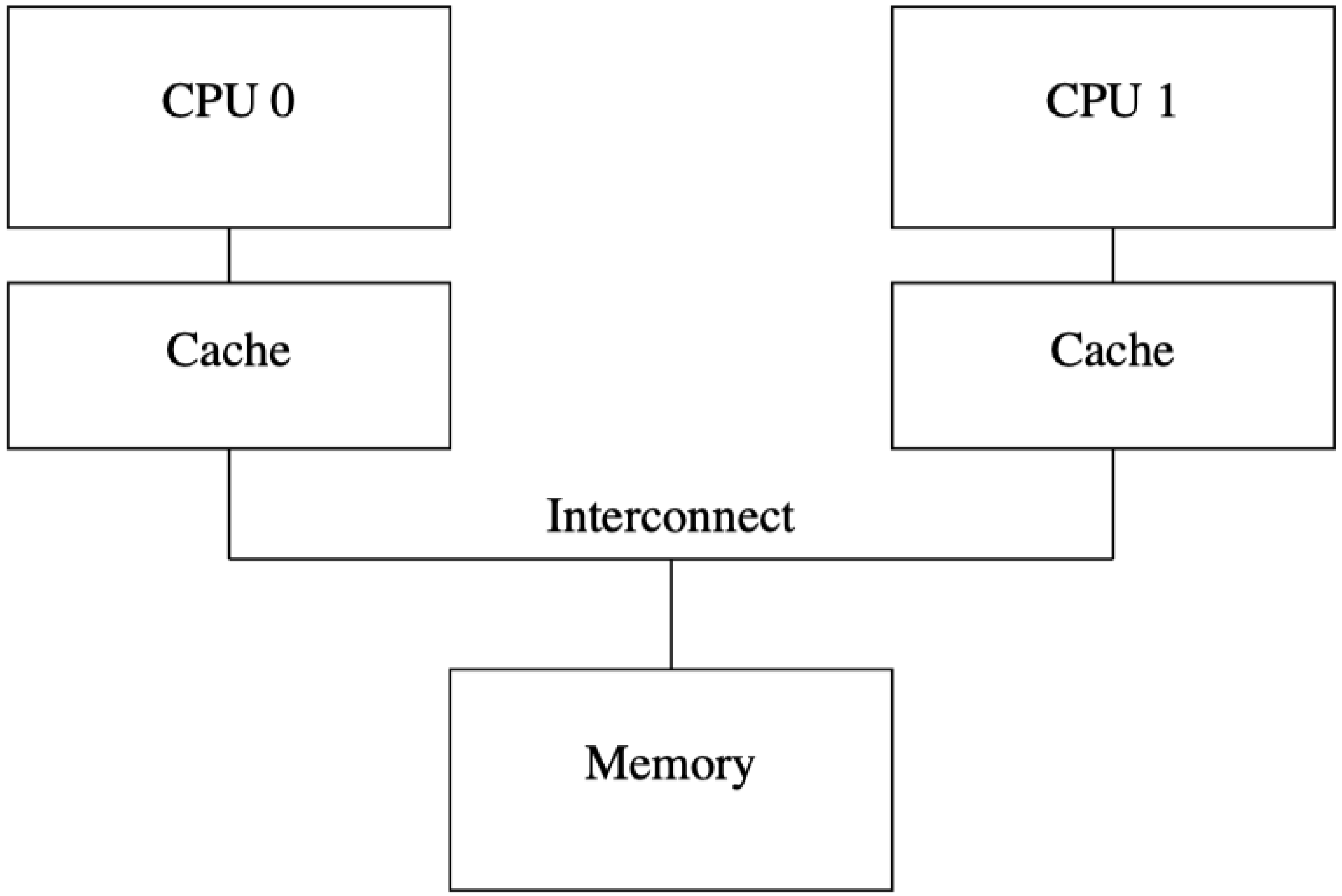
Things that shouldn't happen

- Can this program prints **"00"**?

Thread 1	**Thread 2**
(1) `A = 1`	(3) `B = 1`
(2) `Print(B)`	(4) `Print(A)`

  - For line (2) to print "0", (2) should happen before (3)
  - For line (4) to print "0", (4) should happen before (1)
- Contradiction:
  - (1) -> (2)
  - (2) -> (3)
  - (3) -> (4) **=>** (1) -> (4)

A simple computer cache structure



Cache-coherency protocols

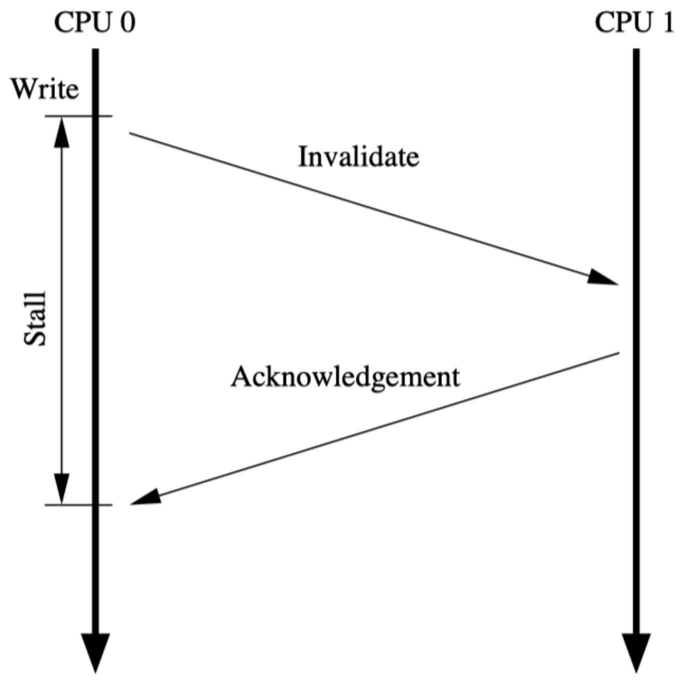
- Memory variables may have **local copy** in each CPU's cache.
- Clearly, much care must be taken to ensure that all CPUs maintain a **coherent** view of the data.
- Cache-coherency **protocols** manage cache-line states so as to prevent inconsistent or lost data.
- The most common used protocol: **MESI**

MESI cache-coherence protocol

- MESI stands for **“modified”**, **“exclusive”**, **“shared”**, and **“invalid”**, the four **states** a given cache line can take on using this protocol.
- The protocol provides **messages** that coordinate the transitions of cache line states:
  - Read, Read-Response, Invalidate, Invalidate-Acknowledge, Read-Invalidate, Writeback
- Each cache **snooping** the bus, read & process the messages.
- In another word, a **message-based state machine**.

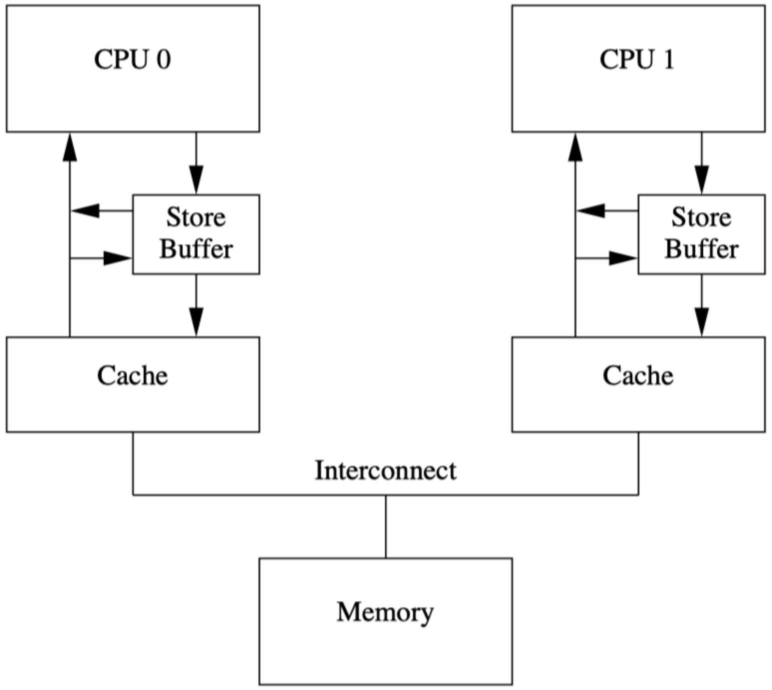
Stores result in unnecessary stalls

- Sometimes, the cache's performance for the first write to a given cacheline is quite **poor**. For example, a write by CPU 0 to a cacheline held in CPU 1’s cache:



Stores result in unnecessary stalls

- From CPU 0's perspective, the waiting is unnecessary if **later read can get the newly written value**.
- The CPU will usually use **store buffer** to resolve this issue:



Store buffers change memory behavior

- `A = 1` can simply record in store buffer and continue executing.
- `r1 = B` may complete very fast.
- The write to `A` then seen by other CPUs.
- Globally it looks like: `r1 = B` move ahead of `A = 1`.
- We need a **weaker** memory model.

Total Store Ordering (TSO)

- Processor P can read B **before** its write to A is **seen** by all processors.
- Reads by other processors cannot return new value of A until the write to A is observed by all processors.
- In TSO, only **W→R order is relaxed**. The W→W constraint still exists. Writes by the same thread are not reordered (they occur in program order)
- **x86** uses an incompletely specified form of TSO.

Total Store Ordering (TSO)

- What is the output? (Initially A = B = 0)

T1	T2
`A = 1;     // store to A`	`B = 1;     // store to B`
`r1 = B;    // load B`	`r2 = A;    // load A`
`Print r1;`	`Print r2;`

- If there was a simultaneous store-load **reorder**, the program may prints "00".

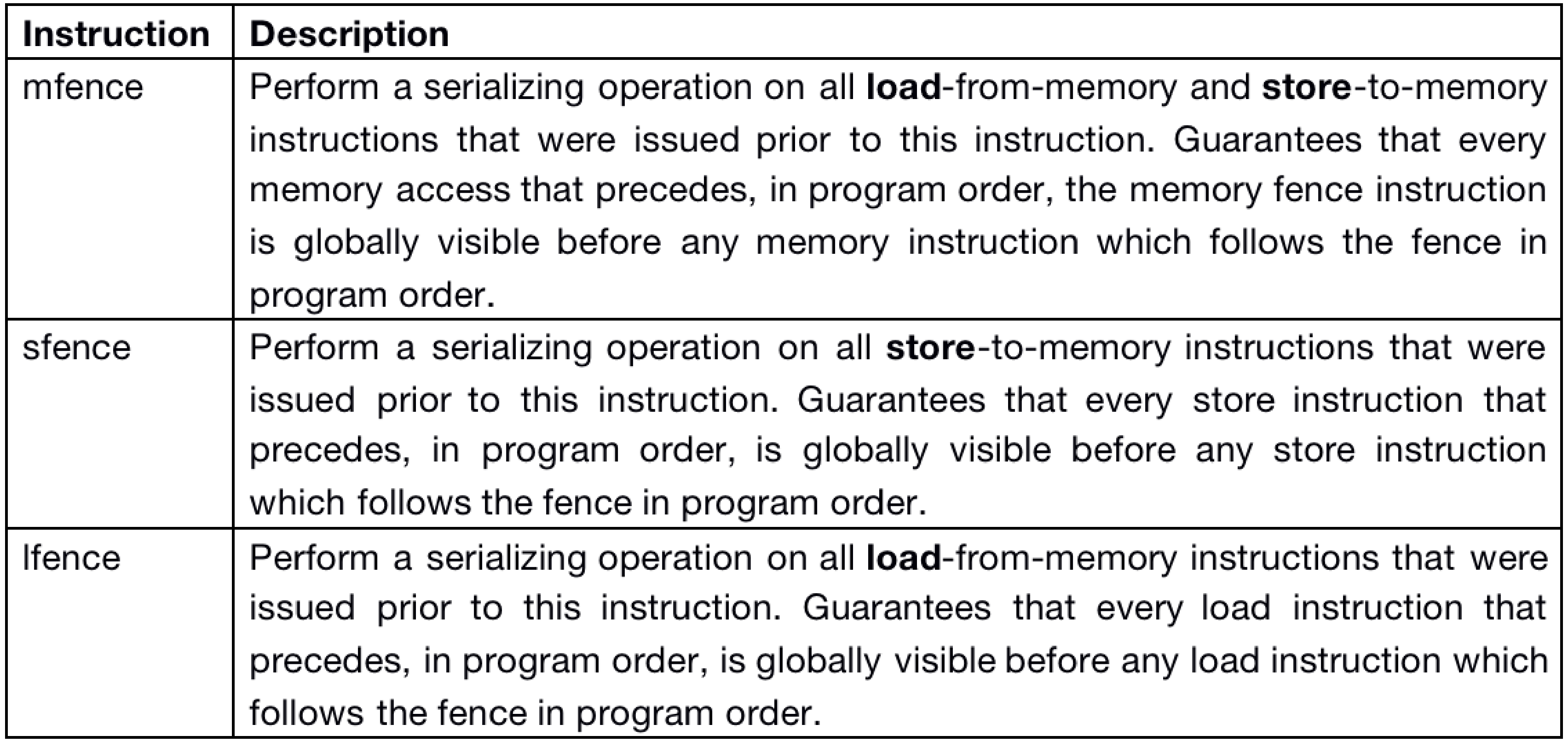
Memory barrier

- The CPUs have no idea which variables are related, let alone how they might be related. Therefore, the hardware designers provide **memory-barrier instructions** to allow the software to **tell** the CPU about such relations.
- The memory barrier can cause the CPU to **flush** its store buffer. (simply stall until the store buffer was empty before proceeding)
- After insert some memory barrier instructions, the program works like in **SC**:

T1	**T2**
`A = 1;     // store to A`	`B = 1;     // store to B`
`MEMORY_BARRIER();`	`MEMORY_BARRIER();`
`r1 = B;    // load B`	`r2 = A;    // load A`
`Print r1;`	`Print r2;`

Memory barrier

- A hardware barrier **also** act as a compiler barrier.
- x86 provides memory barriers in the form of fence instructions:



Instruction	**Description**
mfence	Perform a serializing operation on all **load**-from-memory and **store**-to-memory instructions that were issued prior to this instruction. Guarantees that every memory access that precedes, in program order, the memory fence instruction is globally visible before any memory instruction which follows the fence in program order.
sfence	Perform a serializing operation on all **store**-to-memory instructions that were issued prior to this instruction. Guarantees that every store instruction that precedes, in program order, is globally visible before any store instruction which follows the fence in program order.
lfence	Perform a serializing operation on all **load**-from-memory instructions that were issued prior to this instruction. Guarantees that every load instruction that precedes, in program order, is globally visible before any load instruction which follows the fence in program order.

Partial Store Ordering (PSO)

- Four types of memory operation orderings:

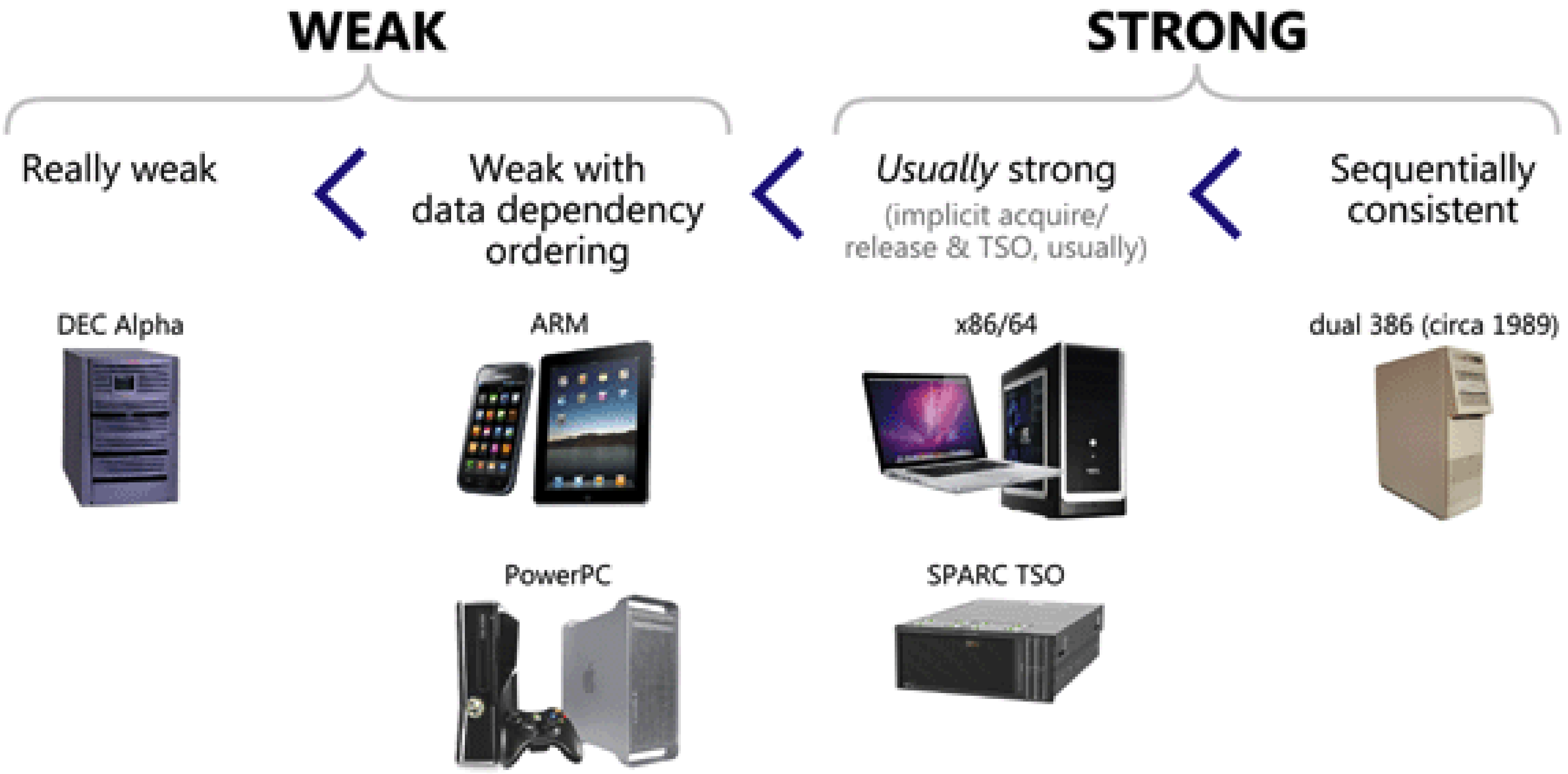
```
~~W→R: write to X must commit before subsequent read from Y~~
R→R: read from X must commit before subsequent read from Y
R→W: read to X must commit before subsequent write to Y
~~W→W: write to X must commit before subsequent write to Y~~
```

- What is the output? (Initially A = flag = 0)

Thread 1	**Thread 2**

`A = 1;`
`flag = 1;`
	`while (flag == 0) { };`
	`Print A;`

Weak vs. Strong Memory Models



Coherence vs. Consistency

- **Memory coherence** defines requirements for the observed behavior of reads and writes to the **same** memory location.
  - All processors must agree on the order of reads/writes to X.
- **Memory consistency** defines the behavior of reads and writes to **different** locations (as observed by other processors).
  - Coherence only guarantees that writes to address X **will** eventually propagate to other processors.
  - Consistency deals with **when** writes to X propagate to other processors, relative to reads and writes to other addresses.

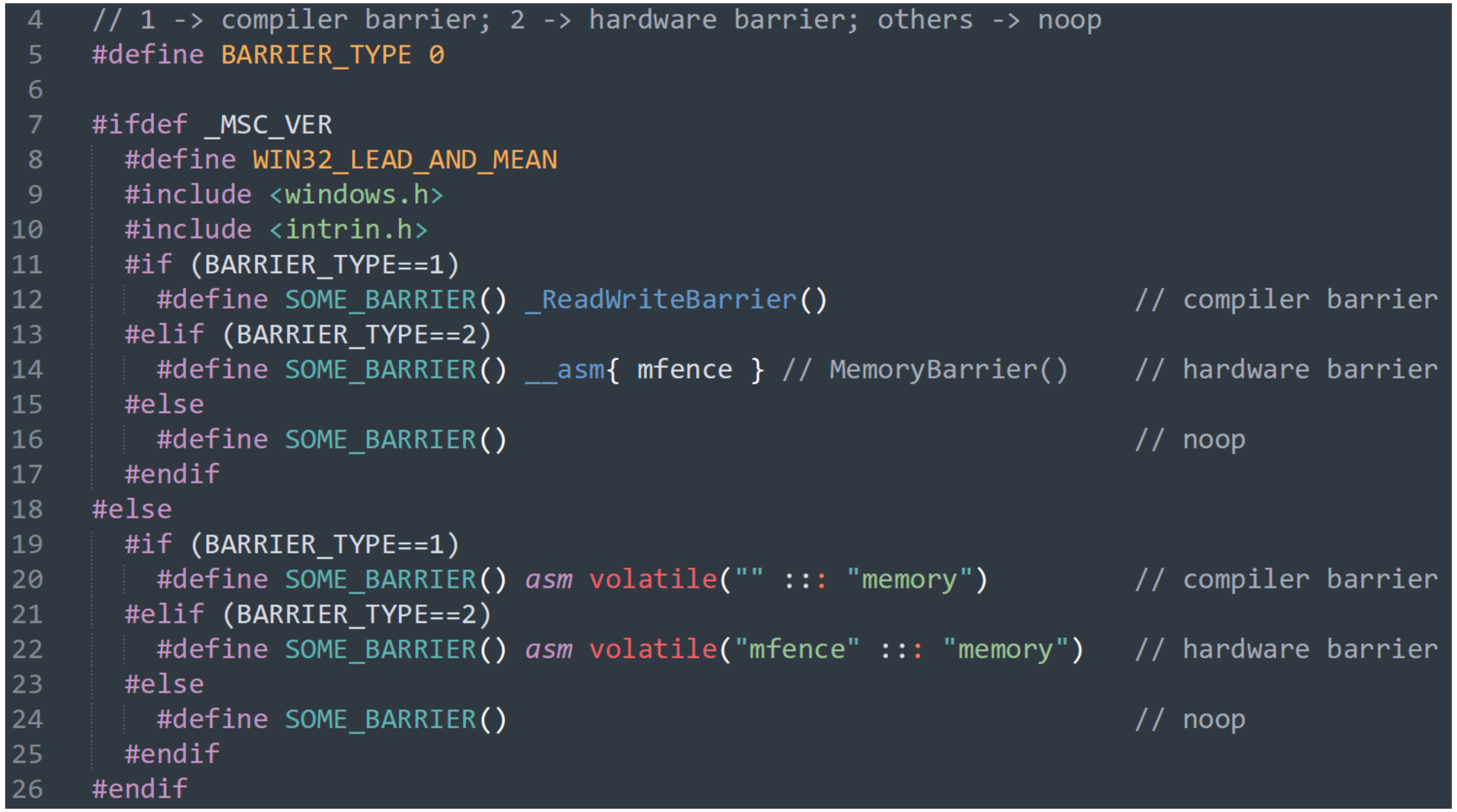
Example 1: Read two 0s

x86 store-load reorder

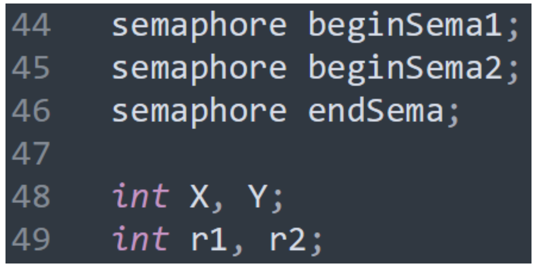
- x86 uses an incompletely specified form of TSO.
- In TSO, the **W→R order is relaxed**.
- Let's test this example on x86:

T1	T2
`A = 1;    // store to A`	`B = 1;    // store to B`
`r1 = B;   // load B`	`r2 = A;   // load A`
`Print r1;`	`Print r2;`

SOME_BARRIER() and globals

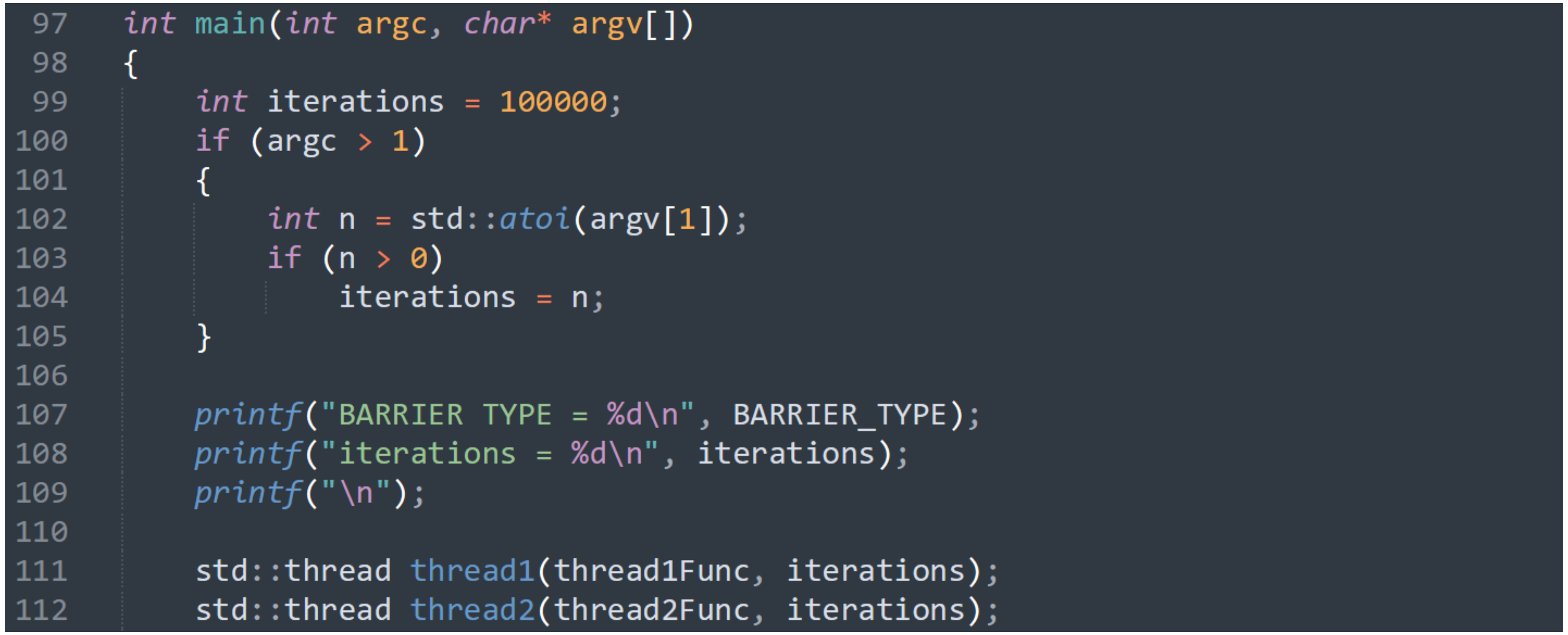


```
// 1 -> compiler barrier; 2 -> hardware barrier; others -> noop
#define BARRIER_TYPE 0

#ifdef _MSC_VER
  #define WIN32_LEAD_AND_MEAN
  #include <windows.h>
  #include <intrin.h>
  #if (BARRIER_TYPE==1)
    #define SOME_BARRIER() _ReadWriteBarrier()                  // compiler barrier
  #elif (BARRIER_TYPE==2)
    #define SOME_BARRIER() __asm{ mfence } // MemoryBarrier()   // hardware barrier
  #else
    #define SOME_BARRIER()                                      // noop
  #endif
#else
  #if (BARRIER_TYPE==1)
    #define SOME_BARRIER() asm volatile("" ::: "memory")        // compiler barrier
  #elif (BARRIER_TYPE==2)
    #define SOME_BARRIER() asm volatile("mfence" ::: "memory")  // hardware barrier
  #else
    #define SOME_BARRIER()                                      // noop
  #endif
#endif
```



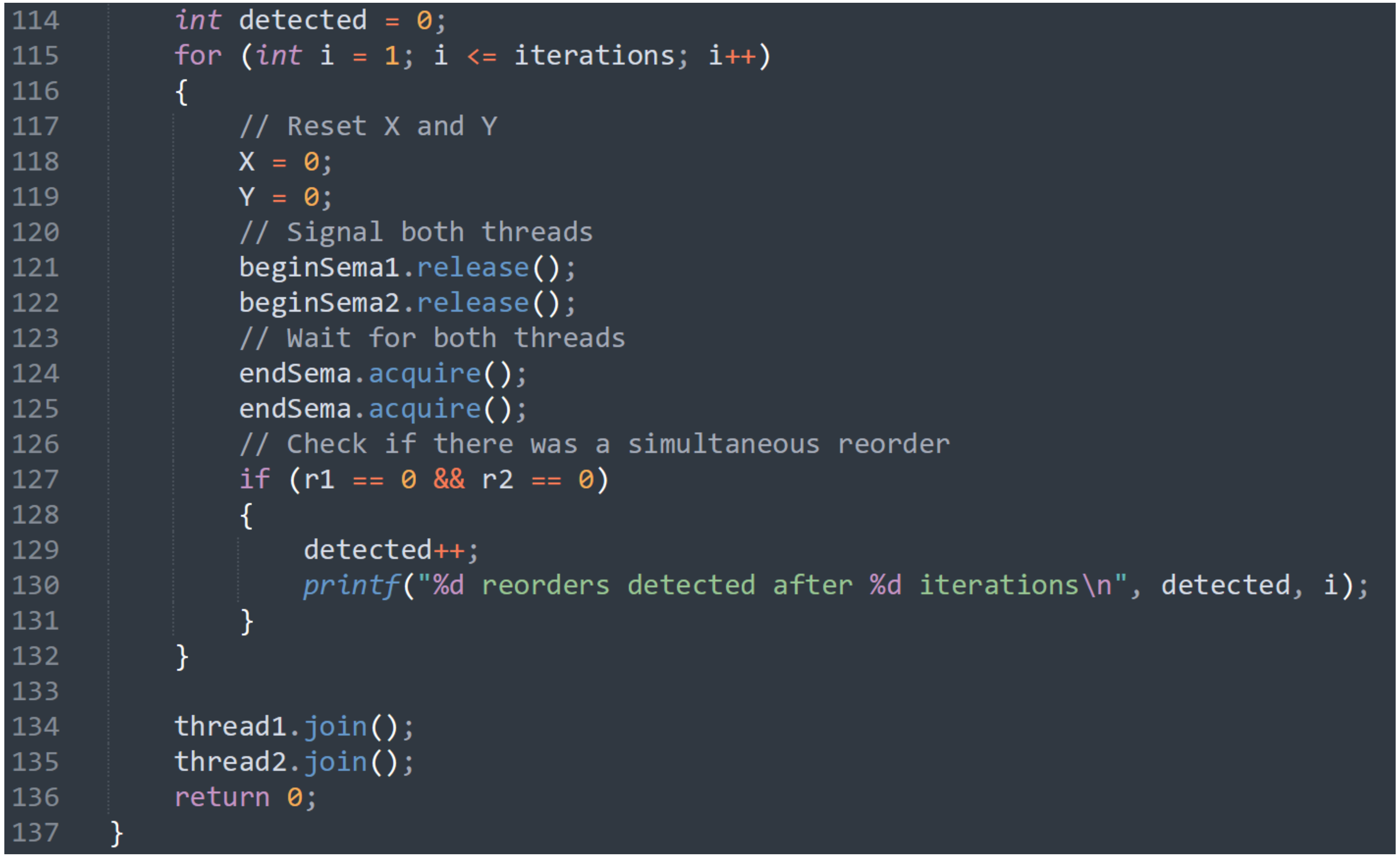
```
semaphore beginSema1;
semaphore beginSema2;
semaphore endSema;

int X, Y;
int r1, r2;
```

main(...)



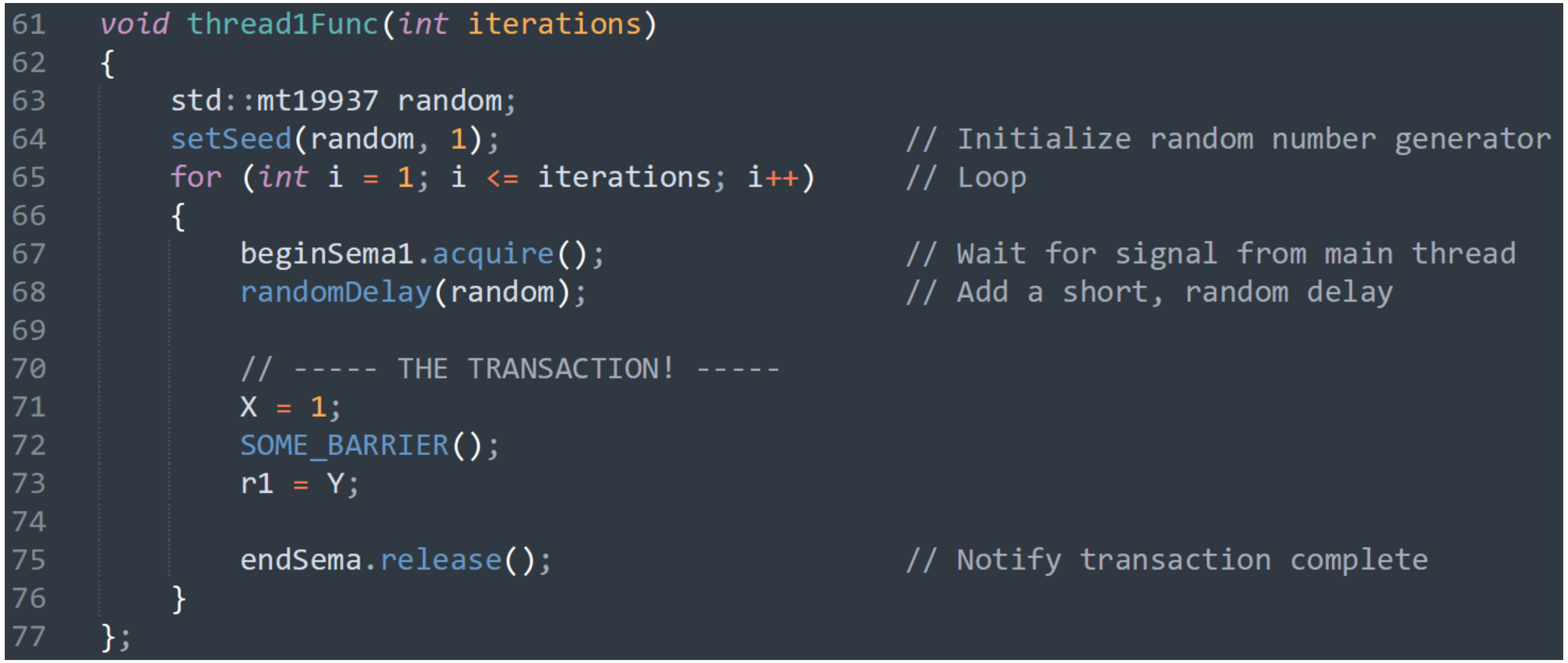
```
int main(int argc, char* argv[])
{
    int iterations = 100000;
    if (argc > 1)
    {
        int n = std::atoi(argv[1]);
        if (n > 0)
            iterations = n;
    }

    printf("BARRIER TYPE = %d\n", BARRIER_TYPE);
    printf("iterations = %d\n", iterations);
    printf("\n");

    std::thread thread1(thread1Func, iterations);
    std::thread thread2(thread2Func, iterations);
```

main(...) cont.



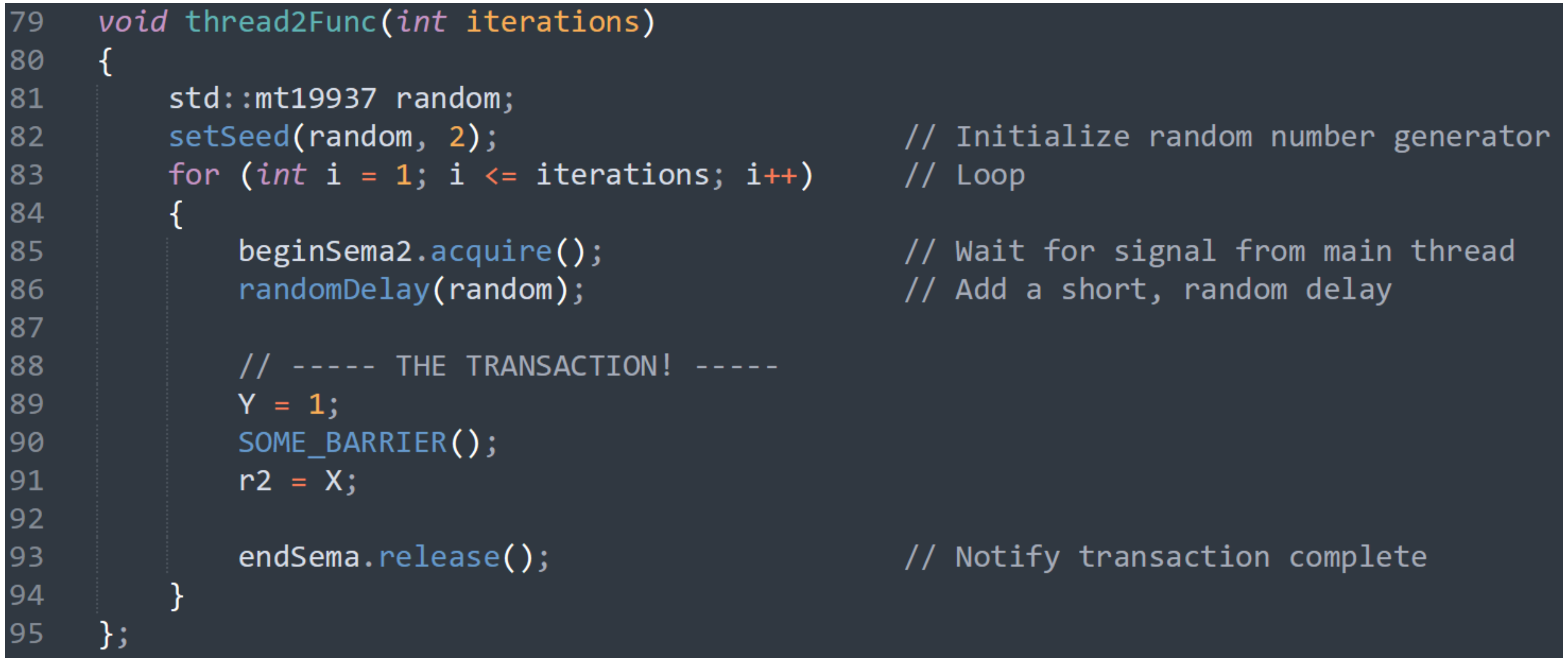
```
    int detected = 0;
    for (int i = 1; i <= iterations; i++)
    {
        // Reset X and Y
        X = 0;
        Y = 0;
        // Signal both threads
        beginSema1.release();
        beginSema2.release();
        // Wait for both threads
        endSema.acquire();
        endSema.acquire();
        // Check if there was a simultaneous reorder
        if (r1 == 0 && r2 == 0)
        {
            detected++;
            printf("%d reorders detected after %d iterations\n", detected, i);
        }
    }

    thread1.join();
    thread2.join();
    return 0;
}
```

thread1Func(...)



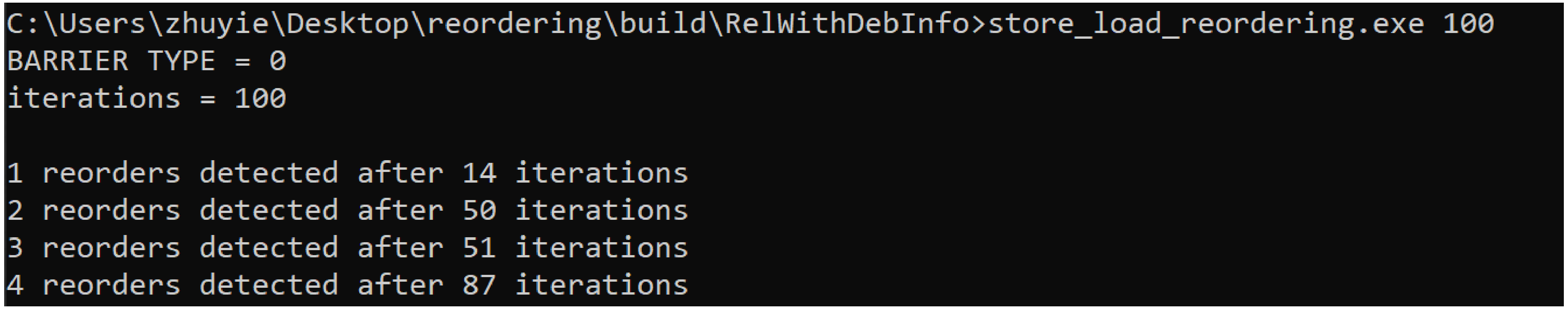
```
void thread1Func(int iterations)
{
    std::mt19937 random;
    setSeed(random, 1);                         // Initialize random number generator
    for (int i = 1; i <= iterations; i++)      // Loop
    {
        beginSema1.acquire();                   // Wait for signal from main thread
        randomDelay(random);                    // Add a short, random delay

        // ----- THE TRANSACTION! -----
        X = 1;
        SOME_BARRIER();
        r1 = Y;

        endSema.release();                      // Notify transaction complete
    }
};
```

thread2Func(...)



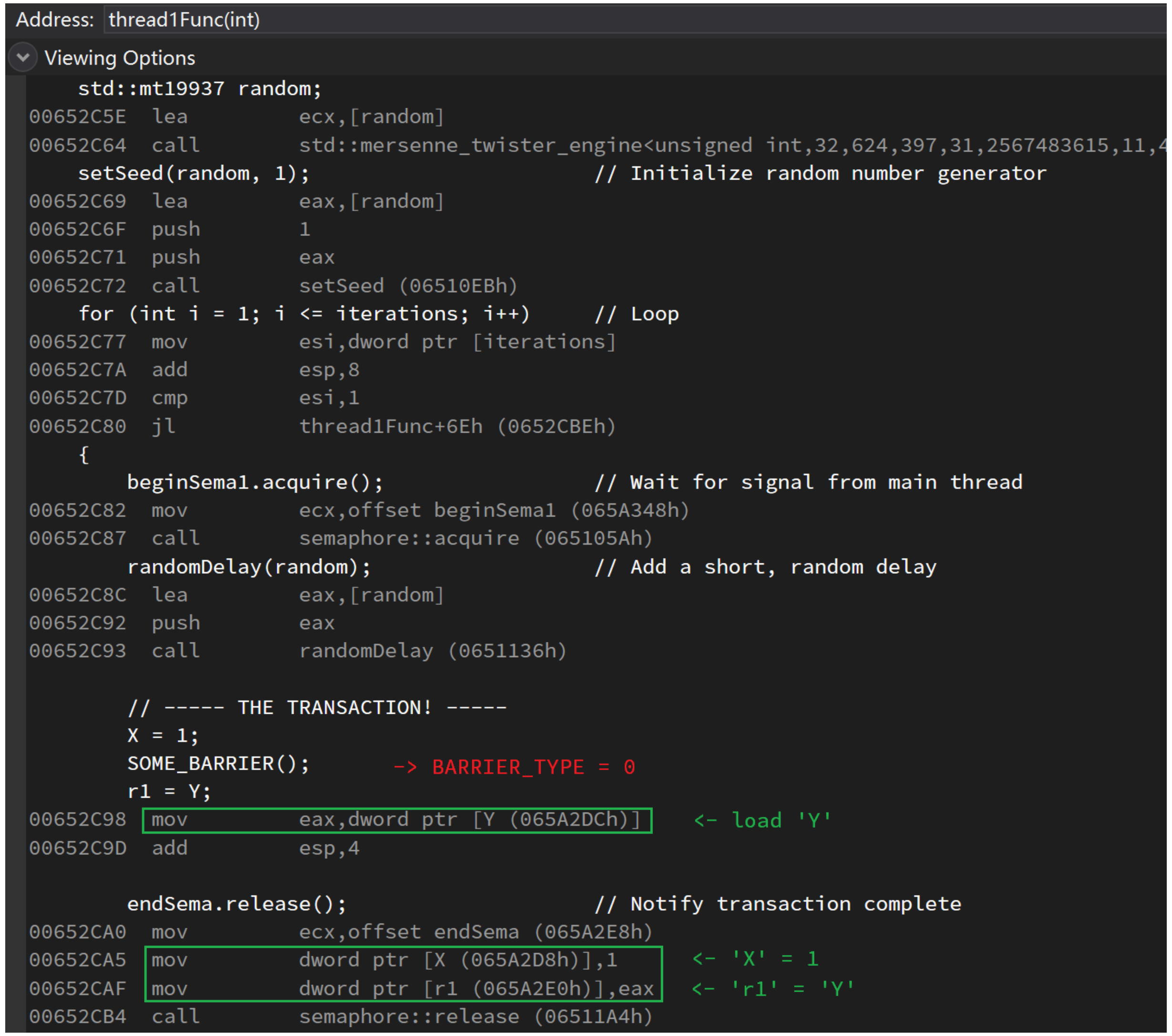
```cpp
void thread2Func(int iterations)
{
    std::mt19937 random;
    setSeed(random, 2);                       // Initialize random number generator
    for (int i = 1; i <= iterations; i++)     // Loop
    {
        beginSema2.acquire();                 // Wait for signal from main thread
        randomDelay(random);                  // Add a short, random delay

        // ----- THE TRANSACTION! -----
        Y = 1;
        SOME_BARRIER();
        r2 = X;

        endSema.release();                    // Notify transaction complete
    }
};
```

BARRIER TYPE = 0



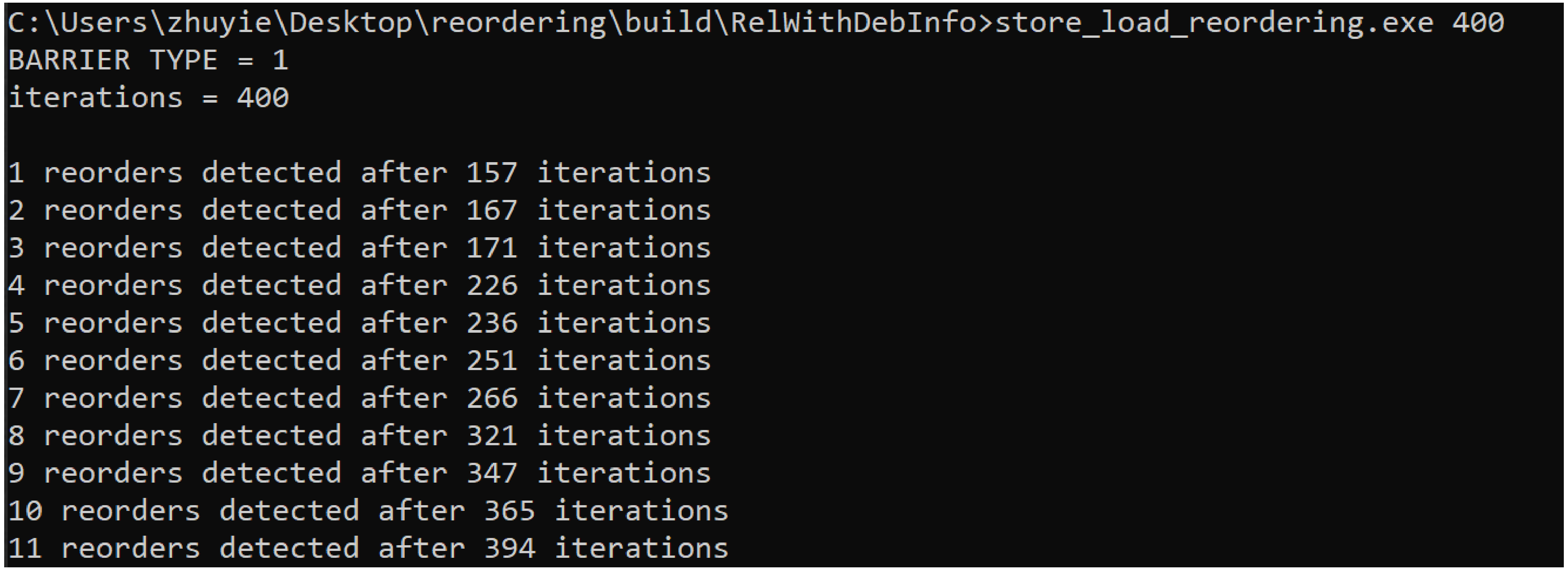
```
C:\Users\zhuyie\Desktop\reordering\build\RelWithDebInfo>store_load_reordering.exe 100
BARRIER TYPE = 0
iterations = 100

1 reorders detected after 14 iterations
2 reorders detected after 50 iterations
3 reorders detected after 51 iterations
4 reorders detected after 87 iterations
```

BARRIER TYPE = 0



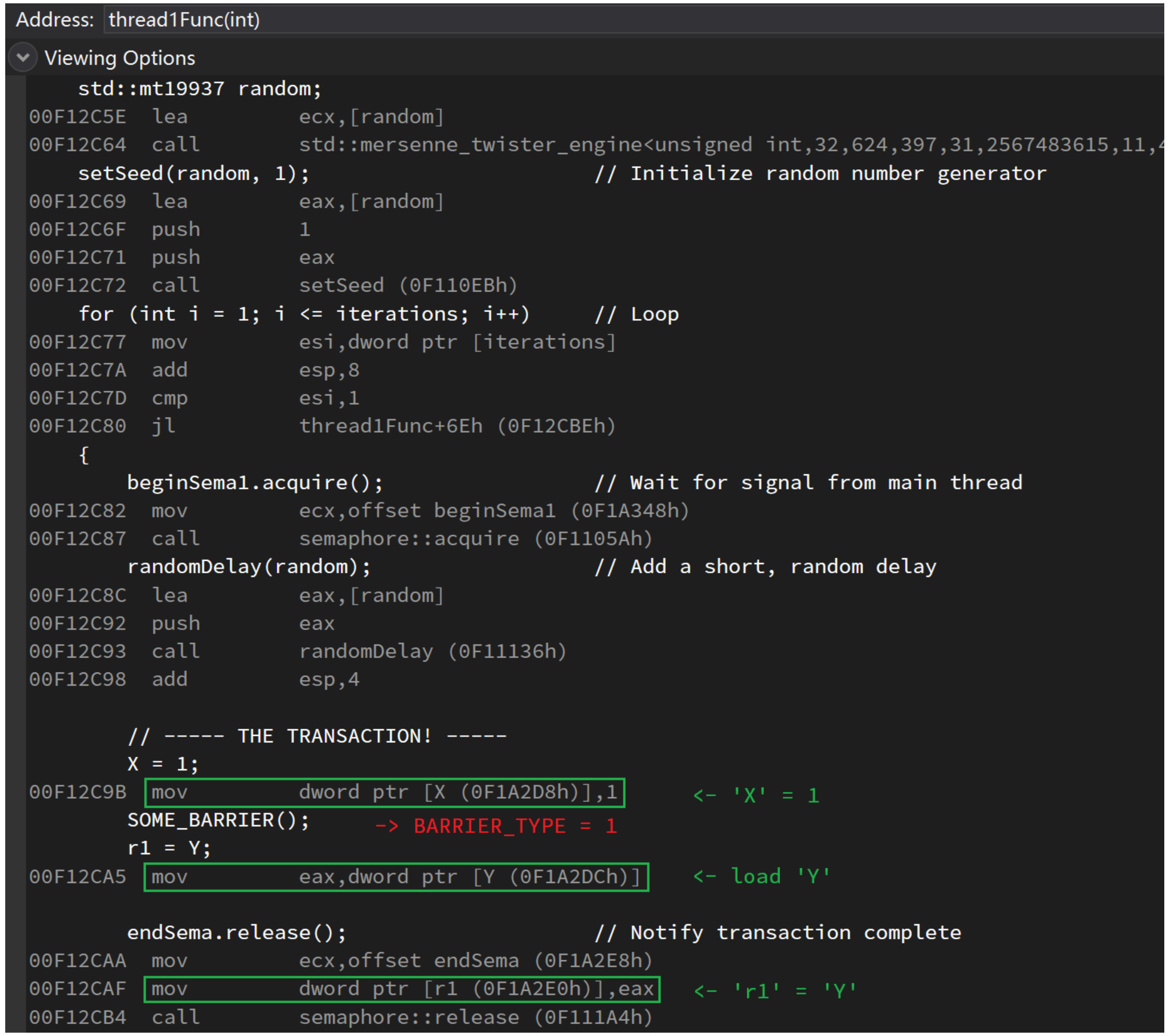
```
Address: thread1Func(int)
Viewing Options
    std::mt19937 random;
00652C5E  lea         ecx,[random]
00652C64  call        std::mersenne_twister_engine<unsigned int,32,624,397,31,2567483615,11,4
    setSeed(random, 1);                    // Initialize random number generator
00652C69  lea         eax,[random]
00652C6F  push        1
00652C71  push        eax
00652C72  call        setSeed (06510EBh)
    for (int i = 1; i <= iterations; i++)      // Loop
00652C77  mov         esi,dword ptr [iterations]
00652C7A  add         esp,8
00652C7D  cmp         esi,1
00652C80  jl          thread1Func+6Eh (0652CBEh)
    {
        beginSema1.acquire();              // Wait for signal from main thread
00652C82  mov         ecx,offset beginSema1 (065A348h)
00652C87  call        semaphore::acquire (065105Ah)
        randomDelay(random);               // Add a short, random delay
00652C8C  lea         eax,[random]
00652C92  push        eax
00652C93  call        randomDelay (0651136h)

        // ----- THE TRANSACTION! -----
        X = 1;
        SOME_BARRIER();      -> BARRIER_TYPE = 0
        r1 = Y;
00652C98  mov         eax,dword ptr [Y (065A2DCh)]   <- load 'Y'
00652C9D  add         esp,4

        endSema.release();                 // Notify transaction complete
00652CA0  mov         ecx,offset endSema (065A2E8h)
00652CA5  mov         dword ptr [X (065A2D8h)],1       <- 'X' = 1
00652CAF  mov         dword ptr [r1 (065A2E0h)],eax    <- 'r1' = 'Y'
00652CB4  call        semaphore::release (06511A4h)
```

BARRIER TYPE = 1



```
C:\Users\zhuyie\Desktop\reordering\build\RelWithDebInfo>store_load_reordering.exe 400
BARRIER TYPE = 1
iterations = 400

1 reorders detected after 157 iterations
2 reorders detected after 167 iterations
3 reorders detected after 171 iterations
4 reorders detected after 226 iterations
5 reorders detected after 236 iterations
6 reorders detected after 251 iterations
7 reorders detected after 266 iterations
8 reorders detected after 321 iterations
9 reorders detected after 347 iterations
10 reorders detected after 365 iterations
11 reorders detected after 394 iterations
```

BARRIER TYPE = 1

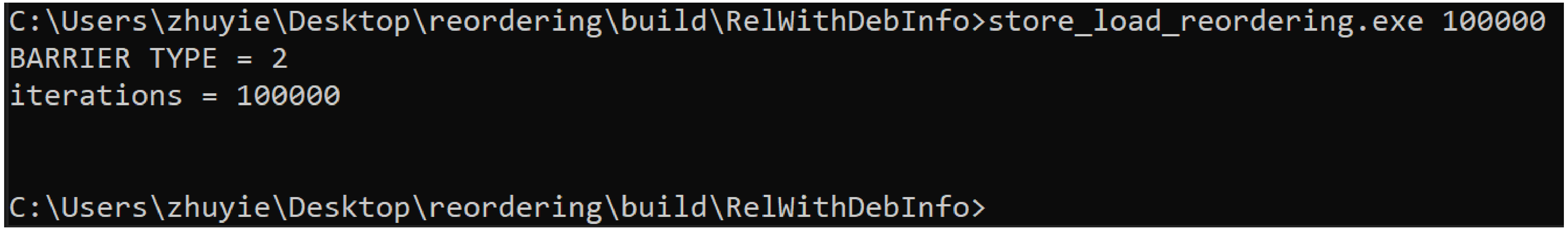


Address: thread1Func(int)

Viewing Options

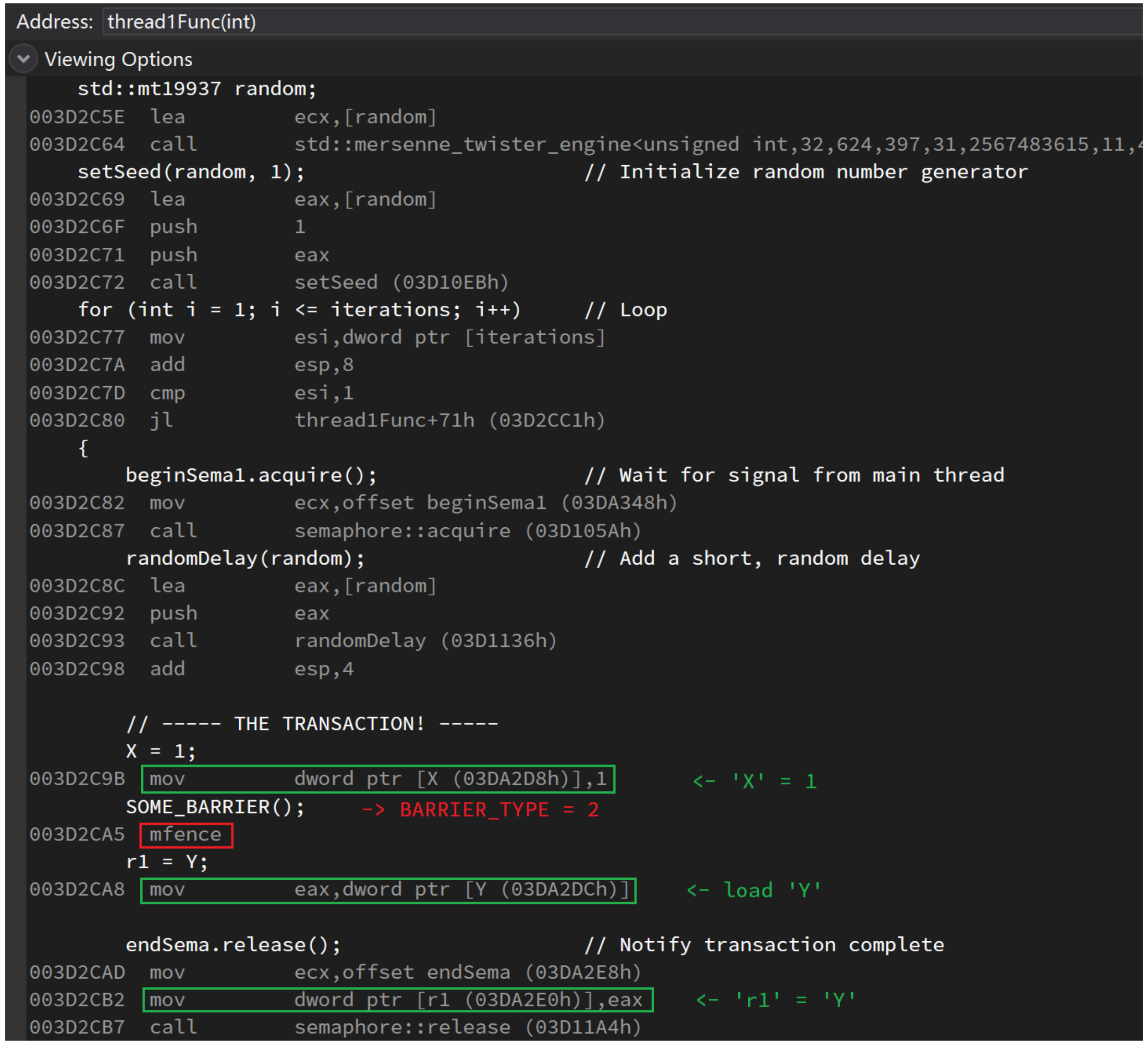
```
    std::mt19937 random;
00F12C5E  lea         ecx,[random]
00F12C64  call        std::mersenne_twister_engine<unsigned int,32,624,397,31,2567483615,11,
    setSeed(random, 1);                         // Initialize random number generator
00F12C69  lea         eax,[random]
00F12C6F  push        1
00F12C71  push        eax
00F12C72  call        setSeed (0F110EBh)
    for (int i = 1; i <= iterations; i++)     // Loop
00F12C77  mov         esi,dword ptr [iterations]
00F12C7A  add         esp,8
00F12C7D  cmp         esi,1
00F12C80  jl          thread1Func+6Eh (0F12CBEh)
    {
        beginSema1.acquire();                   // Wait for signal from main thread
00F12C82  mov         ecx,offset beginSema1 (0F1A348h)
00F12C87  call        semaphore::acquire (0F1105Ah)
        randomDelay(random);                    // Add a short, random delay
00F12C8C  lea         eax,[random]
00F12C92  push        eax
00F12C93  call        randomDelay (0F11136h)
00F12C98  add         esp,4

        // ----- THE TRANSACTION! -----
        X = 1;
00F12C9B  mov         dword ptr [X (0F1A2D8h)],1     <- 'X' = 1
        SOME_BARRIER();      -> BARRIER_TYPE = 1
        r1 = Y;
00F12CA5  mov         eax,dword ptr [Y (0F1A2DCh)]   <- load 'Y'

        endSema.release();                      // Notify transaction complete
00F12CAA  mov         ecx,offset endSema (0F1A2E8h)
00F12CAF  mov         dword ptr [r1 (0F1A2E0h)],eax  <- 'r1' = 'Y'
00F12CB4  call        semaphore::release (0F111A4h)
```

BARRIER TYPE = 2



```
C:\Users\zhuyie\Desktop\reordering\build\RelWithDebInfo>store_load_reordering.exe 100000
BARRIER TYPE = 2
iterations = 100000


C:\Users\zhuyie\Desktop\reordering\build\RelWithDebInfo>
```

BARRIER TYPE = 2



Address: thread1Func(int)

Viewing Options

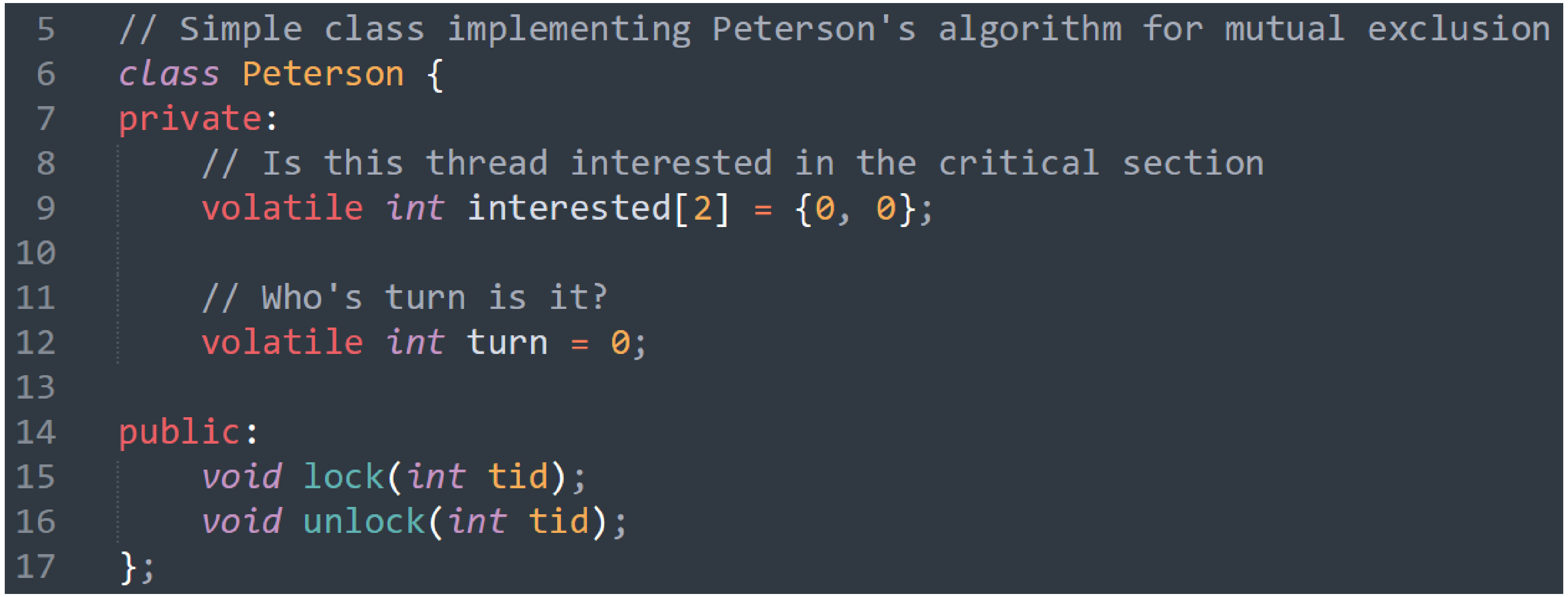
```
    std::mt19937 random;
003D2C5E  lea         ecx,[random]
003D2C64  call        std::mersenne_twister_engine<unsigned int,32,624,397,31,2567483615,11,
    setSeed(random, 1);                        // Initialize random number generator
003D2C69  lea         eax,[random]
003D2C6F  push        1
003D2C71  push        eax
003D2C72  call        setSeed (03D10EBh)
    for (int i = 1; i <= iterations; i++)      // Loop
003D2C77  mov         esi,dword ptr [iterations]
003D2C7A  add         esp,8
003D2C7D  cmp         esi,1
003D2C80  jl          thread1Func+71h (03D2CC1h)
    {
        beginSema1.acquire();                  // Wait for signal from main thread
003D2C82  mov         ecx,offset beginSema1 (03DA348h)
003D2C87  call        semaphore::acquire (03D105Ah)
        randomDelay(random);                   // Add a short, random delay
003D2C8C  lea         eax,[random]
003D2C92  push        eax
003D2C93  call        randomDelay (03D1136h)
003D2C98  add         esp,4

        // ----- THE TRANSACTION! -----
        X = 1;
003D2C9B  mov         dword ptr [X (03DA2D8h)],1      <- 'X' = 1
        SOME_BARRIER();      -> BARRIER_TYPE = 2
003D2CA5  mfence
        r1 = Y;
003D2CA8  mov         eax,dword ptr [Y (03DA2DCh)]     <- load 'Y'

        endSema.release();                     // Notify transaction complete
003D2CAD  mov         ecx,offset endSema (03DA2E8h)
003D2CB2  mov         dword ptr [r1 (03DA2E0h)],eax    <- 'r1' = 'Y'
003D2CB7  call        semaphore::release (03D11A4h)
```

Example 2: Peterson lock

A naïve critical section implemetation

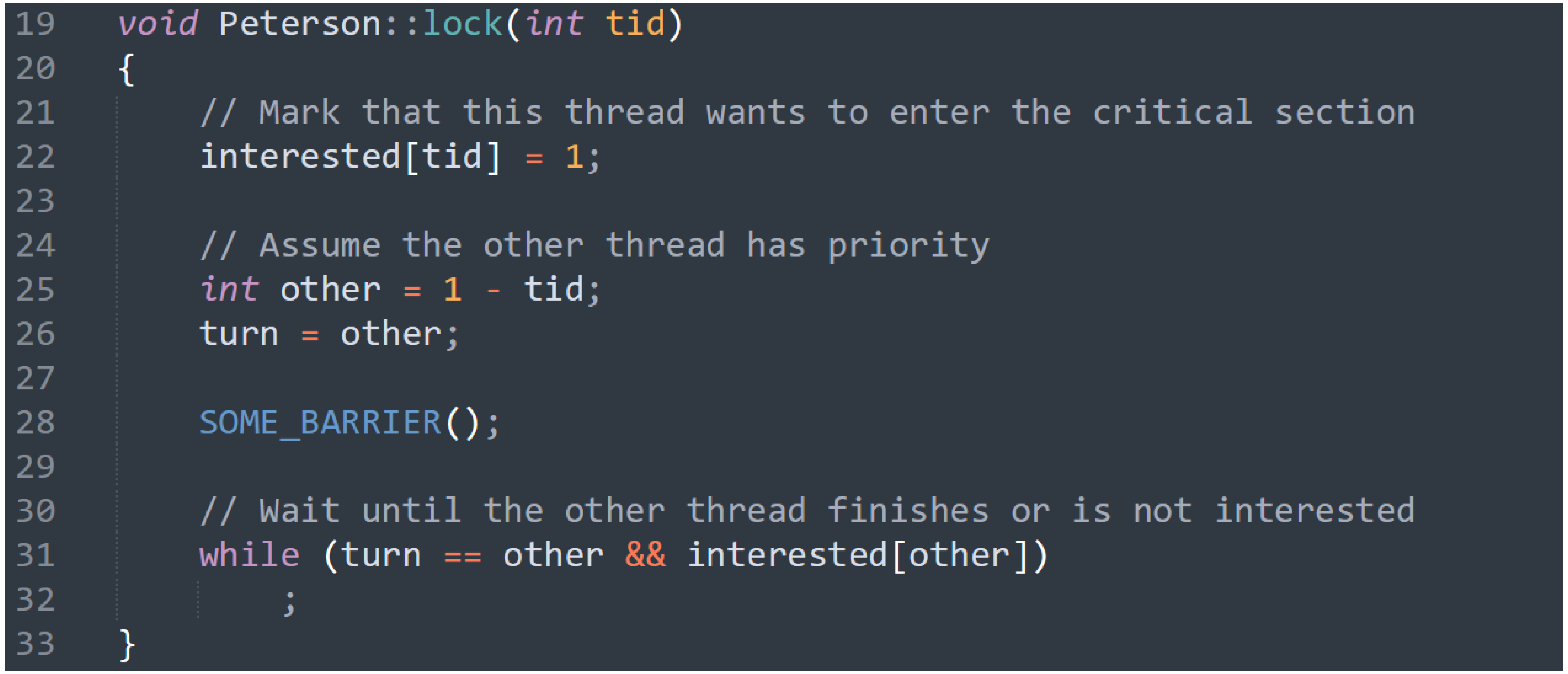
- To understand how hardware memory reordering can cause real-world bugs, let’s take a look at a C++ implementation of Peterson’s algorithm for two threads.
- Peterson's algorithm is a concurrent programming algorithm for **mutual exclusion** that allows two or more processes to share a single-use resource without conflict, using **only** shared memory for communication. It was formulated by Gary L. Peterson in 1981.

class Peterson



```
// Simple class implementing Peterson's algorithm for mutual exclusion
class Peterson {
private:
    // Is this thread interested in the critical section
    volatile int interested[2] = {0, 0};

    // Who's turn is it?
    volatile int turn = 0;

public:
    void lock(int tid);
    void unlock(int tid);
};
```

lock(...)

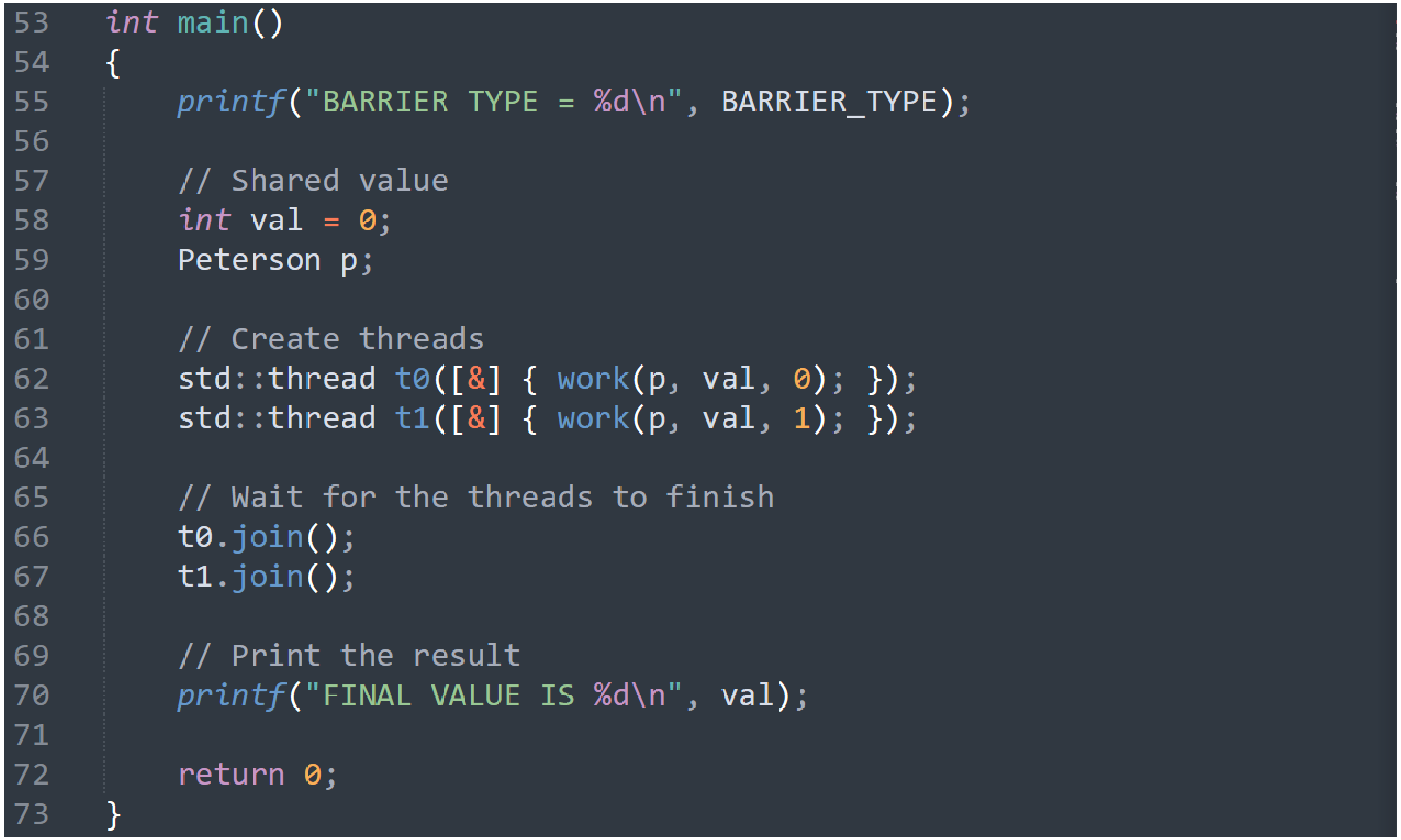


```
void Peterson::lock(int tid)
{
    // Mark that this thread wants to enter the critical section
    interested[tid] = 1;

    // Assume the other thread has priority
    int other = 1 - tid;
    turn = other;

    SOME_BARRIER();

    // Wait until the other thread finishes or is not interested
    while (turn == other && interested[other])
        ;
}
```

unlock(...)

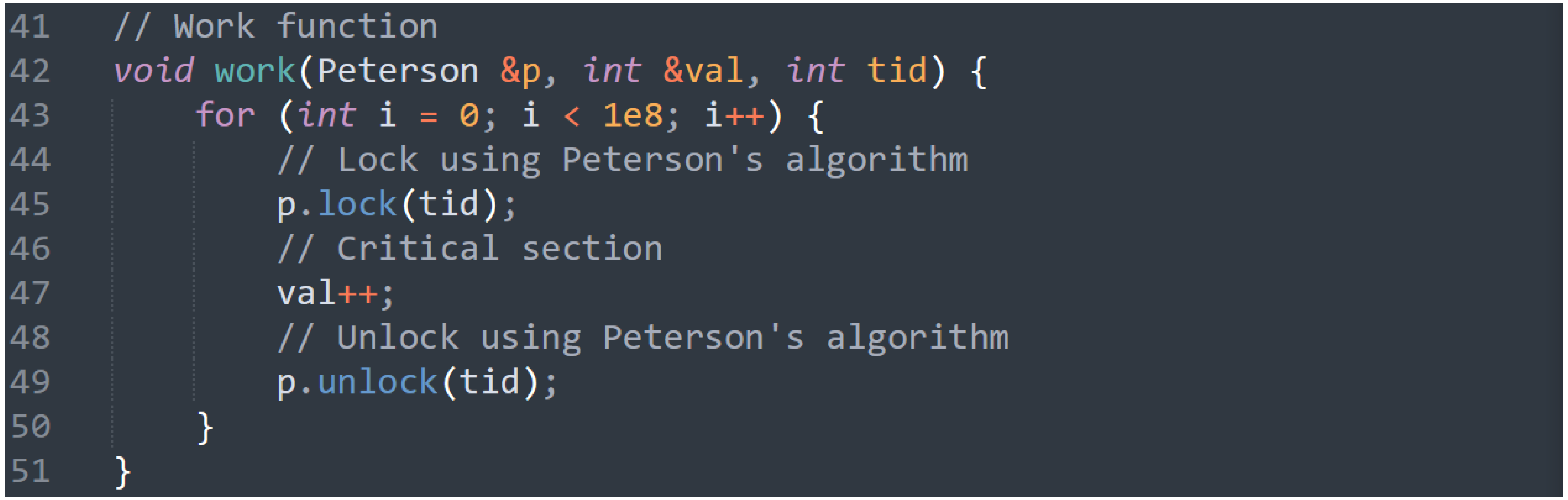
```
void Peterson::unlock(int tid)
{
    // Mark that this thread is no longer interested
    interested[tid] = 0;
}
```

main(...)



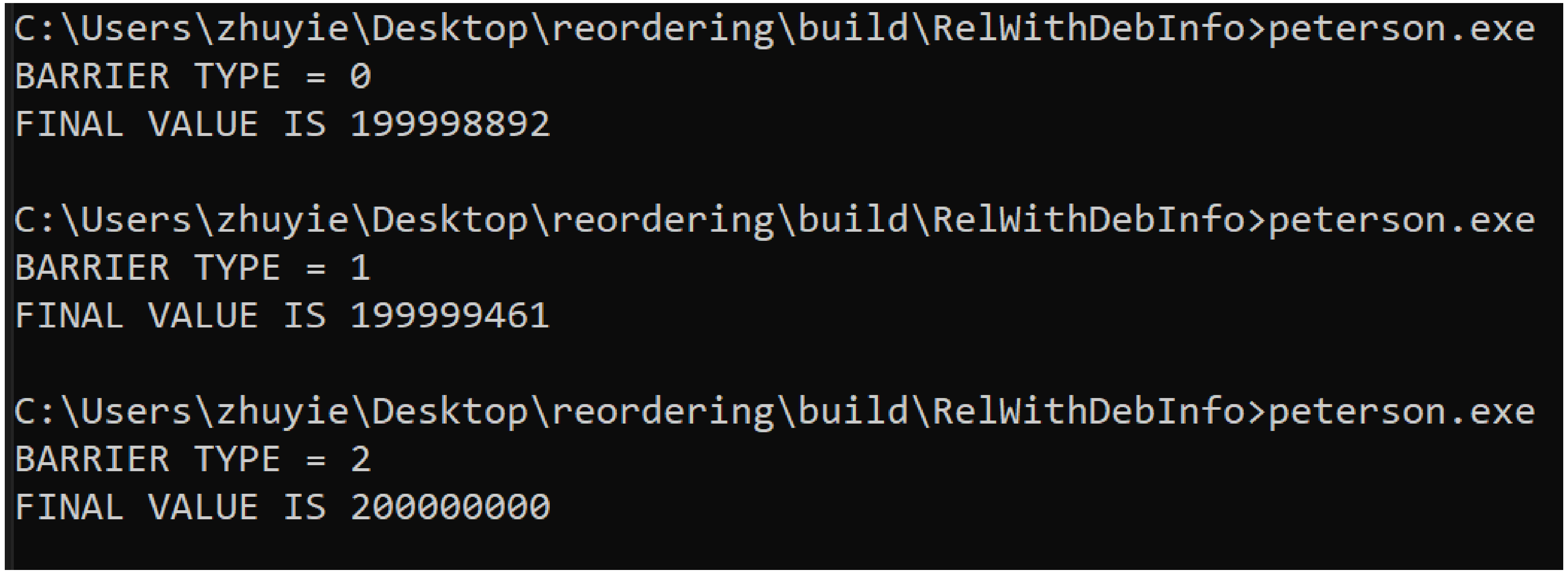
```
int main()
{
    printf("BARRIER TYPE = %d\n", BARRIER_TYPE);

    // Shared value
    int val = 0;
    Peterson p;

    // Create threads
    std::thread t0([&] { work(p, val, 0); });
    std::thread t1([&] { work(p, val, 1); });

    // Wait for the threads to finish
    t0.join();
    t1.join();

    // Print the result
    printf("FINAL VALUE IS %d\n", val);

    return 0;
}
```

work(...)



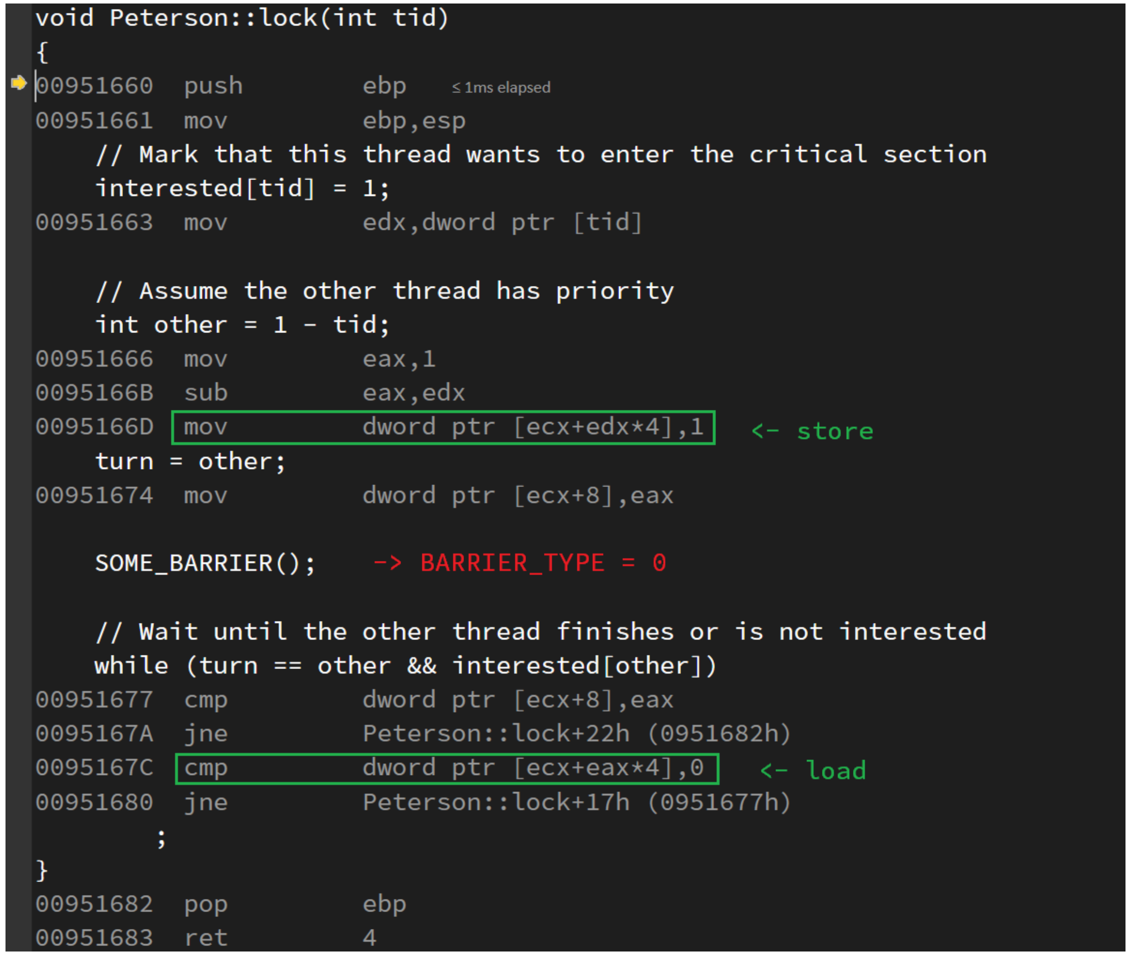
```
// Work function
void work(Peterson &p, int &val, int tid) {
    for (int i = 0; i < 1e8; i++) {
        // Lock using Peterson's algorithm
        p.lock(tid);
        // Critical section
        val++;
        // Unlock using Peterson's algorithm
        p.unlock(tid);
    }
}
```

Output



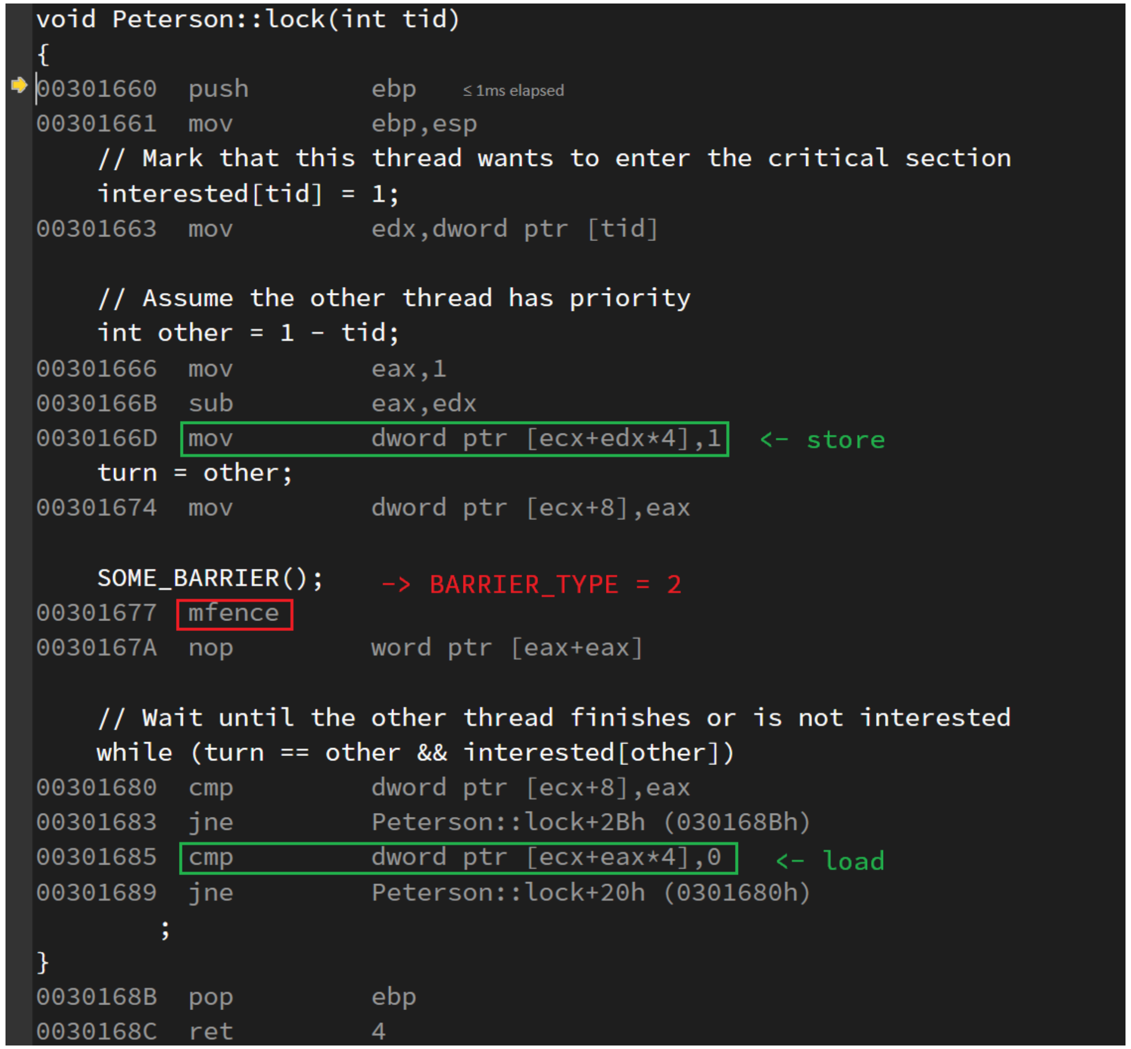
```
C:\Users\zhuyie\Desktop\reordering\build\RelWithDebInfo>peterson.exe
BARRIER TYPE = 0
FINAL VALUE IS 199998892

C:\Users\zhuyie\Desktop\reordering\build\RelWithDebInfo>peterson.exe
BARRIER TYPE = 1
FINAL VALUE IS 199999461

C:\Users\zhuyie\Desktop\reordering\build\RelWithDebInfo>peterson.exe
BARRIER TYPE = 2
FINAL VALUE IS 200000000
```

BARRIER TYPE = 0



```
void Peterson::lock(int tid)
{
00951660  push        ebp     ≤ 1ms elapsed
00951661  mov         ebp,esp
    // Mark that this thread wants to enter the critical section
    interested[tid] = 1;
00951663  mov         edx,dword ptr [tid]

    // Assume the other thread has priority
    int other = 1 - tid;
00951666  mov         eax,1
0095166B  sub         eax,edx
0095166D  mov         dword ptr [ecx+edx*4],1   <- store
    turn = other;
00951674  mov         dword ptr [ecx+8],eax

    SOME_BARRIER();    -> BARRIER_TYPE = 0

    // Wait until the other thread finishes or is not interested
    while (turn == other && interested[other])
00951677  cmp         dword ptr [ecx+8],eax
0095167A  jne         Peterson::lock+22h (0951682h)
0095167C  cmp         dword ptr [ecx+eax*4],0   <- load
00951680  jne         Peterson::lock+17h (0951677h)
        ;
}
00951682  pop         ebp
00951683  ret         4
```

BARRIER TYPE = 2



```
void Peterson::lock(int tid)
{
00301660  push         ebp    ≤ 1ms elapsed
00301661  mov          ebp,esp
    // Mark that this thread wants to enter the critical section
    interested[tid] = 1;
00301663  mov          edx,dword ptr [tid]

    // Assume the other thread has priority
    int other = 1 - tid;
00301666  mov          eax,1
0030166B  sub          eax,edx
0030166D  mov          dword ptr [ecx+edx*4],1   <- store
    turn = other;
00301674  mov          dword ptr [ecx+8],eax

    SOME_BARRIER();    -> BARRIER_TYPE = 2
00301677  mfence
0030167A  nop          word ptr [eax+eax]

    // Wait until the other thread finishes or is not interested
    while (turn == other && interested[other])
00301680  cmp          dword ptr [ecx+8],eax
00301683  jne          Peterson::lock+2Bh (030168Bh)
00301685  cmp          dword ptr [ecx+eax*4],0   <- load
00301689  jne          Peterson::lock+20h (0301680h)
        ;
}
0030168B  pop          ebp
0030168C  ret          4
```

Conclusion

- There are only two hard things in computer science: cache invalidation, naming things.
- Maybe the third hardest thing is: **seeing things in order**.
- Multiprocessors reorder memory operations in **unintuitive** and strange ways.
- This behavior is required for **performance**.
- Carefully think through which portion of code will **run in parallel**.
- **Synchronization** to the rescue.

References

- https://en.wikipedia.org/wiki/Consistency_model
- https://preshing.com/20120930/weak-vs-strong-memory-models/
- https://preshing.com/20120515/memory-reordering-caught-in-the-act/
- https://coffeebeforearch.github.io/2020/11/29/hardware-memory-ordering.html
- https://www.cs.utexas.edu/~bornholt/post/memory-models.html