Lecture 6:
Concurrency Control:
Mutual Exclusion

Xin Liu

xl24j@fsu.edu

COP 4610 Operating Systems

Outline

- Root Cause of Lock Failures

- Peterson Algorithm

- Spin Lock

Introduction to Synchronization

- Two robots are programmed to maintain the milk inventory at a store…
- They are not aware of each other’s presence…

Robot: Dumb

Robot: Dumber

Introduction to Synchronization

```
if (no milk) {
     go get milk;
 }
```

Robot: Dumb

Robot: Dumber

Motivating Example: Too Much Milk

Dumb

10:00 Look into fridge:

Out of milk

Dumber

Motivating Example: Too Much Milk

Dumb

10:00 Look into fridge:

Out of milk

10:05 Head for the warehouse

Dumber

Motivating Example: Too Much Milk

Dumb

10:05 Head for the warehouse

Dumber

10:10 Look into fridge: Out of milk

Motivating Example: Too Much Milk

Dumb

Dumber

10:10 Look into fridge:

Out of milk

10:15 Head for the warehouse

Motivating Example: Too Much Milk

Dumb	Dumber
	10:15 Head for the warehouse
10:20 Arrive with milk	

Motivating Example: Too Much Milk

Dumb

10:20 Arrive with milk

Dumber

10:15 Head for the warehouse

Motivating Example: Too Much Milk

Dumb		Dumber
10:20	Arrive with milk	
10:25	Go party	

Motivating Example: Too Much Milk

Dumb

10:20 Arrive with milk

10:25 Go party

Dumber

10:30 Arrive with milk: “Uh oh…”

Non-Atomic Operations

- Operations like x++ involve multiple steps:
  - Read the current value of x.
  - Increment x by 1.
  - Write the new value back to x.

Example: X++

- Assume x is initially 0, and you have two threads, each of which will perform the x++ operation 10,000 times. In theory, the final result should be 20,000.
- However, without synchronization, the following situation might occur:
  - Thread 1 reads the value of x, let’s say x = 9999.
  - Thread 2 also reads the value of x almost at the same time, and x is still 9999.
  - Thread 1 calculates 9999 + 1 = 10000 and writes it back to x, so now x = 10000.
  - Thread 2 also calculates 9999 + 1 = 10000 and writes it back to x, overwriting Thread 1’s result, so x remains 10000 instead of 10001.
  - This situation can happen frequently, which is why the final result is always less than the expected 20,000, as many increments are being overwritten.
- Example 1:

  https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_1_x%2B%2B.c

Too Much Milk and x++

Load

```
if (no milk) {
    go get milk;
 }
```

Store

X++:

Read the current value of x and increment x by 1.

Write the new value back to x.

Robot: Dumb

Robot: Dumber

Definitions

- **Synchronization**: uses atomic operations to ensure cooperation among threads.
  - It allows two robots to be aware of each other's presence.
- **Mutual exclusion**: ensures one thread can do something without the interference of other threads.
  - Only one robot is allowed to place one box of milk in the fridge at a time.
- **Critical section**: a piece of code that only one thread can execute at a time.
  - The act of placing milk in the fridge. Only one robot can perform this action at a time.

More on Critical Section

- A **lock** prevents a thread from doing something
  - A thread should lock before entering a critical section
  - A thread should unlock when leaving the critical section
  - A thread should wait if the critical section is locked
    - Synchronization often involves waiting

Too Much Milk: Solution 1

```
if (no milk) {
 if (no note) {
    // leave a note;
    // go get milk;
    // remove the note;
 }
}
```

- Basic idea of solution 1
  - Leave a note (kind of like a lock)
  - Remove the note (kind of like a unlock)
  - Don’t go to get milk if the note is around (wait)

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {
```

Dumber

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {
```

Dumber

```
10:01 if (no milk) {
```

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {
```

Dumber

```
10:01 if (no milk) {
10:02   if (no note) {
```

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {


10:03   if (no note) {
```

Dumber

```

10:01 if (no milk) {
10:02   if (no note) {
```

Too Much Milk: Solution 1

Dumb

```
10:00 if (no milk) {
10:03   if (no note) {
10:04     // leave a note
```

Dumber

```
10:01 if (no milk) {
10:02   if (no note) {
```

Too Much Milk: Solution 1

Dumb

```
10:03    if (no note) {
10:04      // leave a note
```

Dumber

```
10:01 if (no milk) {
10:02   if (no note) {
10:05     // leave a note
```

Too Much Milk: Solution 1

Dumb

```

10:03    if (no note) {
10:04       // leave a note

10:06       // go get milk
```

Dumber

```
10:02    if (no note) {


10:05       // leave a note
```

Too Much Milk: Solution 1

Dumb

```
10:03   if (no note) {
10:04     // leave a note

10:06     // go get milk
```

Dumber

```
10:05     // leave a note

10:07     // go get milk
```

Demo of Lock Failure

```
retry:
        if (lock_status != UNLOCK) {  ← Load
          goto retry;
        }
        lock_status = LOCK;  // Acquire the lock  ← Store

        x++;  // Critical section

        lock_status = UNLOCK;  // Release the lock
```

https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_2_x%2B%2B_faulty_lock.c

The Root Cause

```
retry:
        if (lock_status != UNLOCK) {  ←  Load
           goto retry;
        }
        lock_status = LOCK;  // Acquire the lock   ←  Store


        x++;  // Critical section


        lock_status = UNLOCK;  // Release the lock
```

Example 2:
https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_2_x%2B%2B_faulty_lock.c

The processor does not guarantee the atomicity of load and store operations by default.

Important

Too Much Milk: Solution 2

- Okay…solution 1 does not work
- The notes are posted too late…
- What if both robots begin by leaving their own notes?

Too Much Milk: Solution 2

```
// leave a note; ← Store
if (no note from the other) {← Load
 if (no milk) {
    // go get milk;
 }
}
// remove the note;
```

Too Much Milk: Solution 2

Dumb

`10:00 // leave a note`

Dumber

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note
```

Dumber

```
10:01 // leave a note
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note

10:02 if (no note from
 Dumber) {…}
```

Dumber

```
10:01 // leave a note
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note

10:02 if (no note from Dumber) {…}
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb) {…}
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note

10:02 if (no note from
  Dumber) {…}

10:04 // remove the note
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}
```

Too Much Milk: Solution 2

Dumb

```
10:00 // leave a note


10:02 if (no note from
 Dumber) {…}


10:04 // remove the note
```

Dumber

```

10:01 // leave a note


10:03 if (no note from Dumb)
  {…}
```

Too Much Milk: Solution 2

Dumb

```
10:02 if (no note from
 Dumber) {…}

10:04 // remove the note
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}

10:05 // remove the note
```

Too Much Milk: Solution 2

Dumb

```
10:02 if (no note from
 Dumber) {…}

10:04 // remove the note
```

Dumber

```
10:01 // leave a note

10:03 if (no note from Dumb)
  {…}

10:05 // remove the note
```

Too Much Milk: Solution 2

- Solution 2 does not work
- The notes are found too late…

Too Much Milk: Solution 3

Dumb

```
// leave Dumb's note
while (Dumber's note) { };
if (no milk) {
  // go get milk
}
// remove Dumb's note
```

Dumber

```
// leave Dumber's note
if (no Dumb's note) {
  if (no milk) {
    // go get milk
  }
}
// remove Dumber's note
```

Too Much Milk Solution 3

- How do we verify the correctness of a solution?
- Test arbitrary interleaving of locking and checking locks
  - In this case, leaving notes and checking notes

Dumber Challenges Dumb: Case 1

```
Dumb                                       Dumber
// leave Dumb's note
while (Dumber's note) { };
                                           // leave Dumber's note
if (no milk) {
  // go get milk
}
                                           if (no Dumb's note) {
                                           }
                                           // remove Dumber's note
// remove Dumb's note
```

Time

Dumber Challenges Dumb: Case 2

```
Dumb                                  Dumber
// leave Dumb's note

                                      // leave Dumber's note

while (Dumber's note) { };

                                      if (no Dumb's note) {
                                      }
                                      // remove Dumber's note

if (no milk) {
  // go get milk
}
// remove Dumb's note
```

Time

Dumber Challenges Dumb: Case 3

```
Dumb                                    Dumber
// leave Dumb's note

                                        // leave Dumber's note
                                        if (no Dumb's note) {
while (Dumber's note) { };              }
                                        // remove Dumber's note




if (no milk) {
   // go get milk
}
// remove Dumb's note
```

Time

Dumb Challenges Dumber: Case 1

Time

```
Dumb                                   Dumber
                                       // leave Dumber’s note
                                       if (no Dumb’s note) {
// leave Dumb’s note
while (Dumber’s note) { };
                                         if (no milk) {
                                           // go get milk
                                         }
                                       }
                                       // remove Dumber’s note
if (no milk) {
}
// remove Dumb’s note
```

Dumb Challenges Dumber: Case 2

```
Dumb                              Dumber
                                  // leave Dumber’s note

// leave Dumb’s note

                                  if (no Dumb’s note) {
while (Dumber’s note) { };        }
                                  // remove Dumber’s note

if (no milk) {
  // go get milk
}
// remove Dumb’s note
```

Time

Dumb Challenges Dumber: Case 3

```
Dumb                           Dumber
                               // leave Dumber's note
// leave Dumb's note
while (Dumber's note) { };
                               if (no Dumb's note) {
                               }
                               // remove Dumber's note
if (no milk) {
  // go get milk
}
// remove Dumb's note
```

Time

Lessons Learned

- Although it works, Solution 3 is ugly
  - Difficult to verify correctness
    - This is also true for other algorithms.
  - While Dumb is waiting, it consumes CPU time (**busy waiting**)
  - Two threads have different code
    - Difficult to generalize to N threads
    - We need a simple and straightforward approach.

Peterson Algorithm

- Flags (flag[2]):
  - An array indicating each thread's intention to enter the critical section.
  - flag[0] for Thread 0, flag[1] for Thread 1.
- Turn Variable (turn):
  - A shared variable indicating whose turn it is to enter the critical section.

Peterson Algorithm (Cont.)

1. Express Intent to Enter Critical Section:
   - Thread i sets flag[i] = true.
   - Indicates that it wants to enter the critical section.
2. Set Turn to the Other Thread:
   - Sets turn = j (where j is the other thread).
   - Gives priority to the other thread.
3. Wait Loop (Entry Protocol):
   - While flag[j] == true and turn == j, the thread waits.
   - This means if the other thread wants to enter and it's their turn, wait.
4. Critical Section Execution:
   - Once the condition is false, the thread enters the critical section safely.
   - Performs the necessary operations.
5. Exit Protocol:
   - Thread i sets flag[i] = false.
   - Indicates it no longer needs access to the critical section.

Peterson Algorithm (Cont.)

- Essentially, the core idea of the Peterson algorithm is:
  - If I want to enter the critical section, I first check the current priority (using the turn variable).
  - If the priority is in my favor (no other thread has priority), I can enter the critical section.
  - If the priority is with another thread and it also wants to enter the critical section (determined by flag[j]), I wait until the priority shifts to me

- Example 3:
  - https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_3_peterson.c

Peterson Algorithm (Cont.)

- Even when using the Peterson algorithm, errors may still occur on modern multiprocessor systems.
- The reason is that the Peterson algorithm relies on strict memory ordering and atomic access to shared variables, but optimizations in modern CPUs and compilers can violate these assumptions.

Implementing Mutual Exclusion with Shared Memory

- Failed Attempt:
  - ex_2_x++_faulty_lock.c
- (Partially) Successful Attempt:
  - ex_3_peterson.c
- The Fundamental Challenge of Implementing Mutual Exclusion:
- **You cannot read and write shared memory simultaneously.**
  - During load (observing the surroundings), you cannot write, only "take a quick glance and then close your eyes."
    - What you see is immediately outdated.
  - During store (changing the physical state), you cannot read, only "act blindly."
    - You don’t know what exactly you’ve changed.

Two Approaches to Solve the Problem

- Propose an Algorithm
  - Example: Peterson, Dekker, etc.

- Change the Assumptions
  - If software alone isn’t enough, we can turn to hardware for help.
    - Assume the hardware provides a "single atomic" load + store instruction.
    - All robots (thread) close their eyes, take a look (load), and then leave a note (store).
    - If multiple threads request access simultaneously, the hardware selects one to proceed.
    - Others must wait until the selected thread finishes before continuing.

Atomic Operations in x86: Lock Prefix

- Example: ex_4_x++_atomic.c https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_4_x%2B%2B_atomic.c
  - Goal: x = 200000

- The lock prefix itself is not an atomic instruction.
- It makes certain instructions atomic:
  - When the lock prefix is applied, the instruction that follows it is executed atomically, meaning it cannot be interrupted by other processors or threads.
  - It ensures that the operation is performed without interference from other threads in a multi-core system.

Atomic Operations in x86: Lock Prefix (Cont.)

- Examples of atomic instructions using the lock prefix:
  - lock add: Atomic addition
  - lock sub: Atomic subtraction
  - lock xchg: Atomic exchange (swap values)
  - lock cmpxchg: Atomic compare-and-swap
  - lock inc: Atomic increment
  - lock dec: Atomic decrement

Atomic Operations in x86: Lock Prefix (Cont.)

- More Atomic Operations:
  - Standard library header <stdatomic.h> - cppreference.com

xchg

- Atomic exchange (load + store)
  - The xchg instruction implicitly includes the lock prefix for atomic operands, so there's no need to explicitly add lock.

```
int xchg(volatile int *addr, int newval) {
   int result;                          // Holds the original value at *addr

   asm volatile (
      "lock xchg %0, %1"       // Atomically swaps *addr and newval
      : "+m"(*addr),                // *addr is both read and written
        "=a"(result)                // Store the original *addr value in result
      : "1"(newval)                 // newval is swapped with *addr
   );

   return result;                   // Return the original value of *addr
}
```

Spin Lock

```
int lock_status = YES;                    // Shared variable to represent the lock state

void lock() {
retry:
  int got = xchg(&lock_status, NOPE);  // Try to acquire the lock by setting loack_status to NOPE
  if (got == NOPE)                        // If the lock is already held (NOPE), retry
    goto retry;
  assert(got == YES);                     // Ensure the lock was successfully acquired
}

void unlock() {
  xchg(&lock_status, YES);                // Release the lock by setting table to YES
}
```

```
int lock_status = 0;
void lock() { while (xchg(&lock_status, 1)) ; }
void unlock() { xchg(&lock_status, 0); }
```

Model of Spin Lock

- Ensure that previous store operations are written to memory:
  - Atomic instructions guarantee that all previous store operations (writes to memory) have been completed and written to memory before the atomic instruction is executed. This prevents earlier writes from being "overwritten" or causing race conditions if they haven't yet been written.
- Ensure that load/store operations are not reordered with atomic instructions:
  - Atomic instructions ensure that load (read) and store (write) operations are not reordered with the atomic instruction. This guarantees consistent execution order and prevents the CPU from rearranging operations, which could lead to race conditions in a multi-threaded environment.

- The essence of spin lock: **Eliminating Concurrency!**
  - Other threads must keep checking in a loop until the lock is released.
  - Example:

https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_5_x%2B%2B_xchg.c

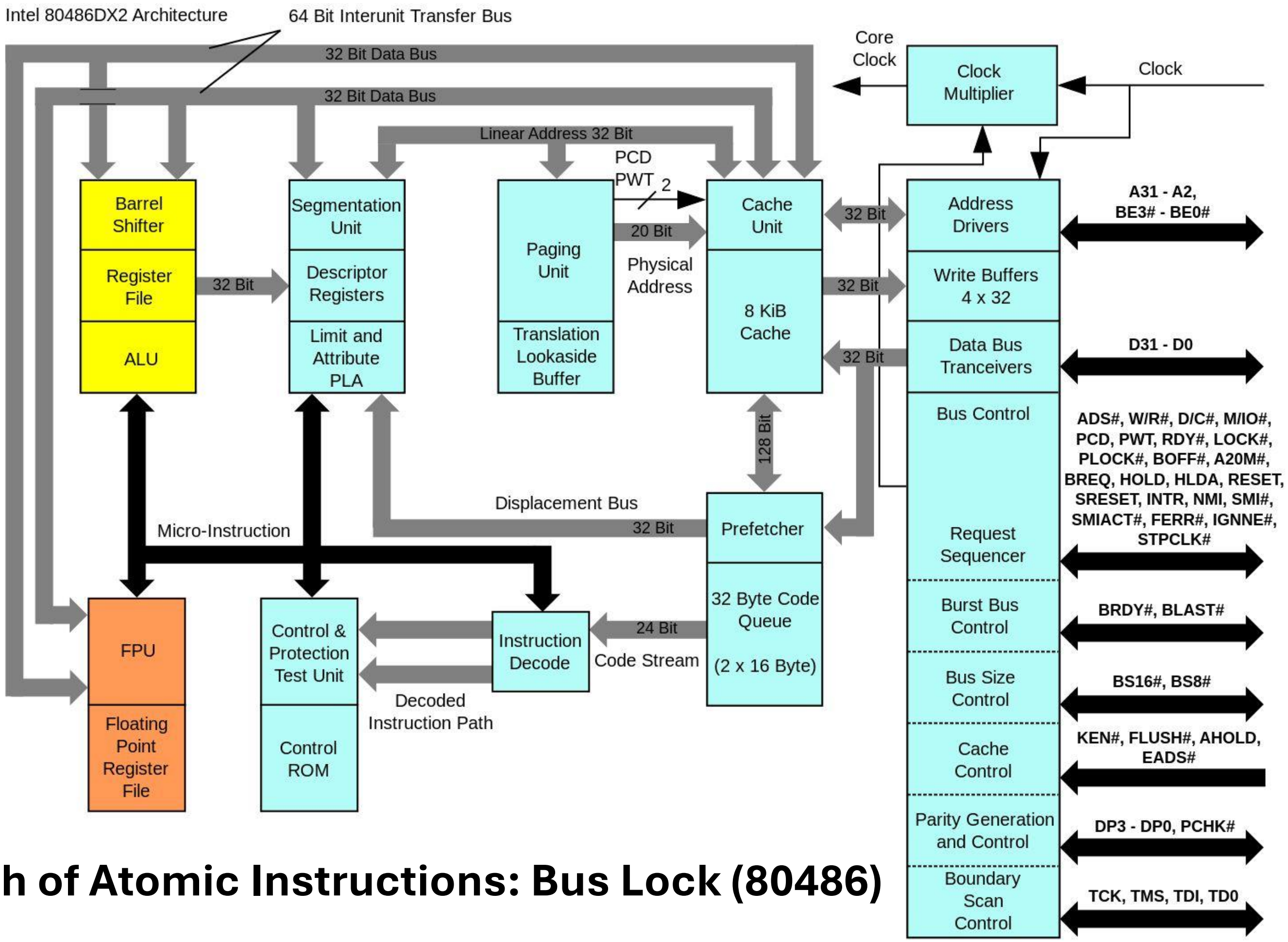
Drawbacks of Spinlocks

- Threads on other processors are spinning idly while only one thread is in the critical section.
  - The more processors competing for the lock, the lower the efficiency.
- The thread holding the spinlock might be switched out by the operating system.
  - The OS is unaware of the thread's activity (but why can't it be?).
  - This leads to 100% resource waste.
  - Example:

    https://github.com/xinliulab/COP4610_Operating_Systems/blob/main/Lecture_6_Mutual_Exclusion/ex_6_spin_scalability.c
- Spinning on a shared variable triggers cache synchronization between processors, increasing latency.

Drawbacks of Spinlocks (Cont.)

- Spinning on a shared variable triggers cache synchronization between processors, increasing latency.



The Birth of Atomic Instructions: Bus Lock (80486)

Use Cases for Spinlocks

- The critical section is rarely "contended."
- Thread context switching is prohibited while holding a spinlock.
- **Use case**
  - Concurrent data structures in the operating system kernel (short critical sections).
  - The operating system can disable interrupts and preemption, ensuring that the lock holder can release the lock in a very short time.

Takeaways

- Non-atomic nature of operations (inability to read and write simultaneously) is the root cause of lock failures or inefficiency.

- Software-based mutual exclusion is unreliable and can still fail.

- Hardware-based mutual exclusion is reliable because it eliminates concurrency.