Mutual Exclusion: Primitives and Implementation Considerations
Too Much Milk: Lessons

- Software solution (Peterson’s algorithm) works, but it is unsatisfactory
  - Solution is complicated; proving correctness is tricky even for the simple example
  - While thread is waiting, it is consuming CPU time
  - Asymmetric solution exists for 2 processes.

- How can we do better?
  - Use hardware features to eliminate busy waiting
  - Define higher-level programming abstractions to simplify concurrent programming
If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}

Thread 2
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}

Answer:
A. 0
B. 1
C. 2
D. More than 2
Schedules/Interleavings

- Model of concurrent execution
- Interleave statements from each thread into a single thread
- If any interleaving yields incorrect results, some synchronization is needed

If \( X = 0 \) initially, \( X = 1 \) at the end. WRONG result!
Locks fix this with Mutual Exclusion

```java
void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}
```

- Mutual exclusion ensures only safe interleavings

  - *When is mutual exclusion too safe?*
Introducing Locks

- **Locks** – implement mutual exclusion
  - Two methods
    - Lock::Acquire() – wait until lock is free, then grab it
    - Lock::Release() – release the lock, waking up a waiter, if any

- With locks, too much milk problem is very easy!
  - Check and update happen as one unit (exclusive access)

```cpp
Lock::Acquire();
if (!noMilk) {
    buy milk;
}
Lock::Release();
Lock::Acquire();
x++;
Lock::Release();
```

How can we implement locks?
How to think about synchronization code

- Every thread has the same pattern
  - Entry section: code to attempt entry to critical section
  - Critical section: code that requires isolation (e.g., with mutual exclusion)
  - Exit section: cleanup code after execution of critical region
  - Non-critical section: everything else

- There can be multiple critical regions in a program
  - Only critical regions that access the same resource (e.g., data structure) need to synchronize with each other

```c
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```
The correctness conditions

- **Safety**
  - Only one thread in the critical region

- **Liveness**
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

- **Bounded waiting**
  - A thread that enters the entry section enters the critical section within some bounded number of operations.

- **Failure atomicity**
  - It is OK for a thread to die in the critical region
  - Many techniques do not provide failure atomicity

```c
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```
Read-Modify-Write (RMW)

- Implement locks using read-modify-write instructions
  - As an atomic and isolated action
    1. read a memory location into a register, **AND**
    2. write a new value to the location
  - Implementing RMW is tricky in multi-processors
    - Requires cache coherence hardware. Caches snoop the memory bus.

- Examples:
  - **Test&set instructions (most architectures)**
    - Reads a value from memory
    - Write “1” back to memory location
  - **Compare & swap (a.k.a. cmpxchg on x86)**
    - Test the value against some constant
    - If the test returns true, set value in memory to different value
    - Report the result of the test in a flag
    - if [addr] == r1 then [addr] = r2;
  - **Double Compare & Swap (68000)**
    - Variant: if [addr1] == r1 then [addr2] = r2
  - **Exchange, locked increment, locked decrement (x86)**
  - **Load linked/store conditional (PowerPC, Alpha, MIPS)**
Implementing Locks with Test&set

int lock_value = 0;
int* lock = &lock_value;

Lock::Acquire() {
    while (test&set(lock) == 1)  
        ; //spin
}

Lock::Release() {
    *lock = 0;
}

- If lock is free (lock_value == 0), then test&set reads 0 and sets value to 1 ➞ lock is set to busy and Acquire completes
- If lock is busy, the test&set reads 1 and sets value to 1 ➞ no change in lock’s status and Acquire loops
- Does this lock have bounded waiting?
Locks and Busy Waiting

- **Busy-waiting:**
  - Threads consume CPU cycles while waiting
  - Low latency to acquire

- **Limitations**
  - Occupies a CPU core
  - What happens if threads have different priorities?
    - Busy-waiting thread remains runnable
    - If the thread waiting for a lock has higher priority than the thread occupying the lock, then?
    - Ugh, I just wanted to lock a data structure, but now I’m involved with the scheduler!
  - What if programmer forgets to unlock?

```cpp
Lock::Acquire() {
    while (test&set(lock) == 1) // spin
}
```
Remember to always release locks

- **Java provides a convenient mechanism.**

```java
import java.util.concurrent.locks.ReentrantLock;

public static final aLock = new ReentrantLock();

aLock.lock();
try {
    ...
} finally {
    aLock.unlock();
}
return 0;
```
Remember to always release locks

- Java also has implicit locks:

```java
synchronized void method(void) {
    XXX
}
```

is short for

```java
void method(void) {
    synchronized(this) {
        XXX
    }
}
```

is short for

```java
void method(void) {
    this.l.lock();
    try {
        XXX
    } finally {
        this.l.unlock();
    }
```
Cheaper Locks with Cheaper busy waiting

Using Test&Set

```cpp
Lock::Acquire() {
    while (test&set(lock) == 1);
}
```

With busy-waiting

```cpp
Lock::Release() {
    *lock = 0;
}
```

```cpp
Lock::Acquire() {
    while(1) {
        if (test&set(lock) == 0) break;
        else sleep(1);
    }
}
```

With voluntary yield of CPU

```cpp
Lock::Release() {
    *lock = 0;
}
```

What is the problem with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Messes up interrupt handling
What happens to lock variable’s cache line when different cpu’s contend for the same lock?

CPU A
```c
while(test&set(lock));
// in critical region
```

CPU B
```c
while(test&set(lock));
```

Load can stall

Main Memory

0xF0 lock: 1
0xF4 ...

Line bounces between caches
Cheap Locks with Cheap busy waiting

Using Test&Test&Set

Lock::Acquire()
{
    while (test&set(lock) == 1);
}

Busy-wait on in-memory copy

Lock::Release()
{
    *lock = 0;
}

Busy-wait on cached copy

Lock::Acquire()
{
    while(1) {
        while (*lock == 1); // spin just reading
        if (test&set(lock) == 0) break;
    }
}

What is the problem with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work
What happens to lock variable’s cache line when different cpu’s contend for the same lock?

CPU A
// in critical region

lock: 1
...

CPU B

while(*lock):
if(test&set(lock)) brk;

Main Memory

0xF0 lock: 1
0xF4 ...

L1
L2
What happens to lock variable’s cache line when different cpu’s contend for the same lock?

CPU A
// in critical region
*lock = 0

L1
lock: 0 ...
L2
lock: 0 ...

CPU B
while(*lock);
if(test&set(lock)) brk;

L1
lock: 0 ...
L2
lock: 0 ...

Main Memory
0xF0 lock: 0
0xF4 ...
0xF0 lock: 1
0xF4 ...
Implementing Locks: Summary

- Locks are higher-level programming abstraction
  - Mutual exclusion can be implemented using locks

- Lock implementation generally requires some level of hardware support
  - Details of hardware support affects efficiency of locking

- Locks can busy-wait, and busy-waiting cheaply is important
  - Soon come primitives that block rather than busy-wait
Best Practices for Lock Programming (So Far...)

- When you enter a critical region, check what may have changed while you were spinning
  - Did Jill get milk while I was waiting on the lock?
- Always unlock any locks you acquire
Implementing Locks without Busy Waiting (blocking)
Using Test&Set

Lock::Acquire() {
    while (test&set(lock) == 1) 
        ; // spin
}

With busy-waiting

Lock::Release() {
    *lock := 0;
}

Lock::Switch() {
    q_lock = 0;
    pid = schedule();
    if(waited_on_lock(pid))
        while(test&set(q_lock)==1)
            dispatch pid
}

Lock::Acquire() {
    if (test&set(q_lock) == 1) {
        Put TCB on wait queue for lock;
        Lock::Switch(); // dispatch thread
    }
}

Without busy-waiting, use a queue

Lock::Release() {
    if (wait queue is not empty) {
        Move 1 (or all?) waiting threads to ready
        queue;
    }
    *q_lock = 0;

Must only 1 thread be awakened?
Implementing Locks: Summary

- Locks are higher-level programming abstraction
  - Mutual exclusion can be implemented using locks

- Lock implementations have 2 key ingredients:
  - Hardware instruction that does atomic read-modify-write
    - Uni- and multi-processor architectures
  - Blocking mechanism
    - Busy waiting, or
    - Block on a scheduler queue in the OS

- Locks are good for mutual exclusion but weak for coordination, e.g., producer/consumer patterns.
Why Locks are Hard (Preview)

- **Coarse-grain locks**
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

- **Fine-grain locks**
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
    LOCK(s);
    LOCK(d);
    tmp = s.remove(key);
    d.insert(key, tmp);
    UNLOCK(d);
    UNLOCK(s);
}

Thread 0
move(a, b, key1);
move(b, a, key2);

Thread 1
DEADLOCK!