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
Concurrent Quiz

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

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

### 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)

```
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

```
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

```
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 (68000)
    - 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;
  - Exchange, locked increment, locked decrement (x86)
  - Load linked/store conditional (PowerPC,Alpha, MIPS)

Implementing Locks with Test&set

```c
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

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

- **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?

Remember to always release locks

- **Java provides convenient mechanism.**
  ```java
  import java.util.concurrent.locks.ReentrantLock;
  public static final aLock = new ReentrantLock();
  aLock.lock();
  try {
      ...
  } finally {
      aLock.unlock();
  }
  return 0;
  ```
Cheaper Locks with Cheaper busy waiting
Using Test&Set

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

With busy-waiting

Lock::Release()
*lock = 0;

With voluntary yield of CPU

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

Test & Set with Memory Hierarchies

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

Load can stall

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

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

0xF0 lock: 1
0xF4 ...

Main Memory

L1
lock: 1 ...

L2
lock: 1 ...

L1
...

L2
...

14
### Cheap Locks with Cheap busy waiting

#### Using Test&Test&Set

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

- **Busy-wait on in-memory copy**
- **Busy-wait on cached copy**

#### What is the problem with this?
- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work

### Test & Set with Memory Hierarchies

**What happens to lock variable's cache line when different cpu's contend for the same lock?**

---

**CPU A**

// in critical region

```
lock: 1
...
L1
```

**CPU B**

```
while(*lock):
    if(test&set(lock))lock:
lock: 1
...
L1
```

**Main Memory**

```
0xF0 lock: 1
0xF4 ...
L2
```

---
Test & Set with Memory Hierarchies

What happens to lock variable's cache line when different CPU's contend for the same lock?

- CPU A
  - // in critical region
  - *lock = 0

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

- L1
  - lock: 0
  - ...

- L2
  - lock: 0
  - ...

Main Memory
- 0xF0 lock: 0
- 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
**Implementing Locks without Busy Waiting (blocking)**

**Using Test&Set**

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

With busy-waiting

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

Without busy-waiting, use a queue

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

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

---

**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
  - Atomic read-modify-write instructions
    - Uni- and multi-processor architectures

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

<table>
<thead>
<tr>
<th>Coarse-grain locks</th>
<th>Fine-grain locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple to develop</td>
<td>Greater concurrency</td>
</tr>
<tr>
<td>Easy to avoid deadlock</td>
<td>Greater code complexity</td>
</tr>
<tr>
<td>Few data races</td>
<td>Potential deadlocks</td>
</tr>
<tr>
<td>Limited concurrency</td>
<td>Not composable</td>
</tr>
<tr>
<td></td>
<td>Potential data races</td>
</tr>
<tr>
<td></td>
<td>Which lock to lock?</td>
</tr>
</tbody>
</table>

// 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);
}

// 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
move(a, b, key1);
move(b, a, key2);

DEADLOCK!