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

Concurrency Quiz

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

If X==0 initially, X == 1 at the end. WRONG result!

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

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)

Introducing Locks

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

```plaintext
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

```plaintext
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
  - Double Compare & Swap (68000)
    - Variant: if [addr1] == r1 then [addr2] = r2
- Load linked/store conditional (PowerPC, Alpha, MIPS)

### Implementing Locks with Test&set

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

Lock::Acquire() {
  while (test&set(lock) == 1) // spin
}
Lock::Release() {
  *lock = 0;
}
```

### 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
  - What if programmer forgets to unlock?

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

- Java also has implicit locks:
  - synchronized void method(void) {
    XXX
  }
  is short for
  - void method(void) {
    synchronized(this) {
      XXX
    }
  }

Some cheaper locks with cheaper busy waiting:

Using Test&Set

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

With busy-waiting

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

With voluntary yield of CPU

```java
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

Cheap Locks with Cheap busy waiting

Using Test&Test&Set

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

Busy-wait on in-memory copy

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

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
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

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

With busy-waiting

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

```c
Lock::Switch() {
    delay(max_delay, max_delay); // delay thread
    *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)

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

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

DEADLOCK!