Review: Too Much Milk Solution #3

- Here is a possible two-note solution:

```c
Thread A
leave note A;
while (note B) {\X
    if (noNote A) {\Y
        do nothing;
    }
    if (noMilk) {
        buy milk;
    }
    if (noMilk) {
    }
    remove note A;
}
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit

- At X:
  - If no note B, safe for A to buy,
  - Otherwise wait to find out what will happen

- At Y:
  - If no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

---

Case 1

- "leave note A" happens before "if (noNote A)"

```c
leave note A;
while (note B) {\X
    do nothing;
}\
if (noNote A) {\Y
    if (noMilk) {
        buy milk;
    }
    remove note B;
}
if (noMilk) {
    buy milk;
}
remove note A;
```

---

Case 1

- "leave note A" happens before "if (noNote A)"

```c
leave note B;
while (note B) {\X
    do nothing;
}\
if (noNote A) {\Y
    if (noMilk) {
        buy milk;
    }
    remove note B;
}
if (noMilk) {
    buy milk;
}
remove note A;
```
Case 1

- "leave note A" happens before "if (noNote A)"

```
leave note A;
while (note B) {
    do nothing;
}

Wait for note B to be remove

if (noMilk) {
    buy milk;
} else {
    remove note A;
}
```

Case 2

- "if (noNote A)" happens before "leave note A"

```
leave note A;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
} else {
    remove note B;
}

leave note B;
if (noMilk) {
    while (note B) {
        do nothing;
    }
} else {
    remove note B;
}
```

Wait for note B to be remove

```
if (noMilk) {
    buy milk;
} else {
    remove note A;
}
```

Case 2

- "if (noNote A)" happens before "leave note A"

```
leave note A;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
} else {
    remove note B;
}
```

```
leave note B;
if (noMilk) {
    while (note B) {
        do nothing;
    }
} else {
    remove note B;
}
```

```
if (noMilk) {
    buy milk;
} else {
    remove note A;
}
```
Review: Solution #3 discussion

- Our solution protects a single “Critical-Section” piece of code for each thread:
  
  ```java
  if (noMilk) {
    buy milk;
  }
  ```

- Solution #3 works, but it’s really unsatisfactory
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A’s code is different from B’s – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called “busy-waiting”

  There’s a better way
  - Have hardware provide higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support

Too Much Milk: Solution #4

- Suppose we have some sort of implementation of a lock
  - `lock.Acquire()` – wait until lock is free, then grab
  - `lock.Release()` – Unlock, waking up anyone waiting
  - These must be atomic operations – if two threads are waiting for the lock and both see it’s free, only one succeeds to grab the lock

- Then, our milk problem is easy:
  
  ```java
  milklock.Acquire();
  if (nomilk) buy milk;
  milklock.Release();
  ```

- Once again, section of code between `Acquire()` and `Release()` called a “Critical Section”
- Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  - Skip the test since you always need more ice cream ;-)
How to Implement Locks?

- **Lock**: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - Important idea: all synchronization involves waiting
    - Should sleep if waiting for a long time
- Atomic Load/Store: get solution like Milk #3
  - Pretty complex and error prone
- Hardware Lock instruction
  - Is this a good idea?
  - What about putting a task to sleep?
    - How do you handle the interface between the hardware and scheduler?
      - Complexity!
    - Done in the Intel 432 – each feature makes HW more complex and slow

Naïve use of Interrupt Enable/Disable

How can we build multi-instruction atomic operations?

- Recall: dispatcher gets control in two ways.
  - Internal: Thread does something to relinquish the CPU
  - External: Interrupts cause dispatcher to take CPU
- On a uniprocessor, can avoid context-switching by:
  - Avoiding internal events (although virtual memory tricky)
  - Preventing external events by disabling interrupts

Consequently, naïve Implementation of locks:

```c
LockAcquire { disable Ints; }  
LockRelease { enable Ints; }
```

Naïve use of Interrupt Enable/Disable: Problems

Can’t let user do this! Consider following:

```c
LockAcquire();  
While(TRUE) {}  
```

Real-Time system—no guarantees on timing!
- Critical Sections might be arbitrarily long
- What happens with I/O or other important events?
- “Reactor about to meltdown. Help!”

Better Implementation of Locks by Disabling Interrupts

Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int value = FREE;  

Acquire() {  
  disable interrupts;  
  if (value == BUSY) {  
    put thread on wait queue;  
    Go to sleep();  
    // Enable interrupts?  
  } else {  
    value = BUSY;  
  }  
  enable interrupts;  
}  

Release() {  
  disable interrupts;  
  if (anyone on wait queue) {  
    take thread off wait queue  
    Place on ready queue;  
  } else {  
    value = FREE;  
  }  
  enable interrupts;  
}
```
New Lock Implementation: Discussion

• Why do we need to disable interrupts at all?
  – Avoid interruption between checking and setting lock value.
  – Otherwise two threads could think that they both have lock.

```c
Acquire()
{
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

• Note: unlike previous solution, the critical section (inside `Acquire()`) is very short.
  – User of lock can take as long as they like in their own critical section:
    doesn’t impact global machine behavior.
  – Critical interrupts taken in time!

Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?

```c
Acquire()
{
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

How to Re-enable After Sleep()??

• In scheduler, since interrupts are disabled when you call sleep:
  – Responsibility of the next thread to re-enable ints.
  – When the sleeping thread wakes up, returns to acquire and re-enables interrupts.

```plaintext
Thread A
- disable ints
  - sleep
  - context
  - switch
  - sleep return
  - enable ints
  - context
  - switch

Thread B
- sleep return
  - enable ints
  - context
  - switch
  - sleep
```

Atomic Read-Modify-Write Instructions

• Problems with previous solution:
  – Can’t give lock implementation to users.
  – Doesn’t work well on multiprocessor.
    » Disabling interrupts on all processors requires messages and would be very time consuming.

• Alternative: atomic instruction sequences
  – These instructions read a value and write a new value atomically.
  – Hardware is responsible for implementing this correctly.
    » on both uniprocessors (not too hard).
    » and multiprocessors (requires help from cache coherence protocol).
  – Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors.
Examples of Read-Modify-Write

- **test&set** (&address) {
  /* most architectures */
  result = M[address]; /* return result from "address" and */
  M[address] = 1; /* set value at "address" to 1 */
  return result;
}

- **swap** (address, register) {
  /* x86 */
  temp = M[address]; /* swap register's value to */
  M[address] = register; /* value at "address" */
  register = temp;
}

- **compare&swap** (address, reg1, reg2) {
  /* 68000 */
  if (reg1 == M[address]) {
    M[address] = reg2;
    return success;
  } else {
    return failure;
  }
}

Implementing Locks with test&set

- Another flawed, but simple solution:
  ```
  int value = 0; // Free
  Acquire() {
    while (test&set(value)); // while busy
    Release() {
      value = 0;
    }
  }
  ```

  Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy
    It returns 0 so while exits
  - If lock is busy, test&set reads 1 and sets value=1 (no change)
    It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock

  - **Busy-Waiting**: thread consumes cycles while waiting

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value
  ```
  int guard = 0;
  int value = FREE;
  ```

  Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
      put thread on wait queue;
      go to sleep(); & guard = 0;
    } else {
      value = BUSY;
      guard = 0;
    }
  }

  Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
    } else {
      value = FREE;
      guard = 0;
    }
  }

  Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Problem: Busy-Waiting for Lock

- Positives for this solution
  - Machine can receive interrupts
  - User code can use this lock
  - Works on a multiprocessor

- Negatives
  - This is very inefficient as thread will consume cycles waiting
  - Waiting thread may take cycles away from thread holding lock (no one wins!)
  - **Priority Inversion**: If busy-waiting thread has higher priority than thread holding lock => no progress!

- Priority Inversion problem with original Martian rover
- For semaphores and monitors, waiting thread may wait for an arbitrary long time!
  - Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
  - Homework/exam solutions should avoid busy-waiting!
Locks using Interrupts vs. test&set

Compare to “disable interrupt” solution

```c
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    Go to sleep();
  } else {
    value = BUSY;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  guard = 0;
  enable interrupts;
}
```

Basically replace
- disable interrupts → while (test&set(guard));
- enable interrupts → guard = 0;

Recap: Locks using interrupts

```c
int value = 0;

Acquire() {
  disable interrupts;
  if (value == 1) {
    put thread on wait queue;
    go to sleep() //??
  } else {
    value = 1;
    guard = 0;
  }
  enable interrupts;
}

Release() {
  disable interrupts;
  if anyone on wait queue {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = 0;
    guard = 0;
  }
  enable interrupts;
}
```

Recap: Locks using test & wait

```c
int guard = 0;
int value = 0;

Acquire() {
  while(test&set(guard));
  if (value == 1) {
    put thread on wait queue;
    go to sleep() & guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}

Release() {
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = 0;
    guard = 0;
  }
}
```

Recap: Locks using interrupts

```c
int value = 0;

Acquire() {
  // Short busy-wait time
  disable interrupts;
  if (value == 1) {
    put thread on wait queue;
    go to sleep() //??
  } else {
    value = 1;
    enable interrupts;
  }
}

Release() {
  // Short busy-wait time
  disable interrupts;
  if anyone on wait queue {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = 0;
    enable interrupts;
  }
}
```

If one thread in critical section, no other activity (including OS) can run!

Threads waiting to enter critical section busy-wait

Recap: Locks using test & wait

```c
int guard = 0;
int value = 0;

Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait queue;
    go to sleep() & guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}

Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = 0;
    guard = 0;
  }
}
```

If one thread in critical section, no other activity (including OS) can run!

Threads waiting to enter critical section busy-wait

Recap: Locks using test & wait

```c
int guard = 0;
int value = 0;

Acquire() {
  // Short busy-wait time
  while(test&set(guard));
  if (value == 1) {
    put thread on wait queue;
    go to sleep() & guard = 0;
  } else {
    value = 1;
    guard = 0;
  }
}

Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = 0;
    guard = 0;
  }
}
```

Administrivia

- Midterm Thursday 9/28 6:30-8PM
- Project 1 Design Document due today
- Project 1 Design reviews upcoming
  - High-level discussion of your approach
    » What will you modify?
    » What algorithm will you use?
    » How will things be linked together, etc.
    » Do not need final design (complete with all semicolons!)
  - You will be asked about testing
    » Understand testing framework
    » Are there things you are doing that are not tested by tests we give you?
- Do your own work!
  - Please do not try to find solutions from previous terms
  - We will be on the look out for anyone doing this…today
Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - What is right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible

- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs

- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents some ways of structuring sharing

Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX

- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation
  - Note that P() stands for “proberen” (to test) and V() stands for “verhogen” (to increment) in Dutch

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can’t read or write value, except to set it initially
  - Operations must be atomic
    - Two P’s together can’t decrement value below zero
    - Similarly, thread going to sleep in P won’t miss wakeup from V – even if they both happen at same time

- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:
Two Uses of Semaphores

Mutual Exclusion (initial value = 1)
• Also called “Binary Semaphore”.
• Can be used for mutual exclusion:
  semaphore.P();
  // Critical section goes here
  semaphore.V();

Scheduling Constraints (initial value = 0)
• Allow thread 1 to wait for a signal from thread 2, i.e., thread 2 schedules thread 1 when a given event occurs
• Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:

  Initial value of semaphore = 0
  ThreadJoin { 
    semaphore.P();
  }
  ThreadFinish {
    semaphore.V();
  }

Correctness constraints for solution

• Correctness Constraints:
  – Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  – Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  – Only one thread can manipulate buffer queue at a time (mutual exclusion)
• Remember why we need mutual exclusion
  – Because computers are stupid
  – Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
• General rule of thumb: Use a separate semaphore for each constraint
  – Semaphore fullBuffers; // consumer’s constraint
  – Semaphore emptyBuffers; // producer’s constraint
  – Semaphore mutex; // mutual exclusion

Full Solution to Bounded Buffer

Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
  emptySlots.P(); // Wait until space
  mutex.P(); // Wait until machine free
  Enqueue(item);
  mutex.V();
  fullSlots.V(); // Tell consumers there is more coke
}

Consumer() {
  fullSlots.P(); // Check if there’s a coke
  mutex.P(); // Wait until machine free
  item = Dequeue();
  mutex.V();
  emptySlots.V(); // tell producer need more
  return item;
}
**Discussion about Solution**

**Why asymmetry?**

- **Producer** does: `emptySlots.P()`, `fullSlots.V()`
- **Consumer** does: `fullSlots.P()`, `emptySlots.V()`

**Discussion about Solution (cont’d)**

**Is order of P’s important?**
- Yes! Can cause deadlock

**Is order of V’s important?**
- No, except that it might affect scheduling efficiency

What if we have 2 producers or 2 consumers?

**Discussion about Solution (cont’d)**

**Motivation for Monitors and Condition Variables**

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores
  - Problem is that semaphores are dual purpose:
    - They are used for both mutex and scheduling constraints
    - Example: the fact that flipping of P’s in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?

- Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints

- Definition: **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

**Monitor with Condition Variables**

- **Lock**: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free

- **Condition Variable**: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can’t wait inside critical section
Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

  ```c
  Lock lock;
  Queue queue;

  AddToQueue(item) {
    lock.Acquire(); // Lock shared data
    queue.enqueue(item); // Add item
    lock.Release(); // Release Lock
  }

  RemoveFromQueue() {
    lock.Acquire(); // Lock shared data
    item = queue.dequeue(); // Get next item or null
    lock.Release(); // Release Lock
    return(item); // Might return null
  }
  ```

- Not very interesting use of “Monitor”
  - It only uses a lock with no condition variables
  - Cannot put consumer to sleep if no work!

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

  ```c
  Lock lock;
  Condition dataready;
  Queue queue;

  AddToQueue(item) {
    lock.Acquire(); // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal any waiters
    lock.Release(); // Release Lock
  }

  RemoveFromQueue() {
    lock.Acquire(); // Get Lock
    while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
    }
    item = queue.dequeue(); // Get next item
    lock.Release(); // Release Lock
    return(item);
  }
  ```

Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone
- **Condition Variable**: a queue of threads waiting for something inside a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can’t wait inside critical section
- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters
- Rule: Must hold lock when doing condition variable ops!
  - In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

Summary (1/2)

- Important concept: **Atomic Operations**
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, compare&swap, conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    » Shouldn’t disable interrupts for long
    » Shouldn’t spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
Summary (2/2)

- **Semaphores:** Like integers with restricted interface
  - Two operations:
    » **P():** Wait if zero; decrement when becomes non-zero
    » **V():** Increment and wake a sleeping task (if exists)
    » Can initialize value to any non-negative value
  - Use separate semaphore for each constraint

- **Monitors:** A lock plus one or more condition variables
  - Always acquire lock before accessing shared data
  - Use condition variables to wait inside critical section
  » Three Operations: **Wait(), Signal(), and Broadcast()**