CMPSCI 377 Operating Systems **Fall 2007**

Lecture 9: October 18

Lecturer: Emery Berger *Scribe: Andrew Huang*

9.1 Synchronization

9.1.1 Consistency

Threads must ensure **consistency**. Otherwise, threads may result in a **race condition** leading to a non-deterministic result. A race is when the result is dependent the order in which the threads finish.

For example:
Thread A says x = 1, and if (x == 2), then print 'What?'
Thread B says x = 2.
What is x?

Threads require **synchronization operations** for this to be answered.

9.2 'The *too much milk* problem

The *too much milk* problem

time	You	Your roommate
3:00	Arrive Home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:30	Arrive home, put milk in fridge	
3:35		Arrive at grocery
3:40		Buy milk
3:45	Arrive home, put milk in fridge	
3:50		Uh oh!

This is why we need to synchronize activities!

Terminology

Definition 9.1 ***Mutual exclusion (mutex)*** *prevents multiple threads from entering. The* ***critical section*** *is code that only one thread can execute at a time. A* ***lock*** *is a mechanism for mutual exclusion.*

- *Lock on entering critical section, accessing shared data*
- *Unlock when complete*
- *Wait if locked*

9.2.1 Solving the Too Much Milk Problem

Correctness properties

- **Only** one person buys milk
  - **Safety:** "nothing bad happens"
- **Someone** buys milk if you need to
  - **Progress:** "something good eventually happens"

First: use atomic loads stores as building blocks

- "Leave a note" (lock)
- "Remove a note" (unlock)
- "Don't buy milk if there's a note" (wait)

9.2.2 Too Much Milk: Attempt 1

thread A

```
if(no milk && no note)
   leave note
   buy milk
   remove note
```

thread B

```
if (no milk && no note)
   leave note
   buy note
   remove note
```

! If both threads switch after reading the if statement, then both will buy milk.

9.2.3 Too Much Milk: Attempt 2

Idea: use *labeled* notes

thread A

```
leave note A
if (no note B)
   if (no milk)
      buy milk
remove note A
```

thread B

```
leave note B
if (no note A)
   if (no milk)
      buy milk
remove note B
```

! If both threads switch after leaving a note, neither will buy milk.

9.2.4 Too Much Milk: Attempt 3

Idea: *wait* for the right note

thread A	**thread B**
leave note A	leave note B
while (note B)	if (no note A)
do nothing	if (no milk)
if (no milk)	buy milk
buy milk	remove note B
remove note A	

Possibility 1: thread A runs first, then thread B.
thread A will buy milk. thread B will see there is milk, and therefore not buy any more milk.

Possibility 2: thread B runs first, then thread A.
thread B will buy milk. thread A will see there is milk, and therefore not buy any more milk.

Possibility 3: Interleaved - A waits buys
thread A and thread B can leave a note, but thread A will wait for thread B to finish before moving on.
thread B will not buy milk because note A exists, and finish the thread.
thread A will continue and buy milk.

Possibility 4: Interleaved - A waits, B buys
thread B will leave a note, and see there is no note A.
A can start and see there is a note B, and wait for B to finish.
B will continue to buy milk if there no milk, and then finish before thread A will continue.
thread A will not buy milk because thread B just bought milk.

So attempt 3 is a correct solution because it preserves the desired properties:

- **Safety:** we only buy milk once
- **Progress:** we always buy milk

but..

9.2.5 Problems with this solution

Complicated

- Difficult to convince outselves that it works

Asymmetrical

- Threads A & B are different
- Adding more threads = different code for each thread

Poor utilization

- **Busy waiting** consumes CPU, no useful work
  - time could be better spent on other work

Possibly non-portable

- Relies on atomicity of loads & stores
  - atomicity is "all of nothing"

9.3 Language Support

Synchronization (making an area of code atomic) is complicated Better way - provide *language-level* support

- Higher-level approach
- Hide gory details in runtime system

Increasingly high-level approaches (pthread supplies all these):

- **Locks, Atomic Operations**
- **Semaphores** - generalized locks
- **Monitors** - tie shared data to synchronization

9.4 Locks

Provide mutual exclusion to shared data via two atomic routines:

- **acquire** - wait for lock, then take it
- **release** - unlock, wake up waiters

Rules:

- Acquire lock *before* accessing shared data
- Release lock afterwards
- Lock initially released

9.4.1 Pthreads Syntax

POSIX standard for C/C++ Mutual exclusion locks

- Ensures only one thread in critical section

pthread_mutex_init(l);
..
pthread_mutex_lock(l);
update data; / critical section */*
pthread_mutex_unlock(l);

9.4.2 Pthreads API

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Threads attributes objects
pthread_mutex_	Mutexes

9.4.3 Too Much Milk: Locks

thread A	**thread B**
p_m_lock(l)	p_m_lock(l)
if (no milk)	if (no milk)
buy milk	buy milk
p_m_lock(l)	p_m_lock(l)

The code in both threads is exactly the same. Clean, symmetric - but how do we implement it? Hardware support prevents both locks from happening at the same time. Dekker's algorithm allows mutual exlusion.

9.4.4 Implementing Locks

Requires hardware support (in general) Can build on atomic operations:

- **Load/Store**
- **Disable interrupts**
  - Uniprocessors only, this does not work on multiprocessors
  - Says to OS, "Shut up, don't switch me out or interrupt me!"
- **Test & Set, Compare & Swap**

9.4.5 Disabling Interrupts

"I want a notification when some time elapses, a hardware signal called an interrupt, and time is up." Disabling interrupts prevent scheduler from switching threads in middle of critical sections

- Ignores quantum expiration (timer interrupt)
- No handling I/O operations
  - (Don't make I/O calls in critical section!)

In cooperative multithreading, a thread can call yield and say it is ok to switch out

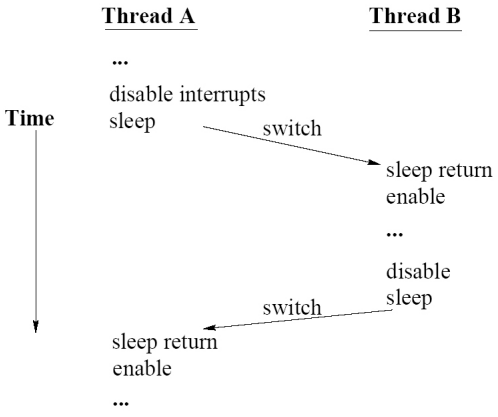
```
Class ={
  private int value;
  private Queue q;

  Lock () {
    value = 0; q = empty;
  }

  public void acquire () {
    disable interrupts;
    if (value == BUSY) {
    add this thread to q;
    enable interrupts;
    sleep();
    } else {
    value = BUSY;
    }
    enable interrupts;
  }

  public void release () {
    disable interrupts;
    if (q not empty) {
    thread t = q.pop();
    put t on ready queue;
    }
    value = FREE;
    enable interrupts;
  }
}
```

9.4.6 Locks via Disabling Interrupts



9.4.7 Barrier

X = 1
Y = 2
Z = X
Could run Z=x, and then X=1, because reads can come first but..

Barrier prevents misordering
X = 1
BARRIER //don't execute until above is done
Y = 2
Z = X

9.5 Summary

Communication between threads:via shared variables **Critical sections** = regions of code that modify or access shared variables Must be protected by synchronization primitives that ensure **mutual exclusion**

- Loads & stores: tricky, error-prone
- Solution: high-level primitives (e.g., locks)