CS 162 Operating Systems and Systems Programming

Professor: Anthony D. Joseph

Spring 2002

Lecture 6: Synchronization

6.0 Main points

- More concurrency examples
- Synchronization primitives

6.1 A Larger Concurrent Program Example

6.1.1 ATM bank server example

Suppose we wanted to implement a server process that handles requests from an ATM network to do things like deposit and withdraw money from bank accounts.

```
BankServer() {
      while (TRUE) {
            ReceiveRequest(&op, &acctId, &amount);
            ProcessRequest(op, acctId, amount);
      }
}

ProcessRequest(op, acctId, amount) {
      if (op == deposit) Deposit(acctId, amount);
      else if …
}

Deposit(acctId, amount) {
      acct = GetAccount(acctId); /* May involve disk I/O */
      acct->balance += amount;
      StoreAccount(acct); /* Involves disk I/O */
}
```

Suppose we had a multiprocessor? Could run each invocation of ProcessRequest in a separate thread to get parallelism.

Suppose we only had one CPU. We'd still like to overlap I/O with computation. Without threads we would have to rewrite the code to look something like:

```
BankServer() {
     while (TRUE) {
          event = WaitForNextEvent();
          if (event == ATMRequest)
               StartOnRequest();
          else if (event == AcctAvail)
               ContinueRequest();
          else if (event == AcctStored)
               FinishRequest();
     }
}
```

With threads one could get overlapped I/O and computation without having to "deconstruct" the code into fragments that run when the appropriate asynchronous event has occurred.

Problem: In the threaded version shared state can get corrupted:

Thread 1 running Deposit:

```
load r1, acct->balance


add r1, amount1
store r1, acct->balance
```

Thread 2 running Deposit:

```
load r1, acct->balance
add r1, amount2
store r1, acct->balance
```

The dispatcher can choose to run each thread to completion or until it blocks. It can time-slice in whatever size chunks it wants. If running on a multiprocessor then instructions may be interleaved one at-a-time.

Threaded programs must work correctly for any interleaving of thread instruction sequences.

Cooperating threaded programs are inherently non-deterministic and non-reproducible. This makes them hard to debug and maintain unless they are designed very carefully.

6.1.2 Another concurrent program example

Two threads, A and B, compete with each other; one tries to increment a shared counter, the other tries to decrement the counter.

For this example, assume that memory load and memory store are atomic, but incrementing and decrementing are **not** atomic.

Thread A

```
i = 0
while (i < 10)
  i = i + 1;
print A wins
```

Thread B

```
i = 0
while (i > -10)
  i = i - 1;
print B wins
```

Questions:

1. Who wins? Could be either.
2. Is it guaranteed that someone wins? Why not?
3. What if both threads have their own CPU, running in parallel at exactly the same speed. Is it guaranteed that it goes on forever?

In fact, if they start at the same time, with A 1/2 an instruction ahead, **B** will win quickly.

4. Could this happen on a uniprocessor?

Yes! Unlikely, but if you depend on it **not** happening, it will happen, and your system will break and it will be very difficult to figure out why.

6.2 Motivation: "Too Much Milk"

	Person A	Person B
3:00	Look in fridge. Out of milk.	
3:05	Leave for store.	

3:10	Arrive at store.	Look in fridge. Out of milk.
3:15	Buy milk.	Leave for store.
3:20	Arrive home, put milk away.	Arrive at store.
3:25		Buy milk.
3:30		Arrive home, put milk away.
		Oh no!

6.3 Definitions

Synchronization: using atomic operations to ensure cooperation between threads.

Mutual exclusion: ensuring that only one thread does a particular thing at a time. One thread doing it *excludes* the other, and vice versa.

Critical section: piece of code that only one thread can execute at once. Only one thread at a time will get into the section of code.

Lock: prevents someone from doing something.

1. Lock before entering critical section, before accessing shared data
2. Unlock when leaving, after done accessing shared data
3. Wait if locked
    - Key idea – all synchronization involves waiting.

6.4 Too Much Milk: Solution #1

What are the correctness properties for the too much milk problem?

- Never more than one person buys
- Someone buys if needed

Restrict ourselves to only use atomic load and store operations as building blocks.

Basic idea of solution #1:

1. Leave a note (kind of like "lock")
2. Remove note (kind of like "unlock")
3. don't buy if note (wait)

Solution #1:

```
if (noMilk) {
      if (noNote){
            leave Note;
            buy milk;
            remove note;
      }
}
```

Why doesn't this work? Thread can get context switched after checking milk and note, but before buying milk!

Our "solution" makes problem worse – fails only occasionally. Makes it really hard to debug. Remember, constraint has to be satisfied, independent of what the dispatcher does – timer can go off, and context switch can happen at any time.

6.5 Too Much Milk Solution #2

How about labeled notes? That way, we can leave the note before checking the milk.

Solution #2:

Thread A

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

Thread B

```
leave note B
if (noNoteA){
     if (noMilk)
          buy milk
}
remove note B
```

Possible for neither thread to buy milk; context switches at exactly the wrong times can lead each to think the other is going to buy.

Illustrates **starvation**: thread waits forever

6.6 Too Much Milk Solution #3

Solution #3:

Thread A

```
leave note A
while (note B) // X
     do nothing;
if (noMilk)
     buy milk;
remove note A
```

Thread B

```
leave note B
if (noNoteA){ // Y
     if (noMilk)
          buy milk
}
remove note B
```

Does this work? Yes. Can guarantee at X and Y that either

(i) safe for me to buy

(ii) other will buy, ok to quit

At Y: if noNote A, safe for B to buy (means A hasn't started yet)
    if note A, A is either buying, or waiting for B to quit,
        so ok for B to quit

At X: if nonote B, safe to buy
    if note B, don't know. A hangs around. Either:
        if B buys, done
    if B doesn't buy, A will.

6.7 Too Much Milk Summary

Solution #3 works, but it's really unsatisfactory:

1. Really complicated – even for this simple an example, hard to convince yourself it really works
2. A's code different than B's – what if lots of threads? Code would have to be slightly different for each thread.
3. While A is waiting, it is consuming CPU time (**busy-waiting**)

There's a better way.

1. Have hardware provide better (higher-level) primitives than atomic load and store. Examples in next lecture.
2. Build even higher-level programming abstractions on this new hardware support. For example, why not use locks as an atomic building block (how we do this in the next lecture):
    `Lock::Acquire` – wait until lock is free, then grab it
    `Lock::Release` – unlock, waking up a waiter if any

These must be atomic operations – if two threads are waiting for the lock, and both see it's free, only one grabs it!

With locks, the too much milk problem becomes really easy!

```
lock->Acquire();
if (nomilk)
      buy milk;
lock->Release();
```