Operating Systems
(1DT020 & 1TT802)

Lecture 6
Synchronization:
mutual exclusion, critical sections
April 24, 2009

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt09
Goals for Today

- Cooperating concurrent threads examples
- Need for synchronization
- Examples of valid synchronization

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)
Cooperating processes/threads

• Why allow cooperating threads?
  – People cooperate; computers help/enhance people’s lives, so computers must cooperate
  – By analogy, the non-reproducibility/non-determinism of people is a notable problem for “carefully laid plans”

• Advantage 1: Share resources
  – One computer, many users
  – One bank balance, many Automatic Teller Machines (ATMs)
    » What if ATMs were only updated at night?
  – Embedded systems (robot control: coordinate arm & hand)

• Advantage 2: Speedup
  – Overlap I/O and computation
    » Many different file systems do read-ahead
  – Multiprocessors – chop up program into parallel pieces

• Advantage 3: Modularity
  – More important than you might think
  – Chop large problem up into simpler pieces
    » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    » Makes system easier to extend
Correctness for systems with concurrent threads

• If dispatcher can schedule threads in any way, programs must work under all circumstances

• Independent Threads:
  – No state shared with other threads
  – Deterministic ⇒ Input state determines results
  – Reproducible ⇒ Can recreate Starting Conditions, I/O
  – Scheduling order doesn’t matter (if context switch works correctly)

• Cooperating Threads:
  – Shared State between multiple threads
  – Non-deterministic
  – Non-reproducible

• Non-deterministic and Non-reproducible means that bugs can be intermittent
  – Sometimes called “Heisenbugs”
ATM Bank Server

• ATM server problem:
  – Service a set of requests
  – Do so without corrupting database
  – Don’t hand out too much money
ATM bank server example

Suppose we wanted to implement a server process to handle requests from an ATM network:

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

```c
ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}
```

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

- How could we speed this up?
  - More than one request being processed at once
    » Event driven (overlap computation and I/O)
    » Multiple threads (multi-processing, or overlap computation and I/O)
Event Driven Version of ATM server

• Suppose we only had one CPU
  – Still like to overlap I/O with computation
  – Without threads, we would have to rewrite in event-driven style

• Example

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

  – What if we missed a blocking I/O step?
  – What if we have to split code into hundreds of pieces which could be blocking?
  – This technique is used for graphical programming
Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments
  - One thread per request

- Requests proceeds to completion, blocking as required:
  ```
  Deposit(acctId, amount) {
      acct = GetAccount(actId); /* May use disk I/O */
      acct->balance += amount;
      StoreAccount(acct); /* Involves disk I/O */
  }
  ```

- Unfortunately, shared state can get corrupted:
  
  Thread 1
  ```
  load r1, acct->balance
  add r1, amount1
  store r1, acct->balance
  ```

  Thread 2
  ```
  load r1, acct->balance
  add r1, amount2
  store r1, acct->balance
  ```
Review: Multiprocessing vs Multiprogramming

• What does it mean to run two threads “concurrently”?  
  – Scheduler is free to run threads in any order and interleaving: FIFO, Random, …  
  – Scheduler can choose to run each thread to completion or time-slice in big chunks or small chunks

• Also recall: Hyperthreading  
  – Possible to interleave threads on a per-instruction basis  
  – Keep this in mind for our examples (like multiprocessing)
Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn’t matter:

  Thread A  |  Thread B
  ---------  |  ---------
  x = 1;    |  y = 2;

- However, What about (Initially, y = 12):

  Thread A  |  Thread B
  ---------  |  ---------
  x = 1;    |  y = 2;
  x = y+1;  |  y = y*2;

  - What are the possible values of x?

- Or, what are the possible values of x below?

  Thread A  |  Thread B
  ---------  |  ---------
  x = 1;    |  x = 2;

  - X could be 1 or 2 (non-deterministic!)
  - Could even be 3 for serial processors:
    » Thread A writes 0001, B writes 0010.
    » Scheduling order ABABABBA yields 3!

  **Race condition (outcome of execution of concurrent threads depends on the particular order in which the processes access shared data)**
Atomic Operations

• To understand a concurrent program, we need to know what the underlying indivisible operations are!

• **Atomic Operation**: an operation that always runs to completion or not at all
  – It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  – Fundamental building block – if no atomic operations, then have no way for threads to work together

• On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

• Many instructions are not atomic
  – Double-precision floating point store often not atomic
  – VAX and IBM 360 had an instruction to copy a whole array
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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0;</td>
<td>$i = 0;</td>
</tr>
<tr>
<td>while ($i &lt; 10)</td>
<td>while ($i &gt; -10)</td>
</tr>
<tr>
<td>$i = i + 1;</td>
<td>$i = i - 1;</td>
</tr>
<tr>
<td>printf(&quot;A wins!&quot;);</td>
<td>printf(&quot;B wins!&quot;);</td>
</tr>
</tbody>
</table>

- Assume that memory loads and stores are atomic, but incrementing and decrementing are not atomic
- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?
Hand Simulation Multiprocessor Example

• Inner loop looks like this (i in memory at location \( M[i] \)):

  Thread A
  - load r1, M[i] (r1=0)
  - add r1, r1, 1 (r1=1)
  - store r1, M[i] (M[i]=1)

  Thread B
  - load r1, M[i] (r1=0)
  - sub r1, r1, 1 (r1=-1)
  - store r1, M[i] (M[i]=-1)

• Hand Simulation:
  - And we’re off. A gets off to an early start
  - B says "hmph, better go fast" and tries really hard
  - A goes ahead and writes "1"
  - B goes and writes "-1"
  - A says "HUH??? I could have sworn I put a 1 there"

• Could this happen on a uniprocessor?
  - Yes! Unlikely, but if you are depending on it not happening, it will and your system will break…
Another concurrency example: “Too Much Milk”

• “Too much milk” : Great thing about OS’s – analogy between problems in OS and problems in real life
  – Help you understand real life problems better
  – But, computers are much stupider than people

• Example: People need to coordinate:

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away</td>
</tr>
</tbody>
</table>
Definitions

• **Synchronization**: using atomic operations to ensure cooperation between threads
  – For now, only loads and stores are atomic
  – We are going to show that it is hard to build anything useful with only reads and writes (loads and stores)

• **Mutual Exclusion**: ensuring that only one thread does a particular thing at a time
  – One thread *excludes* the other while doing its task

• **Critical Section**: piece of code in which a thread can be accessing shared data. When the thread is executing in its critical section, no other thread is allowed to execute in its critical section (wrt the same shared data).
  – Critical section is the result of mutual exclusion
  – Critical section and mutual exclusion are two ways of describing the same thing
More Definitions

- **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

- For example: fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants orange juice
  - Of Course – We don’t know how to make a lock yet
Too Much Milk: Correctness Properties

• Need to be careful about correctness of concurrent programs, since non-deterministic
  – Always write down behavior first
  – Impulse is to start coding first, then when it doesn’t work, pull hair out
  – Instead, think first, then code

• What are the correctness properties for the “Too much milk” problem???
  – Never more than one person buys
  – Someone buys if needed

• Restrict ourselves to use only atomic read and write operations as building blocks
Too Much Milk: Solution #1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (attend to your other own affairs)

- Suppose a computer tries this (remember, only memory read/write are atomic):
  
  ```
  if (!noMilk) {
      if (!noNote) {
          leave Note;
          buy milk;
          remove note;
      }
  }
  ```

- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before leaving note!

- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the scheduler does!
Too Much Milk Solution #2

• How about labeled notes?
  – Now we can leave note before checking

• Algorithm looks like this:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>leave note A;</td>
<td>leave note B;</td>
</tr>
<tr>
<td>if (noNote B){</td>
<td>if(noNote A){</td>
</tr>
<tr>
<td>if (noMilk) {</td>
<td>if (noMilk) {</td>
</tr>
<tr>
<td>buy Milk;</td>
<td>buy Milk;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td>remove note A</td>
<td>remove note B;</td>
</tr>
</tbody>
</table>

• Does this work?

• Possible for neither thread to buy milk
  – Context switches at exactly the wrong times can lead each to think
    that the other is going to buy

• Really insidious:
  – **Extremely unlikely** that this would happen, but will (at the worst
    possible time!)
Too Much Milk Solution #3

• Here is a possible two-note solution:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>leave note A;</td>
<td>leave note B;</td>
</tr>
<tr>
<td>while (note B){ //X</td>
<td>if (noNote A) { //Y</td>
</tr>
<tr>
<td>do nothing;</td>
<td>if (noMilk) {</td>
</tr>
<tr>
<td>}</td>
<td>buy milk;</td>
</tr>
<tr>
<td>if (noMilk) {</td>
<td>}</td>
</tr>
<tr>
<td>buy milk;</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>remove note B;</td>
</tr>
<tr>
<td>remove note A;</td>
<td></td>
</tr>
</tbody>
</table>

• 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
Solution #3 discussion

• Our solution protects a single “Critical-Section” piece of code for each thread:

```plaintext
if (noMilk) {
    buy milk;
}
```

• Solution #3 works, but it is really unsatisfactory
  – Really complex – even for this simple 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”
  – What if the is a third thread (a 3rd room mate)?

• There is a better way
  – Have hardware provide better (higher-level) primitives than atomic load and store
  – Build even higher-level programming abstractions on this new hardware support
Too Much Milk: Solution #4

• Suppose we have some sort of implementation of a lock (more in the next lecture)
  – 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 is free, only one succeeds to grab the lock

• Then, our too much milk problem is easy:

```c
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
  – Only Critical Sections need to be accessed in a synchronized way
Where are we going with synchronization?

<table>
<thead>
<tr>
<th>Programs</th>
<th>Cooperating Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level API</td>
<td>Locks  Semaphores  Monitors  Send/Receive</td>
</tr>
<tr>
<td>Hardware</td>
<td>Load/Store  Disable Ints  Test&amp;Set  Comp&amp;Swap</td>
</tr>
</tbody>
</table>

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level
Summary

- Concurrent threads are a very useful abstraction
  - Allow transparent overlapping of computation and I/O
  - Allow use of parallel processing when available

- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent

- 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

- Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!