Synchronization 2:
Concurrency (Con’t), Mutual Exclusion,
Lock Implementation, Atomic Operations

February 7th, 2023
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Use of Threads

• Version of program with Threads (loose syntax):

```c
main() {
    ThreadFork(ComputePI, "pi.txt");
    ThreadFork(PrintClassList, "classlist.txt");
}
```

• What does `ThreadFork()` do?
  – Start independent thread running given procedure

• What is the behavior here?
  – Now, you would actually see the class list
  – This *should* behave as if there are two separate CPUs

![Diagram of CPU usage over time]
Recall: Memory Footprint for Two-Threads

• If we stopped this program and examined it with a debugger, we would see
  – Two sets of CPU registers
  – Two sets of Stacks

• Questions:
  – How do we position stacks relative to each other?
  – What maximum size should we choose for the stacks?
  – What happens if threads violate this?
  – How might you catch violations?
  – What about n>2 threads?
Recall: the Dispatch Loop

• Conceptually, the scheduling loop of the operating system looks as follows:

```c
Loop {
    RunThread();    /* Needs to exit every now and then! */
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
}
```

• This is an *infinite* loop
  – One could argue that this is all that the OS does
• Should we ever exit this loop???
  – When would that be?
Recall: Running a thread

Consider first portion: RunThread()

• How do I run a thread?
  – Load its state (registers, PC, stack pointer) into CPU
  – Load environment (virtual memory space, etc)
  – Jump to the PC

• How does the dispatcher get control back?
  – Internal events: thread returns control voluntarily
  – External events: thread gets preempted
Internal Events

• Blocking on I/O
  – The act of requesting I/O implicitly yields the CPU

• Waiting on a “signal” from other thread
  – Thread asks to wait and thus yields the CPU

• Thread executes a yield()
  – Thread volunteers to give up CPU

```c
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```
Stack for Yielding Thread

- How do we run a new thread?

```c
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
}
```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack pointer
  - Maintain isolation for each thread
What Do the Stacks Look Like?

- Consider the following code blocks:
  ```
  proc A() {
    B();
  }
  proc B() {
    while(TRUE) {
      yield();
    }
  }
  ```

- Suppose we have 2 threads:
  - Threads S and T

  Thread S's switch returns to Thread T's (and vice versa)
Saving/Restoring state (often called "Context Switch")

\[
\text{Switch}(t_{\text{Cur}}, t_{\text{New}}) \{
\text{/* Unload old thread */}
\text{TCB}[t_{\text{Cur}}].\text{regs}.r7 = \text{CPU}.r7;
\text{...}
\text{TCB}[t_{\text{Cur}}].\text{regs}.r0 = \text{CPU}.r0;
\text{TCB}[t_{\text{Cur}}].\text{regs}.sp = \text{CPU}.sp;
\text{TCB}[t_{\text{Cur}}].\text{regs}.\text{retpc} = \text{CPU}.\text{retpc}; /*return addr*/
\]

\[
\text{/* Load and execute new thread */}
\text{CPU}.r7 = \text{TCB}[t_{\text{New}}].\text{regs}.r7;
\text{...}
\text{CPU}.r0 = \text{TCB}[t_{\text{New}}].\text{regs}.r0;
\text{CPU}.sp = \text{TCB}[t_{\text{New}}].\text{regs}.sp;
\text{CPU}.\text{retpc} = \text{TCB}[t_{\text{New}}].\text{regs}.\text{retpc}; \text{return; /* Return to CPU.retpc */}
\]
Switch Details (continued)

• What if you make a mistake in implementing switch?
  – Suppose you forget to save/restore register 32
  – Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  – System will give wrong result without warning

• Can you devise an exhaustive test to test switch code?
  – No! Too many combinations and inter-leavings

• Cautionary tale:
  – For speed, Topaz kernel saved one instruction in switch()
  – Carefully documented! Only works as long as kernel size < 1MB
  – What happened?
    » Time passed, People forgot
    » Later, they added features to kernel (no one removes features!)
    » Very weird behavior started happening
  – Moral of story: Design for simplicity
How expensive is context switching?

- Switching between threads in same process similar to switching between threads in different processes, but much cheaper:
  - No need to change address space
- Some numbers from Linux:
  - Frequency of context switch: 10-100ms
  - Switching between processes: 3-4 μsec.
  - Switching between threads: 100 ns
- Even cheaper: switch threads (using “yield”) in user-space!

What we are talking about in Today’s lecture

Simple One-to-One Threading Model

Many-to-One

Many-to-Many
What happens when thread blocks on I/O?

- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch

- Thread communication similar
  - Wait for Signal/Join
  - Networking
External Events

• What happens if thread never does any I/O, never waits, and never yields control?
  – Could the ComputePI program grab all resources and never release the processor?
    » What if it didn’t print to console?
  – Must find way that dispatcher can regain control!

• Answer: utilize external events
  – Interrupts: signals from hardware or software that stop the running code and jump to kernel
  – Timer: like an alarm clock that goes off every some milliseconds

• If we make sure that external events occur frequently enough, can ensure dispatcher runs
Recall: Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can’t be disabled
**Example: Network Interrupt**

- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately
Use of Timer Interrupt to Return Control

- Solution to our dispatcher problem
  - Use the timer interrupt to force scheduling decisions

- Timer Interrupt routine:

```
TimerInterrupt()
{
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
Administrivia

• Midterm Thursday 2/16
  – No class on day of midterm
  – 7-9PM
• Project 1 Design Document due next Friday 2/10
• 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
ThreadFork(): Create a New Thread

• ThreadFork() is a user-level procedure that creates a new thread and places it on ready queue

• Arguments to ThreadFork()
  – Pointer to application routine (fcnPtr)
  – Pointer to array of arguments (fcnArgPtr)
  – Size of stack to allocate

• Implementation
  – Sanity check arguments
  – Enter Kernel-mode and Sanity Check arguments again
  – Allocate new Stack and TCB
  – Initialize TCB and place on ready list (Runnable)
How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address ⇒ OS (asm) routine ThreadRoot()
  - Two arg registers (a0 and a1) initialized to fcnPtr and fcnArgPtr, respectively

- Initialize stack data?
  - Minimal initialization ⇒ setup return to go to beginning of ThreadRoot()
  » Important part of stack frame is in registers for RISC-V (ra)
  » X86: need to push a return address on stack
  - Think of stack frame as just before body of ThreadRoot() really gets started
How does Thread get started?

- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread
How does a thread get started?

- How do we make a new thread?
  - Setup TCB/kernel thread to point at new user stack and ThreadRoot code
  - Put pointers to start function and args in registers or top of stack
    » This depends heavily on the calling convention (i.e. RISC-V vs x86)
- Eventually, run_new_thread() will select this TCB and return into beginning of ThreadRoot()
  - This really starts the new thread

```c
SetupNewThread(tNew) {
    ...
    TCB[tNew].regs.sp = newStackPtr;
    TCB[tNew].regs.retpc = &ThreadRoot;
    TCB[tNew].regs.r0 = fcnPtr
    TCB[tNew].regs.r1 = fcnArgPtr
}
```
What does `ThreadRoot()` look like?

- `ThreadRoot()` is the root for the thread routine:

  ```c
  ThreadRoot(fcnPTR, fcnArgPtr) {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    Call fcnPtr(fcnArgPtr);
    ThreadFinish();
  }
  ```

- **Startup Housekeeping**
  - Includes things like recording start time of thread
  - Other statistics

- **Stack will grow and shrink with execution of thread**

- **Final return from thread returns into `ThreadRoot()` which calls `ThreadFinish()`**
  - `ThreadFinish()` wake up sleeping threads
Processes vs. Threads: One Core

• Switch overhead:
  – Same process: low
  – Different proc.: high

• Protection
  – Same proc: low
  – Different proc: high

• Sharing overhead
  – Same proc: low
  – Different proc: high

• Parallelism: no
Processes vs. Threads: MultiCore

- Switch overhead:
  - Same process: low
  - Different proc.: high

- Protection
  - Same proc: low
  - Different proc: high

- Sharing overhead
  - Same proc: low
  - Different proc, simultaneous core: medium
  - Different proc, offloaded core: high

- Parallelism: yes
Recall: Simultaneous MultiThreading/Hyperthreading

- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!

- Original technique called “Simultaneous Multithreading”
  - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5
Processes vs. Threads: Hyper-Threading

- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention for ALUs/FPUs may hurt performance
## Threads vs Address Spaces: Options

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)</td>
<td>Mach, OS/2, Linux Windows 10 Win NT to XP, Solaris, HP-UX, OS X</td>
</tr>
</tbody>
</table>

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space
Goals for Rest of Today

• Challenges and Pitfalls of Concurrency
• Synchronization Operations/Critical Sections
• How to build a lock?
• Atomic Instructions
Multiprocessing vs Multiprogramming

- Some Definitions:
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process

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

![Diagram showing differences between Multiprocessing and Multiprogramming]
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);
    }
}

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

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-proc, or overlap comp 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

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

– This technique is used for graphical programming

• Complication:
  – What if we missed a blocking I/O step?
  – What if we have to split code into hundreds of pieces which could be blocking?
Can Threads Make This Easier?

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

- Requests proceed to completion, blocking as required:

```c
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:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1, acct-&gt;balance</td>
<td>load r1, acct-&gt;balance</td>
</tr>
<tr>
<td>add r1, amount1</td>
<td>add r1, amount2</td>
</tr>
<tr>
<td>store r1, acct-&gt;balance</td>
<td>store r1, acct-&gt;balance</td>
</tr>
</tbody>
</table>

Recall: Possible Executions

Thread 1 □□□□□ Thread 1 □□□□□□□□
Thread 2 □□□□□ Thread 2 □□□□□□□□
Thread 3 □□□□□ Thread 3 □□□□□□□□

a) One execution          b) Another execution

Thread 1 □□□□□
Thread 2 □□□□□
Thread 3 □□□□□□□□

c) Another execution
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!
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
  – Consequently – weird example that produces “3” on previous slide can’t happen

• 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><code>i = 0;</code></td>
<td><code>i = 0;</code></td>
</tr>
<tr>
<td><code>while (i &lt; 10)</code></td>
<td><code>while (i &gt; -10)</code></td>
</tr>
<tr>
<td><code>i = i + 1;</code></td>
<td><code>i = i - 1;</code></td>
</tr>
<tr>
<td><code>printf(“A wins!”);</code></td>
<td><code>printf(“B wins!”);</code></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:

<table>
<thead>
<tr>
<th>Thread A</th>
<th></th>
<th>Thread B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r1=0</td>
<td>load r1, M[i]</td>
<td>r1=0</td>
<td>load r1, M[i]</td>
</tr>
<tr>
<td>r1=1</td>
<td>add r1, r1, 1</td>
<td>r1=-1</td>
<td>sub r1, r1, 1</td>
</tr>
<tr>
<td>M[i]=1</td>
<td>store r1, M[i]</td>
<td>M[i]=-1</td>
<td>store r1, M[i]</td>
</tr>
</tbody>
</table>

• 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? With Hyperthreads?
  – Yes! Unlikely, but if you are depending on it not happening, it will and your system will break…
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's hard to build anything useful with only reads and writes

• **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 that only one thread can execute at once. Only one thread at a time will get into this section of code
  – Critical section is the result of mutual exclusion
  – Critical section and mutual exclusion are two ways of describing the same thing
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

- Locks need to be allocated and initialized:
  - structure `Lock mylock` or `pthread_mutex_t mylock;`
  - `lock_init(&mylock)` or `mylock = PTHREAD_MUTEX_INITIALIZER;`

- Locks provide two **atomic** operations:
  - `acquire(&mylock)` – wait until lock is free; then mark it as busy
    - After this returns, we say the calling thread *holds* the lock
  - `release(&mylock)` – mark lock as free
    - Should only be called by a thread that currently holds the lock
    - After this returns, the calling thread no longer holds the lock
Fix banking problem with Locks!

- Identify critical sections (atomic instruction sequences) and add locking:

```c
Deposit(acctId, amount) {
    acquire(&mylock) // Wait if someone else in critical section!
    acct = GetAccount(actId);
    acct->balance += amount;
    StoreAccount(acct);
    release(&mylock) // Release someone into critical section
}
```

- Must use SAME lock (mylock) with all of the methods (Withdraw, etc…)
  – Shared with all threads!

Threads serialized by lock through critical section.
Only one thread at a time
Correctness Requirements

- Threaded programs must work for all interleavings of thread instruction sequences
  - Cooperating threads inherently non-deterministic and non-reproducible
  - Really hard to debug unless carefully designed!
- Example: Therac-25
  - Machine for radiation therapy
    - Software control of electron accelerator and electron beam/Xray production
    - Software control of dosage
  - Software errors caused the death of several patients
    - A series of race conditions on shared variables and poor software design
    - “They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred.”

Figure 1: Typical Therac-25 facility
Motivating Example: “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>
Solve with a lock?

- **Recall:** Lock prevents someone from doing something
  - Lock before entering critical section
  - Unlock when leaving
  - 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 OJ

- Of Course – We don’t know how to make a lock yet
  - Let’s see if we can answer this question!
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Impulse is to start coding first, then when it doesn’t work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first
- What are the correctness properties for the “Too much milk” problem???
  - Never more than one person buys
  - Someone buys if needed
- First attempt: Restrict ourselves to use only atomic load and store 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 (wait)

• Suppose a computer tries this (remember, only memory read/write are atomic):

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

• Suppose a computer tries this (remember, only memory read/write are atomic):

```
Thread A                                      Thread B
if (noMilk) {
    if (noMilk) {
        if (noNote) {
            leave Note;
            buy Milk;
            remove Note;
        }
    }
}
```

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 (wait)
• Suppose a computer tries this (remember, only memory read/write are atomic):

```java
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 buying milk!
• Solution makes problem worse since fails intermittently
  – Makes it really hard to debug…
  – Must work despite what the dispatcher does!
Too Much Milk: Solution #1½

• Clearly the Note is not quite blocking enough
  – Let’s try to fix this by placing note first
• Another try at previous solution:

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

• What happens here?
  – Well, with human, probably nothing bad
  – With computer: no one ever buys milk
Too Much Milk Solution #2

• How about labeled notes?
  – Now we can leave note before checking
• Algorithm looks like this:

```plaintext
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;
```

• 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 this would happen, but will at worse possible time
  – Probably something like this in UNIX
Too Much Milk Solution #2: problem!

- *I’m* not getting milk, *You’re* getting milk
- This kind of lockup is called “starvation!”
Too Much Milk Solution #3

• Here is a possible two-note solution:

  Thread A                        Thread B
  leave note A;                   leave note B;
  while (note B) {
    do nothing;
  }
  if (noMilk) {
    buy milk;
  } remove note A;

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

• 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)”
Case 1

• “leave note A” happens before “if (noNote A)”

```cpp
leave note A;
while (note B) {
    do nothing;
}
if (noMilk) {
    buy milk;
}
remove note A;
```

```cpp
leave note B;
if (noNote A) {
    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 A;
while (note B) {
    do nothing;
};
if (noMilk) {
    buy milk;
}
remove note A;

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

Wait for note B to be removed

"leave note A" happens before "if (noNote A)"
Case 2

• “if (noNote A)” happens before “leave note A”

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

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

```
if (noMilk) {
    buy milk;
}
remove note A;
```
Case 2

- “if (noNote A)” happens before “leave note A”

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

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

if (noMilk) {
    buy milk;
}
remove note A;
Case 2

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

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

if (noMilk) {
  buy milk;
}
remove note A;
```
This Generalizes to $n$ Threads...

• Leslie Lamport’s “Bakery Algorithm” (1974)
Solution #3 discussion

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

```
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 got to be 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?

- Recall our target lock interface:
  - `acquire(&milklock)` – wait until lock is free, then grab
  - `release(&milklock)` – 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:
  ```
  acquire(&milklock);
  if (nomilk)
      buy milk;
  release(&milklock);
  ```
Where are we going with synchronization?

- 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.

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

• Every thread has both a user and kernel stack
  – Showed more details about context-switching mechanisms

• 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 a simple construction for a lock that uses interrupt disable mechanism
  – 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