CS162
Operating Systems and
Systems Programming
Lecture 7

Synchronization 2: Concurrency (Con’t),
Lock Implementation, Atomic Instructions

September 15th, 2022
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu
Recall: Multithreaded Stack Example

• Consider the following code blocks:

```c
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)
Recall: 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();
}
```
Hardware context switch support in x86

- Syscall/Intr (U \rightarrow K)
  - PL 3 \rightarrow 0;
  - TSS \leftarrow EFLAGS, CS:EIP;
  - SS:ESP \leftarrow k-thread stack (TSS PL 0);
  - push (old) SS:ESP onto (new) k-stack
  - push (old) eflags, cs:eip, <err>
  - CS:EIP \leftarrow <k target handler>

- Then
  - Handler saves other regs, etc
  - Does all its works, possibly choosing other threads, changing PTBR (CR3)
  - kernel thread has set up user GPRs

- iret (K \rightarrow U)
  - PL 0 \rightarrow 3;
  - Eflags, CS:EIP \leftarrow popped off k-stack
  - SS:ESP \leftarrow popped off k-stack

Pintos: tss.c, intr-stubs.S

pg 2,942 of 4,922 of x86 reference manual
Pintos: Kernel Crossing on Syscall or Interrupt

user code

user stack

kernel code

kernel thread stack

syscall / interrupt

system stack

PTBR

TCB

saves

ready to resume

iret

Time
Pintos: Context Switch – Scheduling

User code

User stack

Kernel code

Kernel thread stack

syscall / interrupt

Switch kernel threads

PTBR

TCB

saves

Time

Schedule

iret

 TTC

Processing

ready to resume

C: eip

ss: esp

C: eip

ss: esp

C: eip

ss: esp

C: eip

ss: esp

C: eip

ss: esp

Pintos: switch.S
• Each user process/thread associated with a kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel thread.
In User thread, with Kernel thread waiting

- x86 CPU holds interrupt SP in register
- During user thread execution, associated kernel thread is “standing by”
In Kernel Thread: No User Component

- Kernel threads execute with small stack in thread structure
- Pure kernel threads have no corresponding user-mode thread
User → Kernel (exceptions, syscalls)

- Mechanism to resume k-thread goes through interrupt vector
Kernel $\rightarrow$ User

- Interrupt return (iret) restores user stack, IP, and PL
Pintos Interrupt Processing

intrNN_stub()

***

wrap 0x20 (int #)
jmp intr_entry

push 0x21 (int #)
jmp intr_entry

***

intr_entry:
save regs as frame
set up kernel env.
call intr_handler

intr_exit:
restore regs
iret

Wrapper for
generic handler

stubs.S

Hardware
interrupt
vector
User → Kernel via interrupt vector

- Interrupt transfers control through the Interrupt Vector (IDT in x86)
- iret restores user stack and priority level (PL)
Switch to Kernel Thread for Process

Kernel User

Kernel

User

Stack

Heap

Code

Data

Proc Regs

PL: 0

IP

SP

K SP

Magic #

List

Priority

Status

PID

User

Stack

Kernel
Pintos Interrupt Processing

intrNN_stub():

- push 0x20 (int #)
- jmp intr_entry

intr_entry:

- save regs as frame
- set up kernel env.
- call intr_handler

intr_exit:

- restore regs
- iret

Wrapper for generic handler

Intr_handler(*frame):
- classify
- dispatch
- ack IRQ
- maybe thread yield

Pintos intr_handlers

interrupt.c

timer_intr(*frame): tick++ thread_tick()

timer.c
Timer may trigger thread switch

- **thread_tick**
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- **thread_yield**
  - On path to rtn from interrupt
  - Sets current thread back to READY
  - Pushes it back on ready_list
  - Calls schedule to select next thread to run upon iret
- **Schedule**
  - Selects next thread to run
  - Calls `switch_threads` to change regs to point to stack for thread to resume
  - Sets its status to RUNNING
  - If user thread, activates the process
  - Returns back to intr_handler
Thread Switch (switch.S)

- switch_threads: save regs on current small stack, change SP, return from destination threads call to switch_threads
Pintos Return from Processing

intrNN_stub()

***
push 0x20 (int #)
jmp intr_entry
push 0x20 (int #)
jmp intr_entry

intr_entry:
  save regs as frame
  set up kernel env.
  call intr_handler

intr_exit:
  restore regs
  iret

Wrapper for
generic handler

interrupt.c

Intr_handler(*frame)
- classify
- dispatch
- ack IRQ
- maybe thread yield

stubs.S

Hardware
interrupt
vector

Resume Some Thread

Pintos
intr_handlers

interrupt.c

timer_intr(*frame)
tick++
thread_tick()

thread_yield()
- schedule

Pintos Return from Processing

Lec 7.18
9/15/22
Kubiatowicz CS162 © UCB Fall 2022
Kernel → Different User Thread

- iret restores user stack and priority level (PL)
Famous Quote WRT Scheduling: Dennis Richie

Dennis Richie,
Unix V6, slp.c:

```
2230 /*
2231 * If the new process paused because it was
2232 * swapped out, set the stack level to the last call
2233 * to savu(u_ssav). This means that the return
2234 * which is executed immediately after the call to aretu
2235 * actually returns from the last routine which did
2236 * the savu.
2237 */
2238 /* You are not expected to understand this. */
```

“If the new process paused because it was swapped out, set the stack level to the last call to savu(u_ssav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu.”

“You are not expected to understand this.”

Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society(tuhs.org); gif by Eddie Koehler.

Included by Ali R. Butt in CS3204 from Virginia Tech
Administrivia

• Project 1 in full swing! Released on Saturday!
  – We expect that your design document will give intuitions behind your designs, not just a dump of pseudo-code
  – Think of this like you are in a company and your TA is your manager

• Paradox: need code for design document?
  – Not full code, just enough to prove you have thought through complexities of design

• Should be attending your permanent discussion section!
  – Discussion section attendance is mandatory, but don’t come in if sick!!
    » Email your TA if you cannot come to your discussion for a valid reason

• Midterm I: September 27th, 7-9PM (Two weeks from today!)
  – Fill out conflict request by Friday!
Goals for Rest of Today

• Challenges and Pitfalls of Concurrency
• Synchronization Operations/Critical Sections
• How to build a lock?
• Atomic Instructions
Recall: 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.
Recall: 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-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:
  ```
  Deposit(acctId, amount) {
      acct = GetAccount(acctId); /* 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
  ```
Recall: Possible Executions

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

a) One execution  b) Another execution

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**
  
  ```
  x = 1;
  ```

  **Thread B**
  
  ```
  y = 2;
  ```

- However, what about (Initially, \( y = 12 \)):
  
  **Thread A**
  
  ```
  x = 1;
  x = y+1;
  ```

  **Thread B**
  
  ```
  y = 2;
  y = y*2;
  ```

  – What are the possible values of \( x \)?

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

  **Thread A**
  
  ```
  x = 1;
  ```

  **Thread B**
  
  ```
  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(&quot;A wins!&quot;);</code></td>
<td><code>printf(&quot;B wins!&quot;);</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>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1=0   load r1, M[i]</td>
<td>r1=0   load r1, M[i]</td>
</tr>
<tr>
<td>r1=1   add r1, r1, 1</td>
<td>r1=-1  sub r1, r1, 1</td>
</tr>
<tr>
<td>M[i]=1  store r1, M[i]</td>
<td>M[i]=-1 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:
  - `struct 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.”
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):
  ```
  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

```java
if (noMilk) {
    if (noMilk) {
        if (noNote) {
            leave Note;
            buy Milk;
            remove Note;
        }
    }
}
```

Thread B

```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):
  
  ```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:

```
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!”
• 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) {</td>
<td>if (noNote A) {</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>remove note B;</td>
</tr>
<tr>
<td>}</td>
<td>}</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
Case 1

• “leave note A” happens before “if (noNote A)”
Case 1

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

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

```java
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
} remove note B;
if (noMilk) {
    buy milk;
} remove note A;
```
• “leave note A” happens before “if (noNote A)”
Case 2

• “if (noNote A)” happens before “leave note A”
Case 2

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

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

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

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

- “if (noNote A)” happens before “leave note A”
This Generalizes to $n$ Threads...

- Leslie Lamport’s “Bakery Algorithm” (1974)

A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport
Massachusetts Computer Associates, Inc.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate.
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 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:

```c
acquire(&milklock);
if (nomilk)
  buy milk;
release(&milklock);
```
Where are we going with synchronization?

<table>
<thead>
<tr>
<th>Programs</th>
<th>Shared Programs</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, Compare&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
Back to: 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?
    » What is 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:
  ```
  LockAcquire { disable Ints; }
  LockRelease { enable Ints; }
  ```

• Problems with this approach:
  – Can’t let user do this! Consider following:
    ```
    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

    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;
}
```
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;
}
```

• Before Putting thread on the wait queue?
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;
}
```

Enable Position

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread
Interrupt Re-enable in Going to Sleep

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

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

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread
• After putting the thread on the wait queue
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;
}
```

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread

• After putting the thread on the wait queue
  – Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  – Misses wakeup and still holds lock (deadlock!)
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;
}
```

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread

• After putting the thread on the wait queue
  – Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
  – Misses wakeup and still holds lock (deadlock!)

• Want to put it after `sleep()`. But – how?
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

Thread A

•

Thread B

•
INIT

int value = 0;

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

Release() {
    disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    enable interrupts;
}
**In-Kernel Lock: Simulation**

**INIT**

```
int value = 0;
```

**Acquire()**

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

**Release()**

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

Thread A

Thread B

Value: 1

waiting

owner

READY

Running

Thread A

lock.Acquire();
...
critical section;
...
lock.Release();

lock.Acquire();
...
critical section;
...
lock.Release();

Thread B

Value: 1

waiters

owner

Ready
**In-Kernel Lock: Simulation**

```
int value = 0;

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

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

Thread A

Thread B
INIT

```
int value = 0;

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

lock.Acquire();
... critical section; ...
lock.Release();

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

Value: 1

waiters owner

READY

Thread A

Thread B

In-Kernel Lock: Simulation

Thread A

Thread B

In-Kernel Lock: Simulation
```c
INIT

int value = 0;

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

lock.Acquire();
...
critical section;
...
lock.Release();

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

lock.Acquire();
...
critical section;
...
lock.Release();
```
In-Kernel Lock: Simulation

Value: 1

waiters | owner

READY

Thread A

Thread B

INIT

int value = 0;

Acquire()

disable interrupts;

if (value == 1)

put thread on wait-queue;

goto sleep()

else

value = 1;

enable interrupts;

Release()

disable interrupts;

if anyone on wait queue

take thread off wait-queue

Place on ready queue;

else

value = 0;

enable interrupts;

lock.Acquire();

... critical section;

... lock.Release();

lock.Acquire();

... critical section;

... lock.Release();

lock.Acquire();

... critical section;

... lock.Release();
Conclusion

• 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

• Talked about hardware atomicity primitives:
  – Disabling of Interrupts, test&set, swap, compare&swap, load-locked & store-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