Parallelism, Multicore, and Synchronization

Anne Bracy

CS 3410

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer. Also some slides from Amir Roth & Milo Martin in here.

P & H Chapter 4.10, 1.7, 1.8, 5.10, 6
Performance Improvement 101

\[ \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}} \]

2 Classic Goals of Architects:

\( \downarrow \text{Clock period} \ (\uparrow \text{Clock frequency}) \)

\( \downarrow \text{Cycles per Instruction} \ (\uparrow \text{IPC}) \)
Clock frequencies have stalled

Darling of performance improvement for decades

Why is this no longer the strategy?

Hitting Limits:

• Pipeline depth
• Clock frequency
• Moore’s Law & Technology Scaling
• Power
Improving IPC via ILP

Exploiting Intra-instruction parallelism:
  Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):
  Multiple issue pipeline (2-wide, 4-wide, etc.)
  - Statically detected by compiler (VLIW)
  - Dynamically detected by HW

Dynamically Scheduled (OoO)
Static Multiple Issue

a.k.a. Very Long Instruction Word (VLIW)

Compiler groups instructions to be issued together
  • Packages them into “issue slots”

How does HW detect and resolve hazards?
  It doesn’t. 😊 Compiler must avoid hazards

Example: Static Dual-Issue 32-bit MIPS
  • Instructions come in pairs (64-bit aligned)
    – One ALU/branch instruction (or nop)
    – One load/store instruction (or nop)
MIPS with Static Dual Issue

Two-issue packets

- One ALU/branch instruction
- One load/store instruction
- 64-bit aligned
  - ALU/branch, then load/store
  - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>
## Scheduling Example

Schedule this for dual-issue MIPS

### Loop:
- **lw** $t0, 0($s1)  # $t0=array element
- **addu** $t0, $t0, $s2  # add scalar in $s2
- **sw** $t0, 0($s1)  # store result
- **addi** $s1, $s1, −4  # decrement pointer
- **bne** $s1, $zero, Loop  # branch $s1!=0

### Schedule

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loop:</strong></td>
<td><strong>lw</strong> $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1,−4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

### Clicker Question: What is the IPC of this machine?
- (A) 0.8  
- (B) 1.0  
- (C) 1.25  
- (D) 1.5  
- (E) 2.0
Techniques and Limits of Static Scheduling

Goal: larger instruction windows (to play with)
- Predication
- Loop unrolling
- Function in-lining
- Basic block modifications (superblocks, etc.)

Roadblocks
- Memory dependences (aliasing)
- Control dependences
Improving IPC via ILP

Exploiting Intra-instruction parallelism:
   Pipelining (decode A while fetching B)

Exploiting Instruction Level Parallelism (ILP):
   Multiple issue pipeline (2-wide, 4-wide, etc.)
   • Statically detected by compiler (VLIW)
   • Dynamically detected by HW

Dynamically Scheduled (OoO)
Dynamic Multiple Issue

aka SuperScalar Processor (c.f. Intel)

- CPU chooses multiple instructions to issue each cycle
- Compiler can help, by reordering instructions....
- ... but CPU resolves hazards

Even better: Speculation/Out-of-order Execution

- Execute instructions as early as possible
- Aggressive register renaming (indirection to the rescue!)
- Guess results of branches, loads, etc.
- Roll back if guesses were wrong
- Don’t commit results until all previous insns committed
Effectiveness of OoO Superscalar

It was awesome, but then it stopped improving

Limiting factors?

• Programs dependencies
• Memory dependence detection → be conservative
  – e.g. Pointer Aliasing: A[0] += 1; B[0] *= 2;
• Hard to expose parallelism
  – Still limited by the fetch stream of the static program
• Structural limits
  – Memory delays and limited bandwidth
• Hard to keep pipelines full, especially with branches
Improving IPC via ILP TLP

Exploiting Thread-Level parallelism

Hardware multithreading to **improve utilization**:

- Multiplexing multiple threads on single CPU
- Sacrifices latency for throughput
- Single thread cannot fully utilize CPU? *Try more!*
- Three types:
  - Course-grain (has preferred thread)
  - Fine-grain (round robin between threads)
  - Simultaneous (hyperthreading)
What is a thread?

Process: multiple threads, code, data and OS state

Threads: share code, data, files, **not** regs or stack

![Diagram showing single-threaded process vs multithreaded process]
Time evolution of issue slots

- Color = thread, white = no instruction

4-wide Superscalar

Switch to thread B on thread A L2 miss

CGMT

Switch threads every cycle

FGMT

SMT

Insns from multiple threads coexist
# Power Efficiency

<table>
<thead>
<tr>
<th>CPU</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc III</td>
<td>2003</td>
<td>1950MHz</td>
<td>14</td>
<td>4</td>
<td>No</td>
<td>1</td>
<td>90W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core i5 Nehal</td>
<td>2010</td>
<td>3300MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>1</td>
<td>87W</td>
</tr>
<tr>
<td>Core i5 Ivy Br</td>
<td>2012</td>
<td>3400MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>8</td>
<td>77W</td>
</tr>
<tr>
<td>UltraSparc T1</td>
<td>2005</td>
<td>1200MHz</td>
<td>6</td>
<td>1</td>
<td>No</td>
<td>8</td>
<td>70W</td>
</tr>
</tbody>
</table>

Those simpler cores did something very right.
Why Multicore?

Moore’s law

• A law about transistors
• Smaller means more transistors per die
• And smaller means faster too

But: Power consumption growing too...
Power Wall

Power = capacitance * voltage^2 * frequency

In practice: Power ~ voltage^3

Reducing voltage helps (a lot)
... so does reducing clock speed
Better cooling helps

The power wall

• We can’t reduce voltage further
• We can’t remove more heat
Why Multicore?

- **Single-Core Overclocked +20%**
  - Performance: 1.2x
  - Power: 1.7x

- **Single-Core**
  - Performance: 1.0x
  - Power: 1.0x

- **Single-Core Underclocked -20%**
  - Performance: 0.8x
  - Power: 0.51x

- **Dual-Core Underclocked -20%**
  - Performance: 1.6x
  - Power: 1.02x
Parallel Programming

Q: So let's just all use multicore from now on!
A: Software must be written as parallel program

Multicore difficulties

- Partitioning work
- Coordination & synchronization
- Communications overhead
- How do you write parallel programs?
  ... without knowing exact underlying architecture?
Work Partitioning

Partition *work* so all cores have something to do
Load Balancing

Need to partition so all cores are actually working
Amdahl’s Law

If tasks have a serial part and a parallel part...

Example:

step 1: divide input data into $n$ pieces
step 2: do work on each piece
step 3: combine all results

Recall: Amdahl’s Law

As number of cores increases ...

• time to execute parallel part? goes to zero
• time to execute serial part? Remains the same
• Serial part eventually dominates
Parallelism is a necessity

Necessity, not luxury
Power wall

Not easy to get performance out of

Many solutions
Pipelining
Multi-issue
Multithreading
Multicore
So let's just all use multicore from now on!

A: Software must be written as parallel program

**Multicore difficulties**

- Partitioning work
- **Coordination & synchronization**
- Communications overhead
- How do you write parallel programs?
  
  ... without knowing exact underlying architecture?
Parallelism & Synchronization

Cache Coherency

- Processors cache shared data → they see different (incoherent) values for the same memory location

Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes
Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- Hardware provides *single physical address* space for all processors
Thread A (on Core0)
for(int i = 0, i < 5; i++) {
    x = x + 1;
}

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
    x = x + 1;
}

What will the value of x be after both loops finish?
Thread A (on Core0)
for(int i = 0, i < 5; i++) {
x = x + 1;
}

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
x = x + 1;
}

What will the value of \( x \) be after both loops finish?

a) 6
b) 8
c) 10
d) Could be any of the above
e) Couldn’t be any of the above
Cache Coherency Problem, WB $\$

*Thread A (on Core0)*

```java
for(int i = 0, i < 5; i++) {
    $t0=0  LW $t0, addr(x)
    $t0=1  ADDIU $t0, $t0, 1
    x=1   SW $t0, addr(x)
}
```

*Thread B (on Core1)*

```java
for(int j = 0; j < 5; j++) {
    $t0=0  LW $t0, addr(x)
    $t0=1  ADDIU $t0, $t0, 1
    x=1   SW $t0, addr(x)
}
```

**Problem!**
## Not just a problem for Write-Back Caches

### Executing on a write-thru cache:

<table>
<thead>
<tr>
<th>Time step</th>
<th>Event</th>
<th>CPU A’s cache</th>
<th>CPU B’s cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

![Diagram of Core0, Core1, CoreN caches, Memory, and I/O with interconnect]
Two issues

Coherence

- What values can be returned by a read
- Need a globally uniform (consistent) view of a single memory location

**Solution:** Cache Coherence Protocols

Consistency

- When a written value will be returned by a read
- Need a globally uniform (consistent) view of *all* memory locations relative to each other

**Solution:** Memory Consistency Models
Coherence
- all copies have same data at all times

Coherence controller:
- Examines bus traffic (addresses and data)
- Executes coherence protocol
  - What to do with local copy when you see different things happening on bus

Three processor-initiated events
- **Ld**: load
- **St**: store
- **WB**: write-back

Two remote-initiated events
- **LdMiss**: read miss from *another* processor
- **StMiss**: write miss from *another* processor
VI Coherence Protocol

**VI (valid-invalid) protocol:**
- Two states (per block in cache)
  - V (valid): have block
  - I (invalid): don’t have block
+ Can implement with valid bit

**Protocol diagram (left)**
- If you load/store a block: transition to V
- If anyone else wants to read/write block:
  - Give it up: transition to I state
  - Write-back if your own copy is dirty

This is an **invalidate protocol**

**Update protocol:** copy data, don’t invalidate
- Sounds good, but wastes a lot of bandwidth
VI Protocol (Write-Back Cache)

Thread A
lw t0, 0(r3),
ADDIU $t0,$t0,1
sw t0,0(r3)

Thread B
lw t0, 0(r3)
ADDIU $t0,$t0,1
sw t0,0(r3)

lw by Thread B generates an “other load miss” event (LdMiss)

• Thread A responds by sending its dirty copy, transitioning to I
VI protocol is inefficient
- Only one cached copy allowed in entire system
- Multiple copies can’t exist even if read-only
  - Not a problem in example
  - Big problem in reality

**MSI (modified-shared-invalid)**
- Fixes problem: splits “V” state into two states
  - M (modified): local dirty copy
  - S (shared): local clean copy
- Allows **either**
  - Multiple read-only copies (S-state)  --OR--
  - Single read/write copy (M-state)
## MSI Protocol (Write-Back Cache)

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>S:0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>M:1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>S:1</td>
<td>S:1</td>
<td>1</td>
</tr>
<tr>
<td>I:</td>
<td>M:2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Thread A**
- `lw t0, 0(r3),`
- `ADDIU $t0,$t0,1`
- `sw t0,0(r3)`

**Thread B**
- `lw t0, 0(r3),`
- `ADDIU $t0,$t0,1`
- `sw t0,0(r3)`

### lw
- by Thread B generates a “other load miss” event (LdMiss)
  - Thread A responds by sending its dirty copy, transitioning to S

### sw
- by Thread B generates a “other store miss” event (StMiss)
  - Thread A responds by transitioning to I
Cache Coherence and Cache Misses

Coherence introduces two new kinds of cache misses

• **Upgrade miss**
  – On stores to read-only blocks
  – Delay to acquire write permission to read-only block

• **Coherence miss**
  – Miss to a block evicted by another processor’s requests

Making the cache larger...

• Doesn’t reduce these type of misses
• As cache grows large, these sorts of misses dominate

False sharing

• Two or more processors sharing parts of the same block
• But *not* the same bytes within that block (no actual sharing)
• Creates pathological “ping-pong” behavior
• Careful data placement may help, but is difficult
More Cache Coherence

In reality: many coherence protocols

- Snooping: VI, MSI, MESI, MOESI, ...
  - But Snooping doesn’t scale
- Directory-based protocols
  - Caches & memory record blocks’ sharing status in directory
  - Nothing is free → directory protocols are slower!

Cache Coherency:

- requires that reads return most recently written value
- Is a hard problem!
Are We Done Yet?

Thread A

lw t0, 0(r3)

ADDIU $t0,$t0,1

sw t0,0(x)

Thread B

lw t0, 0(r3)

ADDIU $t0,$t0,1

sw t0,0(x)

What just happened??
Is MSI Cache Coherency Protocol Broken??

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>S:0</td>
<td>S:0</td>
<td>0</td>
</tr>
<tr>
<td>l:</td>
<td>M:1</td>
<td>0</td>
</tr>
<tr>
<td>M:1</td>
<td>l:</td>
<td>1</td>
</tr>
</tbody>
</table>
Programming with threads

Within a thread: execution is sequential

Between threads?

- No ordering or timing guarantees
- Might even run on different cores at the same time

**Problem:** hard to program, hard to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Cache coherency is necessary but **not** sufficient...

Need explicit synchronization to make guarantees about concurrent threads!
Race conditions

Timing-dependent error involving access to shared state

Race conditions depend on how threads are scheduled
  • i.e. who wins “races” to update state

Challenges of Race Conditions
  • Races are intermittent, may occur rarely
  • Timing dependent = small changes can hide bug

Program is correct only if all possible schedules are safe
  • Number of possible schedules is huge
  • Imagine adversary who switches contexts at worst possible time
Hardware Support for Synchronization

Atomic **read & write** memory operation

- Between read & write: *no writes to that address*

Many atomic hardware primitives

- **test and set** (x86)
- **atomic increment** (x86)
- **bus lock prefix** (x86)
- **compare and exchange** (x86, ARM deprecated)
- **linked load / store conditional** (pair of insns) (MIPS, ARM, PowerPC, DEC Alpha, ...)


Synchronization in MIPS

Load linked: \texttt{LL rt, offset(rs)}

“I want the value at address X. Also, start monitoring any writes to this address.”

Store conditional: \texttt{SC rt, offset(rs)}

“If no one has changed the value at address X since the LL, perform this store and tell me it worked.”

- Data at location has not changed since the LL?
  - SUCCESS:
    - Performs the store
    - Returns 1 in rt
- Data at location has changed since the LL?
  - FAILURE:
    - Does not perform the store
    - Returns 0 in rt
Using LL/SC to create Atomic Increment

Load linked: \( \text{LL } rt, \text{ offset}(rs) \)
Store conditional: \( \text{SC } rt, \text{ offset}(rs) \)

\[ i++ \]
\[ \downarrow \]
\[ \text{LW } \text{t0}, 0(\text{s0}) \]
\[ \text{ADDIU } \text{t0}, \text{t0}, 1 \]
\[ \text{SW } \text{t0}, 0(\text{s0}) \]

atomic(\(i++\)) \[\downarrow\]
try: \( \text{LL } \text{t0}, 0(\text{s0}) \)
\[\rightarrow\]
\[ \text{ADDIU } \text{t0}, \text{t0}, 1 \]
\[ \text{SC } \text{t0}, 0(\text{s0}) \]
\[ \text{BEQZ } \text{t0}, \text{try} \]

Value in memory changed between LL and SC ?
\[ \rightarrow \text{SC returns 0 in } \text{t0} \rightarrow \text{retry} \]
## Atomic Increment in Action

**Load linked:** \( \text{LL} \ $t0, \ \text{offset}($s0) \)

**Store conditional:** \( \text{SC} \ $t0, \ \text{offset}($s0) \)

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread B $t0</th>
<th>Mem [$s0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>try: ( \text{LL} \ $t0, \ 0($s0) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>try: ( \text{LL} \ $t0, \ 0($s0) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( \text{ADDIU} \ $t0, \ $t0, \ 1 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>( \text{ADDIU} \ $t0, \ $t0, \ 1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>( \text{SC} \ $t0, \ 0($s0) )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>( \text{BEQZ} \ $t0, \ \text{try} )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>( \text{SC} \ $t0, \ 0 ($s0) )</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>( \text{BEQZ} \ $t0, \ \text{try} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

*Success!*  
*Failure!*
Critical Sections

Create atomic version of every instruction? NO

Does not scale or solve the problem

To eliminate races: identify *Critical Sections*

- only one thread can be in
- Contending threads must wait to enter

```
T1
CSEnter();
Critical section
CSExit();
```

```
T2
CSEnter();
# wait
# wait
Critical section
CSExit();
```
Mutual Exclusion Lock (Mutex)

Implementation of CSEnter and CSExit

• Only one thread can hold the lock at a time
  “I have the lock”
Mutex from LL and SC

m = 0;
mutex_lock(int *m) {
    test_and_set:  LI $t0, 1
    LL $t1, 0($a0)
    BNEZ $t1, test_and_set
    SC $t0, 0($a0)
    BEQZ $t0, test_and_set
}

mutex_unlock(int *m) {
    SW $zero, 0($a0)
}
# 2 threads attempt to grab the lock

`mutex_lock(int *m)`

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread A</th>
<th>Thread B</th>
<th>ThreadA</th>
<th>ThreadB</th>
<th>Mem M[a0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>SC $t0, 0 ($a0)</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>SC $t0, 0($a0)</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>BEQZ $t0, try</td>
<td>BEQZ $t0, try</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>try: LI $t0, 1</td>
<td>Critical section</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

**Failure!**  **Success!**
Goal: enforce data structure invariants

// invariant:
// data in A[h ... t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
    A[t] = c;
    t = (t+1)%n;
}

// consumer: take from head
char get() {
    while (t == h) { }
    char c = A[h];
    h = (h+1)%n;
    return c;
}
Goal: enforce data structure invariants

// invariant:
// data in A[h ... t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
    A[t] = c;
    t = (t+1)%n;
}

// consumer: take from head
char get() {
    while (t == h) { }
    char c = A[h];
    h = (h+1)%n;
    return c;
}

Clicker Q:
What’s wrong here?
a) Will lose update to t and/or h
b) Invariant is not upheld
c) Will produce if full
d) Will consume if empty
e) All of the above
Goal: enforce data structure invariants

// invariant:
// data in A[h ... t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
    A[t] = c;
    t = (t+1)%n;
}

// consumer: take from head
char get() {
    while (t == h) {
    }
    char c = A[h];
    h = (h+1)%n;
    return c;
}

What’s wrong here?
• Could miss an update to t or h
• Breaks invariants: only produce if not full, only consume if not empty
→ Need to synchronize access to shared data
Goal: enforce data structure invariants

// invariant:
// data in A[h ... t-1]
char A[100];
int h = 0, t = 0;

// producer: add to tail if room
void put(char c) {
    A[t] = c;
    t = (t+1)%n;
}

// consumer: take from head
char get() {
    while (t == h) {
    }
    char c = A[h];
    h = (h+1)%n;
    return c;
}
Language-level Synchronization

Lots of synchronization variations...

Reader/writer locks

- Any number of threads can hold a read lock
- Only one thread can hold the writer lock

Semaphores

- N threads can hold lock at the same time

Monitors

- Concurrency-safe data structure with 1 mutex
- All operations on monitor acquire/release mutex
- One thread in the monitor at a time