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.

P&H Chapter 2.11, 5.10, and 6.5
Topics: Goals for Today

Understand Cache Coherency

Synchronizing parallel programs
• Atomic Instructions
• HW support for synchronization

How to write parallel programs
• Threads and processes
• Critical sections, race conditions, and mutexes
Parallelism and Synchronization

Cache Coherency Problem: What happens when two or more processors cache shared data?

i.e. the view of memory held by two different processors is through their individual caches.

As a result, processors can see different (incoherent) values to the same memory location.
Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume physical addresses (ignore virtual memory)
- Assume uniform memory access (ignore NUMA)
Cache Coherency Problem

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) All of the above
e) None of the above
Cache Coherency Problem

Thread A (on Core0)
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)
}
x should be greater than 1 after both threads loop at least once!

Thread B (on Core1)
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)
}
**Cache Coherence Problem**
Suppose two CPU cores share a physical address space
- Write-through caches

<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</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

---

Core0

Cache

Core1

Cache

... ... ...

CoreN

Cache

---

Interconnect

---

Memory

I/O
Two issues

Coherence

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

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
Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence

• Migration of data to local caches
  – Reduces bandwidth for shared memory

• Replication of read-shared data
  – Reduces contention for access

Snooping protocols

• Each cache monitors bus reads/writes
Snooping
Snooping for Hardware Cache Coherence

- All caches monitor bus and all other caches
- Bus read: respond if you have dirty data
- Bus write: update/invalidiate your copy of data
Invalidating Snooping Protocols

Cache gets exclusive access to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read in another cache misses
  - Owning cache supplies updated value

<table>
<thead>
<tr>
<th>Time Step</th>
<th>CPU activity</th>
<th>Bus activity</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></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>Cache miss for 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>Invalidate for X</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>CPU B read X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Writing

Write-back policies for bandwidth

Write-invalidate coherence policy
  • First invalidate all other copies of data
  • Then write it in cache line
  • Anybody else can read it

Permits one writer, multiple readers

In reality: many coherence protocols
  • Snooping doesn’t scale
  • Directory-based protocols
    – Caches and memory record sharing status of blocks in a directory
Informally, Cache Coherency requires that reads return most recently written value

Cache coherence hard problem

Snooping protocols are one approach
Next Goal: Synchronization

Is cache coherency sufficient?

i.e. Is cache coherency (what values are read) sufficient to maintain consistency (when a written value will be returned to a read). Both coherency and consistency are required to maintain consistency in shared memory programs.
Synchronization

- Threads
- Critical sections, race conditions, and mutexes
- Atomic Instructions
  - HW support for synchronization
  - Using sync primitives to build concurrency-safe data structures
- Example: thread-safe data structures
- Language level synchronization
- Threads and processes
Programming with Threads

Need it to exploit multiple processing units
...to parallelize for multicore
...to write servers that handle many clients

Problem: hard even for experienced programmers
• Behavior can depend on subtle timing differences
• Bugs may be impossible to reproduce

Needed: synchronization of threads
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 not sufficient...

Need explicit synchronization to make sense of concurrency!
Programming with Threads

Concurrency poses challenges for:

Correctness
  • Threads accessing shared memory should not interfere with each other

Liveness
  • Threads should not get stuck, should make forward progress

Efficiency
  • Program should make good use of available computing resources (e.g., processors).

Fairness
  • Resources apportioned fairly between threads
Example: Multi-Threaded Program

Apache web server

```java
void main() {
    setup();
    while (c = accept_connection()) {
        req = read_request(c);
        hits[req]++;
        send_response(c, req);
    }
    cleanup();
}
```
Example: web server

Each client request handled by a separate thread (in parallel)

- Some shared state: hit counter, ...

```
Thread 52
read hits
addiu
write hits
```

(look familiar?)

```
Thread 205
read hits
addiu
write hits
```

Timing-dependent failure $\Rightarrow$ race condition

- hard to reproduce $\Rightarrow$ hard to debug
Two threads, one counter

Possible result: lost update!

hit\(s = 0\)

\(\text{time}\)

\(\text{ADDIU/SW: hit}\(s = 0 + 1\)\)

hit\(s = 1\)

\(\text{T1}\)

\(\text{LW (0)}\)

\(\text{T2}\)

\(\text{LW (0)}\)

\(\text{ADDIU/SW: hit}\(s = 0 + 1\)\)

Timing-dependent failure \(\Rightarrow\) race condition

- Very hard to reproduce \(\Rightarrow\) Difficult to debug
Race conditions

Def: timing-dependent error involving access to shared state

Whether a race condition happens depends on

- how threads scheduled
- i.e. who wins “races” to instruction that updates state vs. instruction that accesses state

Challenges about Race conditions

- Races are intermittent, may occur rarely
- Timing dependent = small changes can hide bug

A program is correct only if all possible schedules are safe

- Number of possible schedule permutations is huge
- Need to imagine an adversary who switches contexts at the worst possible time
Critical sections

What if we can designate parts of the execution as critical sections

- Rule: only one thread can be “inside” a critical section

<table>
<thead>
<tr>
<th>Thread 52</th>
<th>Thread 205</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSEnter()</td>
<td>CSEnter()</td>
</tr>
<tr>
<td>read hits</td>
<td>read hits</td>
</tr>
<tr>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td>write hits</td>
<td>write hits</td>
</tr>
<tr>
<td>CSExit()</td>
<td>CSExit()</td>
</tr>
</tbody>
</table>
Critical Sections

To eliminate races: use critical sections that only one thread can be in

- Contending threads must wait to enter

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

```
T2
CSEnter();
# wait
# wait
Critical section
CSExit();
```
Mutexes

Q: How to implement critical sections in code?
A: Lots of approaches....
Mutual Exclusion Lock (mutex)
lock(m): wait till it becomes free, then lock it
unlock(m): unlock it

```c
safe_increment() {
    pthread_mutex_lock(&m);
    hits = hits + 1;
    pthread_mutex_unlock(&m);
}
```
Mutexes

Only one thread can hold a given mutex at a time

Acquire (lock) mutex on entry to critical section
  • Or block if another thread already holds it

Release (unlock) mutex on exit
  • Allow one waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(&m);
pthread_mutex_lock(&m);
pthread_mutex_lock(&m);    // # wait
hits = hits+1;

pthread_mutex_unlock(&m);
```

**T1**

```
# wait
hits = hits+1;
```

**T2**

```
pthread_mutex_unlock(&m);
```
Next Goal

How to implement mutex locks?
What are the hardware primitives?

Then, use these mutex locks to implement critical sections, and use critical sections to write parallel safe programs.
Synchronization

Synchronization requires hardware support

- Atomic read/write memory operation
- No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory (e.g. ATS, BTS; x86)
- Or an atomic pair of instructions (e.g. LL and SC; MIPS)
Synchronization in MIPS

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

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

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in $t0, causing the code to try again.
  i.e. use this value 0 in $t0 to try again.
Synchronization in MIPS

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

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

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Example: atomic incrementor

```assembly
i++
\downarrow
\texttt{LW $t0, 0($s0)}
\texttt{ADDIU $t0, $t0, 1}
\texttt{SW $t0, 0($s0)}
```

```assembly
\texttt{atomic(i++)}
\downarrow
\texttt{try: LL $t0, 0($s0)}
\texttt{ADDIU $t0, $t0, 1}
\texttt{SC $t0, 0($s0)}
\texttt{BEQZ $t0, try}
```
# Synchronization in MIPS

Load linked: \[\text{LL } rt, \text{ offset(rs)}\]

Store conditional: \[\text{SC } rt, \text{ offset(rs)}\]

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Example: atomic incrementor

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread B $t0</th>
<th>Memory M[$s0]</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: LL $t0, 0($s0)</td>
<td>try: LL $t0, 0($s0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADDIU $t0, $t0, 1</td>
<td>ADDIU $t0, $t0, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SC $t0, 0($s0)</td>
<td>SC $t0, 0($s0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>BEQZ $t0, try</td>
<td>BEQZ $t0, try</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Synchronization in MIPS

Load linked: \( LL \ rt, \ offset(rs) \)

Store conditional: \( SC \ rt, \ offset(rs) \)

- Succeeds if location not changed since the LL
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Example: atomic incrementor

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread B $t0</th>
<th>Memory M[$s0]</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: ( LL \ $t0, 0($s0) )</td>
<td>try: ( LL \ $t0, 0($s0) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( ADDIU \ $t0, \ $t0, 1 )</td>
<td>( ADDIU \ $t0, \ $t0, 1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( SC \ $t0, 0($s0) )</td>
<td>( SC \ $t0, 0($s0) )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>( BEQZ \ $t0, try )</td>
<td>( BEQZ \ $t0, try )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Failure – try again

Success!
Mutex from LL and SC

Linked load / Store Conditional
m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked
mutex_lock(int *m) {
    while(test_and_set(m)){}
}

int test_and_set(int *m) {
    old = *m;  LL  Atomic
    *m = 1;   SC
    return old;
}
Mutex from LL and SC

Linked load / Store Conditional
m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked
mutex_lock(int *m) {
    while(test_and_set(m)){}
}

int test_and_set(int *m) {
    try: LI $t0, 1
    LL $t1, 0($a0)
    SC $t0, 0($a0)
    MOVE $v0, $t1
    BEQZ $t0, try
}
Mutex from LL and SC

Linked load / Store Conditional

\[ m = 0; \] // m=0 means lock is free; otherwise, if m=1, then lock locked

\[
\text{mutex_lock}(\text{int } *m) \{
    \text{while(\text{test\_and\_set}(m)){} }
\}
\]

\[
\text{int test\_and\_set(\text{int } *m) } \{ \\
    \text{try:} \\
    \quad \text{LI } \$t0, 1 \\
    \quad \text{LL } \$t1, 0(\$a0) \\
    \quad \text{SC } \$t0, 0(\$a0) \\
    \quad \text{BEQZ } \$t0, \text{try} \\
    \quad \text{BEQZ } \$t0, \text{try} \\
    \quad \text{MOVE } \$v0, \$t1 \\
\}\]

Mutex from LL and SC

Linked load / Store Conditional

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) {
    *m = 0;
}
Mutex from LL and SC

Linked load / Store Conditional

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

mutex_unlock(int *m) {
    SW $zero, 0($a0)
}
Mutex from LL and SC

Linked load / Store Conditional

\[ m = 0; \]

```
mutex_lock(int *m) {
```

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread A $t1</th>
<th>Thread B $t0</th>
<th>Thread B $t1</th>
<th>Mem M[$a0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SC $t0, 0($a0)</td>
<td>SC $t0, 0 ($a0)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>BEQZ $t0, try</td>
<td>BEQZ $t0, try</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Mutex from LL and SC

## Linked load / Store Conditional

\[ m = 0; \]

\[
\text{mutex\_lock}(\text{int} * m) \{ \\
\]

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0$</th>
<th>Thread A $t1$</th>
<th>Thread B $t0$</th>
<th>Thread B $t1$</th>
<th>Mem M[$a0$]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></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></td>
<td></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>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>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>SC $t0$, 0($a0$)</td>
<td>SC $t0$, 0($a0$)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>BEQZ $t0$, try</td>
<td>BEQZ $t0$, try</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mutex from LL and SC

Linked load / Store Conditional

\[ m = 0; \]

`mutex_lock(int *m) {`

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

- Failed to get mutex lock – try again
- Success grabbing mutex lock! Inside Critical section
Mutex from LL and SC

Linked load / Store Conditional

\[ m = 0; \]

\[
\text{mutex\_lock}(\text{int} \ *m) \ { \{ \\
\text{test\_and\_set:} \\
\quad \text{LI} \ \$t0, 1 \\
\quad \text{LL} \ \$t1, 0(\text{a0}) \\
\quad \text{BNEZ} \ \$t1, \text{test\_and\_set} \\
\quad \text{SC} \ \$t0, 0(\text{a0}) \\
\quad \text{BEQZ} \ \$t0, \text{test\_and\_set} \\
\}} \]

\[
\text{mutex\_unlock}(\text{int} \ *m) \ { \{ \\
\quad \text{SW} \ \$\text{zero}, 0(\text{a0}) \\
\} \]

This is called a Spin lock
Aka spin waiting
Mutex from LL and SC

Linked load / Store Conditional

```c
m = 0;
mutex_lock(int *m) {
```

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread A $t1</th>
<th>Thread B $t0</th>
<th>Thread B $t1</th>
<th>Mem M[$a0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Mutex from LL and SC**

Linked load / Store Conditional

\[ m = 0; \]

\[ \text{mutex_lock(int } *m) \} \]

<table>
<thead>
<tr>
<th>Time Step</th>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A $t0</th>
<th>Thread A $t1</th>
<th>Thread B $t0</th>
<th>Thread B $t1</th>
<th>Mem M[$a0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>try: LI $t0, 1</td>
<td>try: LI $t0, 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>LL $t1, 0($a0)</td>
<td>LL $t1, 0($a0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>BNEZ $t1, try</td>
<td>BNEZ $t1, try</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Now we can write parallel and correct programs

Thread A
for(int i = 0, i < 5; i++) {
    for(int j = 0; j < 5; j++) {
        x = x + 1;
    }
    mutex_lock(m);
    x = x + 1;
    mutex_unlock(m);
}

Thread B
for(int j = 0; j < 5; j++) {
    mutex_lock(m);
    x = x + 1;
    mutex_unlock(m);
}
Alternative Atomic Instructions

Other atomic hardware primitives
- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional (MIPS, ARM, PowerPC, DEC Alpha, ...)

Synchronization

Synchronization techniques

clever code

• must work despite adversarial scheduler/interrupts
• used by: hackers
• also: noobs

disable interrupts

• used by: exception handler, scheduler, device drivers, ...

disable preemption

• dangerous for user code, but okay for some kernel code

mutual exclusion locks (mutex)

• general purpose, except for some interrupt-related cases
Summary

Need parallel abstractions, especially for multicore

Writing correct programs is hard
  Need to prevent data races

Need critical sections to prevent data races
  Mutex, mutual exclusion, implements critical section
  Mutex often implemented using a lock abstraction

Hardware provides synchronization primitives such as LL and SC (load linked and store conditional) instructions to efficiently implement locks
Next Goal

How do we use synchronization primitives to build concurrency-safe data structure?
Attempt #1: Producer/Consumer

Access to shared data must be synchronized

- goal: enforce data structure invariants

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

// producer: add to list tail
void put(char c) {
    A[t] = c;
    t = (t+1)%n;
}
```
Access to shared data must be synchronized

- goal: enforce datastructure invariants

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

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

// consumer: take from list head
char get() {
    while (h == t) { };
    char c = A[h];
    h = (h+1)%n;
    return c;
}
```
Attempt #1: Producer/Consumer
Access to shared data must be synchronized

- goal: enforce datastructure invariants

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

// producer: add to list tail // consumer: take from list head
void put(char c) {
    A[t] = c;
    t = (t+1)%n;
}
char get() {
    while (h == t) { };
    char c = A[h];
    h = (h+1)%n;
    return c;
}
```

What is wrong with code?

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
Attempt #1: Producer/Consumer

Access to shared data must be synchronized

• goal: enforce datastructure invariants

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

// producer: add to list tail // consumer: take from list head
void put(char c) {
    A[t] = c;
    t = (t+1)%n;
}
char get() {
    while (h == t) { };
    char c = A[h];
    h = (h+1)%n;
    return c;
}

Error: could miss an update to t or h due to lack of synchronization
Current implementation will break invariant:
    only produce if not full and only consume if not empty

Need to synchronize access to shared data
Attempt #2: Protecting an invariant

// invariant: (protected by mutex m)
// data is in A[h ... t-1]

pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;

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

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

Rule of thumb: all access and updates that can affect invariant become critical sections
Attempt #2: Protecting an invariant

// invariant: (protected by mutex m)
// data is in A[h ... t-1]

pthread_mutex_t *m = pthread_mutex_create();
char A[100];
int h = 0, t = 0;

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

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

Rule of thumb: all access and updates that can affect invariant become critical sections

BUG: Can’t wait while holding lock
Guidelines for successful mutexing

Insufficient locking can cause races

• Skimping on mutexes? Just say no!

Poorly designed locking can cause deadlock

P1: lock(m1);  P2: lock(m2);
lock(m2);  lock(m1);

Circular Wait

• know why you are using mutexes!
• acquire locks in a consistent order to avoid cycles
• use lock/unlock like braces (match them lexically)
  – lock(&m); ...; unlock(&m)
  – watch out for return, goto, and function calls!
  – watch out for exception/error conditions!
Attempt #3: Beyond mutexes

Writers must check for full buffer & Readers must check if for empty buffer

```java
char get() {
    do {
        acquire(L);
        empty = (h == t);
        if (!empty) {
            c = A[h];
            h = (h+1)%n;
        }
        release(L);
    } while (empty);
    return c;
}
```

Does this work?
Yes, but, it is wasteful
Due to the spinning
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 datastructure with 1 mutex
- All operations on monitor acquire/release mutex
- One thread in the monitor at a time

Message-passing, sockets, queues, ring buffers, ...
- transfer data and synchronize
Summary

Hardware Primitives: test-and-set, LL/SC, barrier, ...
... used to build ...

Synchronization primitives: mutex, semaphore, ...
... used to build ...

Language Constructs: monitors, signals, ...