Synchronization 2

CS 3410, Spring 2014
Computer Science
Cornell University

See P&H Chapter: 2.11, 6.5
Next 3 weeks

- Week 12 (this week): Proj3 due Fri-Sun
  - Note Lab 4 is now IN CLASS
  - Prelim 2 review Sunday and Monday
- Week 13 (Apr 29): Proj4 release, Lab4 due Tue, Prelim2
- Week 14 (May 6): Proj3 tournament Mon, Proj4 design doc due

Final Project for class

- Week 15 (May 13): Proj4 due Wed
- Remember: No slip days for PA4
Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

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

Thread A (on Core0)
for(int i = 0, i < 5; i++) {
    A1) LW $t0, addr(x)
    A2) ADDIU $t0, $t0, 1
    A3) SW $t0, addr(x)
}

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
    B1) LW $t0, addr(x)
    B2) ADDIU $t0, $t1, 1
    B3) 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 to X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Diagram:**
- Core0 Cache
- Core1 Cache
- CoreN Cache
- ... ...
- Interconnect
- Memory
- I/O

**Legend:**
- Vertical arrows indicate data transfers between cache and memory.
- Horizontal arrows indicate data transfers between cores via interconnect.
Informal: **Reads** return most recently **written** value

Formal: For concurrent processes $P_1$ and $P_2$

- $P$ writes $X$ before $P$ reads $X$ (with no intervening writes)
  $\Rightarrow$ read returns written value
- $P_1$ writes $X$ before $P_2$ reads $X$
  $\Rightarrow$ read returns written value
- $P_1$ writes $X$ and $P_2$ writes $X$
  $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for $X$
  - Aka write serialization
Coherence Defined

Formal: For concurrent processes $P_1$ and $P_2$

- $P$ writes $X$ before $P$ reads $X$ (with no intervening writes)
  $\Rightarrow$ read returns written value
  - (preserve program order)

- $P_1$ writes $X$ before $P_2$ reads $X$
  $\Rightarrow$ read returns written value
  - (coherent memory view, can’t read old value forever)

- $P_1$ writes $X$ and $P_2$ writes $X$
  $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for $X$
  - Aka write serialization
  - (else $X$ can see $P_2$’s write before $P_1$ and $Y$ can see the opposite; their final understanding of state is wrong)
Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence and support shared memory

• Migration of data to local caches
  – Reduces bandwidth for shared memory (performance)
• Replication of read-shared data
  – Reduces contention for access (performance)

Snooping protocols

• Each cache monitors bus reads/writes (correctness)
Snooping

Snooping for Hardware Cache Coherence

• All caches monitor bus and all other caches

Write invalidate protocol

• Bus read: respond if you have dirty data
• Bus write: update/invalidate 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 is another cache miss
  - 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>0</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></td>
<td>1</td>
</tr>
</tbody>
</table>
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 is another cache miss
  - 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>0</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>1</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>
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

Works with one writer, multiple readers

In reality: many coherence protocols
- Snooping doesn’t scale
- MOESI, MOSI, ... (mod, own, exclusive, share, inv)
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory
Summary of cache coherence

Cache coherence requires that reads return most recently written value

Cache coherence is hard
   Snooping protocols are one approach

Cache coherence protocols alone are not enough
   Need more for consistency
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 isn’t sufficient...
Need explicit synchronization to make sense of concurrency!
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

```c
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
addi
write hits
(look familiar?)

Thread 205
read hits
addi
write hits

Timing-dependent failure $\Rightarrow$ race condition

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

Possible result: lost update!

hits = 0

\[ \text{time} \downarrow \]

T1

LW (0)

ADDIU/SW: hits = 0 + 1

T2

LW (0)

ADDIU/SW: hits = 0 + 1

hits = 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</th>
<th>52</th>
<th>Thread</th>
<th>205</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>hits</td>
<td>read</td>
<td>hits</td>
</tr>
<tr>
<td>addi</td>
<td></td>
<td>addi</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>hits</td>
<td>write</td>
<td>hits</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();
```
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

```c
pthread_mutex_init(&m);
pthread_mutex_lock(&m);
# wait
hit = hits+1;
# wait
pthread_mutex_unlock(&m);

T1

T2
```

```c
pthread_mutex_lock(&m);
# wait
hit = hits+1;
```

```c
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: \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

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in $t0

Use this value 0 to try again
Mutex from LL and SC

Linked load / Store Conditional

m = 0; // 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;
    *m = 1;
    return old;
}
Mutex from LL and SC

Linked load / Store Conditional

m = 0;
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)
        BEQZ $t0, try
        MOVE $v0, $t1
    }

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

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

- Succeeds if location not changed since the \( \text{LL} \): 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: ( \text{LL } t0, 0($s0) )</td>
<td>try: ( \text{LL } t0, 0($s0) )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( \text{ADDIU } t0, t0, 1 )</td>
<td>( \text{ADDIU } t0, t0, 1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( \text{SC } t0, 0($s0) )</td>
<td>( \text{SC } t0, 0 ($s0) )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>( \text{BEQZ } t0, \text{ try} )</td>
<td>( \text{BEQZ } t0, \text{ try} )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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

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)
}
Mutex from LL and SC

\[ m = 0; \]
\[
\text{mutex\_lock}(\text{int } *m) \{ \\
\]

<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></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

```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>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></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

```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>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>
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, ...)

Summary

Need parallel abstraction like 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
Topics

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
Next Goal

How do we use synchronization primitives to build concurrency-safe data structure?
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

```
// 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) {
    // Need: check if list full
    A[t] = c;
    t = (t+1)%n;
}
```
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
void put(char c) {
    // Need: check if list full
    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

// 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;

// 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
Rule of thumb: all access and updates that can affect invariant become critical sections
Guidelines for successful mutexing

Insufficient locking can cause races
  • Skimping on mutexes? Just say no!

But poorly designed locking can cause deadlock

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

- ideal: don’t busy wait… go to sleep instead

```cpp
char get() {
    while (h == t) {
    }
    lock (L);
    char c = A[h];
    h = (h+1)%n;
    unlock (L);
    return c;
}
```

Cannot check condition while Holding the lock, BUT, empty condition may no longer hold in critical section

Dilemma: Have to check while holding lock
Attempt#3: Beyond mutexes

Writers must check for full buffer & Readers must check if for empty buffer
  • ideal: don’t busy wait... go to sleep instead

```c
char get() {
    lock (L);
    while (h == t) {};
    char c = A[h];
    h = (h+1)%n;
    unlock (L);
    return c;
}
```

Dilemma: Have to check while holding lock, but cannot wait while holding lock
Writers must check for full buffer
& Readers must check if for empty buffer

- ideal: don’t busy wait... go to sleep instead

```c
char get() {
  do {
    lock (L);
    empty = (h == t);
    if (!empty) {
      c = A[h];
      h = (h+1)%n;
    }
  }
  while (empty);
  unlock (L);
  return c;
}
```
Language-Level Synchronization

Condition variables

Wait for condition to be true
Thread sleeps while waiting
Can wake up one thread or all threads

Monitors

...
Summary

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

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

Language Constructs: monitors, signals, ...