Synchronization

Hakim Weatherspoon
CS 3410, Spring 2013
Computer Science
Cornell University

P&H Chapter 2.11 and 5.8
Big Picture: Parallelism and Synchronization

How do I take advantage of multiple processors; parallelism?

How do I write (correct) parallel programs, cache coherency and synchronization?

What primitives do I need to implement correct parallel programs?
Goals for Today

Understand Cache Coherency Problem
  • Define Cache coherency problem

Synchronizing parallel programs
  • Atomic Instructions
  • HW support for synchronization

How to write parallel programs
  • Threads and processes
  • Critical sections, race conditions, and mutexes
Big Picture: Parallelism and Synchronization
Next Goal: Parallelism and Synchronization

Cache Coherency Problem: What happens when two or more processors cache *shared* data?
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.
Each processor core has its own L1 cache
Big Picture: Parallelism and Synchronization

Each processor core has its own L1 cache
Each processor core has its own L1 cache
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)
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)
What will the value of $x$ be after both loops finish?
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?

a) 6  
b) 8  
c) 10  
d) All of the above  
e) None of the above
Thread A (on Core0)
for(int i = 0, i < 5; i++) {
    $t0=0  \text{ LW } $t0, addr(x)
    $t0=1  \text{ ADDIU } $t0, $t0, 1
    x=1  \text{ 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  \text{ LW } $t0, addr(x)
    $t0=1  \text{ ADDIU } $t0, $t0, 1
    x=1  \text{ 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 showing cache coherence problem with interconnect, memory, and I/O](image-url)
Coherence Defined

Cache coherence defined...

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) \implies \text{read returns written value}
  - $P_1$ writes $X$ before $P_2$ reads $X$
  \implies \text{read returns written value}

- $P_1$ writes $X$ and $P_2$ writes $X$
  \implies \text{all processors see writes in the same order}
  - all see the same final value for $X$
  - Aka write serialization
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 for Hardware Cache Coherence

• All caches monitor bus and all other caches
• **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 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. With multiprocessors maintaining cache coherency can be difficult and requires cache coherency protocols like Snooping cache coherency protocols.
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.
Is Cache Coherency Sufficient?

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

Very expensive and difficult to maintain consistency

Thread B (on Core1)
for(int j = 0; j < 5; j++) {
    LW $t0, addr(x)
    ADDIU $t0, $t0, 1
    SW $t0, addr(x)
}
Synchronization

Two processors sharing an area of memory

• P1 writes, then P2 reads
• **Data race** if P1 and P2 don’t *synchronize*
  – Result depends of order of accesses

Hardware support required

• 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

Example: atomic swap (to test/set lock variable)

\begin{verbatim}
try: MOVE $t0,$s4 ;copy exchange value
  LL $t1,0($s1); load linked
  SC $t0,0($s1); store conditional
  BEQZ $t0,try ; branch store fails
  MOVE $s4,$t1 ; put load value in $s4
\end{verbatim}

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.
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_test(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_test(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

mutex_lock(int *m) {
    while(test_and_test(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
}
Mutex from LL and SC

Linked load / Store Conditional

```c
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

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)
}
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, ...)

\[\textit{very expensive}\]
Now we can write parallel and correct programs

Thread A
for(int i = 0, i < 5; i++) {
    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);
}
Informally, Cache Coherency requires that reads return most recently written value.

With multiprocessors maintaining cache coherency can be difficult and requires cache coherency protocols like Snooping cache coherency protocols.

Cache coherency controls what values are read, but may be insufficient or very expensive to maintain consistency (when a written value will be returned to a read). We need synchronization primitives to more efficiently implement parallel and correct programs.
Next Goal

How do we write parallel programs?
Processes

How do we cope with lots of activity?

Simplicity? Separation into processes

Reliability? Isolation

Speed? Program-level parallelism
### Process and Program

#### Process
OS abstraction of a running computation
- The unit of execution
- The unit of scheduling
- Execution state + address space

From process perspective
- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, ...

<table>
<thead>
<tr>
<th>Process</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OS abstraction of a running computation</strong></td>
<td><strong>“Blueprint” for a process</strong></td>
</tr>
<tr>
<td>• The unit of execution</td>
<td>• Passive entity (bits on disk)</td>
</tr>
<tr>
<td>• The unit of scheduling</td>
<td>• Code + static data</td>
</tr>
<tr>
<td>• Execution state + address space</td>
<td></td>
</tr>
</tbody>
</table>

#### Program

<table>
<thead>
<tr>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>Initialized data</td>
</tr>
<tr>
<td>BSS</td>
</tr>
<tr>
<td>Symbol table</td>
</tr>
<tr>
<td>Line numbers</td>
</tr>
<tr>
<td>Ext. refs</td>
</tr>
</tbody>
</table>
Role of the OS

Context Switching
• Provides illusion that every process owns a CPU

Virtual Memory
• Provides illusion that process owns some memory

Device drivers & system calls
• Provides illusion that process owns a keyboard, ...

To do:
How to start a process?
How do processes communicate / coordinate?
How to create a process?

Q: How to create a process?
A: Double click

After boot, OS starts the first process

...which in turn creates other processes

• parent / child → the process tree
$ pstree | view -
init-+--NetworkManager-+-dhclient
 | -apache2
 | -chrome-+-chrome
 |   `-chrome
 | -chrome-+--chrome
 | -clementine
 | -clock-applet
 | -cron
 | -cupsd
 | -firefox-+-run-mozilla.sh-+-firefox-bin-+-plugin-cont
 | -gnome-screensaver
 | -grep
 | -in.tftpd
 | -ntpd
 `--sshd--sshd--sshd--bash-+-gcc-+-gcc-+-cc1
    `-pstree
    `-vim
   `--view
Processes Under UNIX

Init is a special case. For others...

Q: How does parent process create child process?
A: `fork()` system call

Wait. what? int `fork()` returns TWICE!
Example

main(int ac, char **av) {
    int x = getpid(); // get current process ID from OS
    char *hi = av[1]; // get greeting from command line
    printf("I’m process %d\n", x);
    int id = fork();
    if (id == 0)
        printf("%s from %d\n", hi, getpid());
    else
        printf("%s from %d, child is %d\n", hi, getpid(), id);
}

$ gcc -o strange strange.c
$ ./strange “Hey”
I’m process 23511
Hey from 23512
Hey from 23511, child is 23512
Parent can pass information to child

- In fact, *all parent data* is passed to child
- But isolated after (C-O-W ensures changes are invisible)

Q: How to continue communicating?
A: Invent OS “IPC channels” : send(msg), recv(), ...
Inter-process Communication

Parent can pass information to child

- In fact, *all parent data* is passed to child
- But isolated after (C-O-W ensures changes are invisible)

Q: How to continue communicating?

A: Shared (Virtual) Memory!
Processes and Threads
Processes are heavyweight

Parallel programming with processes:

• They share almost everything
code, shared mem, open files, filesystem privileges, ...

• Pagetables will be *almost* identical

• Differences: PC, registers, stack

Recall: process = *execution context + address space*
## Processes and Threads

<table>
<thead>
<tr>
<th>Process</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS abstraction of a running computation</td>
<td>OS abstraction of a single thread of control</td>
</tr>
<tr>
<td>- The unit of execution</td>
<td>- The unit of scheduling</td>
</tr>
<tr>
<td>- The unit of scheduling</td>
<td>- Lives in one single process</td>
</tr>
<tr>
<td>- Execution state + address space</td>
<td></td>
</tr>
<tr>
<td><strong>From process perspective</strong></td>
<td><strong>From thread perspective</strong></td>
</tr>
<tr>
<td>- a virtual CPU</td>
<td>- one virtual CPU core on a virtual multi-core machine</td>
</tr>
<tr>
<td>- some virtual memory</td>
<td></td>
</tr>
<tr>
<td>- a virtual keyboard, screen, ...</td>
<td></td>
</tr>
</tbody>
</table>
#include <pthread.h>

int counter = 0;

void PrintHello(int arg) {
    printf("I'm thread %d, counter is %d\n", arg, counter++);
    ... do some work ...
    pthread_exit(NULL);
}

int main () {
    for (t = 0; t < 4; t++) {
        printf("in main: creating thread %d\n", t);
        pthread_create(NULL, NULL, PrintHello, t);
    }
    pthread_exit(NULL);
}
Threads versus Fork

in main: creating thread 0
I’m thread 0, counter is 0

in main: creating thread 1
I’m thread 1, counter is 1

in main: creating thread 2

in main: creating thread 3
I’m thread 3, counter is 2

I’m thread 2, counter is 3

If processes?
Example Multi-Threaded Program

Example: Apache web server

```c
void main() {
    setup();
    while (c = accept_connection()) {
        req = read_request(c);
        hits[req]++;
        send_response(c, req);
    }
    cleanup();
}
```
### Race Conditions

**Example:** Apache web server

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

- Some shared state: hit counter. ...

<table>
<thead>
<tr>
<th>Thread 52</th>
<th>Thread 205</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

(look familiar?)

Timing-dependent failure $\Rightarrow$ **race condition**

- hard to reproduce $\Rightarrow$ hard to debug
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!
Race conditions

Race Condition
Timing-dependent error when accessing shared state
  • Depends on scheduling happenstance
    ... e.g. who wins “race” to the store instruction?

Concurrent Program Correctness = all possible schedules are safe
  • Must consider *every possible* permutation
  • In other words...
    ... the scheduler is your adversary
Critical sections

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

- Rule: only one thread can be “inside”

<table>
<thead>
<tr>
<th>Thread 52</th>
<th>Thread 205</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
Mutexes

Q: How to implement critical section in code?
A: Lots of approaches....

Mutual Exclusion Lock (mutex)

acquire(m): wait till it becomes free, then lock it
release(m): unlock it

```c
apache_got_hit() {
    pthread_mutex_lock(m);
    hits = hits + 1;
    pthread_mutex_unlock(m)
}
```
Informally, Cache Coherency requires that reads return most recently written value.

With multiprocessors maintaining cache coherency can be difficult and requires cache coherency protocols like Snooping cache coherency protocols.

Cache coherency controls *what* values are read, but may be insufficient or very expensive to maintain consistency (*when* a written value will be returned to a read). We need synchronization primitives to more efficiently implement parallel and correct programs.

Processes and Threads are the abstraction that we use to write parallel programs? Fork and Joint and Interprocesses communication (IPC) can be used to coordinate processes. Threads are used to coordinate use of shared memory within a process.
Next time.

Higher level synchronization primitives (other abstractions to implement a critical section beyond mutexes)?
Administrivia

Project 3 due next week, Monday, April 22\textsuperscript{nd}
- Design Doc due \textit{yesterday}, Monday, April 15\textsuperscript{th}
- Games night Friday, April 26\textsuperscript{th}, 5-7pm.
- Location: B17 Upson

Homework 4 is due tomorrow, Wednesday, April 17\textsuperscript{th}
- Work alone
- Question 1 on Virtual Memory is pre-lab question for Lab 4

Prelim 3 is next week, Thursday, April 25\textsuperscript{th}
- Time and Location: 7:30pm in Phillips 101 and Upson B17
- Old prelims are online in CMS
Next three weeks

- Week 12 (Apr 15): Project3 design doc due and HW4 due
- Week 13 (Apr 22): Project3 due and Prelim3
- Week 14 (Apr 29): Project4 handout

Final Project for class

- Week 15 (May 6): Project4 design doc due
- Week 16 (May 13): Project4 due