Parallelism, Multicore, and Synchronization

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.
Announcements

• C practice assignment
  • Due Monday, April 23rd

• P4-Buffer Overflow is due tomorrow
  • Due Wednesday, April 18th

• P5-Cache Collusion!
  • Due Friday, April 27th
  • *Pizza party & Tournament, Monday, May 7th*

• Prelim2
  • Thursday, May 3rd, 7pm 185 Statler Hall
It took a lot of work, but this latest Linux patch enables support for machines with 4,096 CPUs, up from the old limit of 1,024.

Do you have support for smooth full-screen Flash video yet?

No, but who uses that?
Pitfall: Amdahl’s Law

Execution time after improvement =
affected execution time

\[ T_{\text{improved}} = \frac{T_{\text{affected}}}{ \text{improvement factor}} + T_{\text{unaffected}} \]
Pitfall: Amdahl’s Law
Improving an aspect of a computer and expecting a proportional improvement in overall performance

\[ T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}} \]

Example: multiply accounts for 80s out of 100s
• Multiply can be parallelized
Scaling Example

Workload: sum of 10 scalars, and $10 \times 10$ matrix sum

• Speed up from 10 to 100 processors?

Single processor: Time = $(10 + 100) \times t_{\text{add}}$

10 processors

100 processors
Scaling Example

What if matrix size is $100 \times 100$?

Single processor: Time = $(10 + 10000) \times t_{add}$

10 processors

100 processors
  • Time = 10

Assuming load balanced
Unfortunately, we cannot obtain unlimited scaling (speedup) by adding unlimited parallel resources, eventual performance is dominated by a component needing to be executed sequentially. Amdahl's Law is a caution about this diminishing return
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:

\(\downarrow\) Clock period (\(\uparrow\) Clock frequency)

\(\downarrow\) Cycles per Instruction (\(\uparrow\) 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)
Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?

A: Deeper pipeline
  – E.g. 250MHz 1-stage; 500Mhz 2-stage; 1GHz 4-stage; 4GHz 16-stage

Pipeline depth limited by...
  – max clock speed (less work per stage ⇒ shorter clock cycle)
  – min unit of work
  – dependencies, hazards / forwarding logic
Instruction-Level Parallelism (ILP)

Pipelining: execute multiple instructions in parallel

Q: How to get more instruction level parallelism?
A: Multiple issue pipeline

  – Start multiple instructions per clock cycle in duplicate stages
Multiple issue pipeline

Static multiple issue
aka Very Long Instruction Word
Decisions made by compiler

Dynamic multiple issue
Decisions made on the fly

Cost: More execute hardware
Reading/writing register files: more ports
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>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scheduling Example

Schedule this for dual-issue MIPS

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Speculation

Reorder instructions

To fill the issue slot with useful work

Complicated: exceptions may occur
Optimizations to make it work

Move instructions to fill in nops
Need to track hazards and dependencies

Loop unrolling
### Scheduling Example

Compiler scheduling for dual-issue MIPS...

**Loop:**

- `lw   $t0, 0($s1)     # $t0 = A[i]`
- `lw   $t1, 4($s1)    # $t1 = A[i+1]`
- `addu $t0, $t0, $s2    # add $s2`
- `addu $t1, $t1, $s2    # add $s2`
- `sw   $t0, 0($s1)     # store A[i]`
- `sw   $t1, 4($s1)     # store A[i+1]`
- `addi $s1, $s1, +8    # increment pointer`
- `bne $s1, $s3, Loop  # continue if $s1!=end`

<table>
<thead>
<tr>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop: nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 4($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>sw $t0, 0($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addi $s1, $s1, +8</td>
<td>sw $t1, 4($s1)</td>
<td>5</td>
</tr>
<tr>
<td>bne $s1, $s3, Loop</td>
<td>nop</td>
<td>6</td>
</tr>
</tbody>
</table>
**Limits of Static Scheduling**

Compiler scheduling for dual-issue MIPS...

```plaintext
lw   $t0, 0($s1)     # load A
addi $t0, $t0, +1  # increment A
sw   $t0, 0($s1)    # store A
lw   $t0, 0($s2)     # load B
addi $t0, $t0, +1  # increment B
sw   $t0, 0($s2)    # store B
```

<table>
<thead>
<tr>
<th>ALU/branch slot</th>
<th>Load/store slot</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>lw   $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addi $t0, $t0, +1</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>nop</td>
<td>sw   $t0, 0($s1)</td>
<td>4</td>
</tr>
<tr>
<td>nop</td>
<td>lw   $t0, 0($s2)</td>
<td>5</td>
</tr>
<tr>
<td>nop</td>
<td>nop</td>
<td>6</td>
</tr>
<tr>
<td>addi $t0, $t0, +1</td>
<td>nop</td>
<td>7</td>
</tr>
<tr>
<td>nop</td>
<td>sw   $t0, 0($s2)</td>
<td>8</td>
</tr>
</tbody>
</table>
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)

Improving IPC via ILP
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
Dynamic Multiple Issue
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 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
Standard Multithreading Picture

Time evolution of issue slots

- Color = thread, white = no instruction

- 4-wide Superscalar
  - Switch to thread B on thread A L2 miss

- CGMT

- FGMT
  - Switch threads every cycle

- SMT
  - Insns from multiple threads coexist
Q: Does multiple issue / ILP cost much?

A: Yes.

→ Dynamic issue and speculation requires power

<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>
</tbody>
</table>
Curve shows ‘Moore’s Law’: transistor count doubling every two years.
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?

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

Performance
Power
1.0x
1.0x
Single-Core

Performance
Power
1.6x
1.02x
Dual-Core
Underclocked -20%
**Power Efficiency**

Q: Does multiple issue / ILP cost much?

A: Yes.

→ Dynamic issue and speculation requires power

<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.
Inside the Processor

AMD Barcelona Quad-Core: 4 processor cores
Inside the Processor

Intel Nehalem Hex-Core

Memory Controller

Core  Core  Core  Core  Core  Core  Core

Shared L3 Cache

Execution Units

L1 Data Cache

L2 Cache & Interrupt Servicing

Memory Ordering & Execution

Paging

Instruction Reordering, Scheduling & Retirement

Instruction Decode & Microcode

Branch Prediction

Instruction Fetch & L1 Cache

4-wide pipeline
Hyperthreading
Multi-Core vs. Multi-Issue vs. HT

<table>
<thead>
<tr>
<th>Programs:</th>
<th>$N$</th>
<th>$1$</th>
<th>$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Num. Pipelines:</td>
<td>$N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>Pipeline Width:</td>
<td>$1$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

Hyperthreads
- HT = Multilssue + extra PCs and registers – dependency logic
- HT = MultiCore – redundant functional units + hazard avoidance

Hyperthreads (Intel)
- Illusion of multiple cores on a single core
- Easy to keep HT pipelines full + share functional units
Example: All of the above

8 die (aka 8 sockets)
4 core per socket
2 HT per core

Note: a socket is a processor, where each processor may have multiple processing cores, so this is an example of a multiprocessor multicore hyperthreaded system.
Parallel Programming

Q: So lets 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*
Amdahl’s Law
Parallelism is a necessity

Necessity, not luxury
Power wall

Not easy to get performance out of

Many solutions
Pipelining
Multi-issue
Hyperthreading
Multicore
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?
Big Picture: Parallelism and Synchronization

How do I take advantage of *parallelism*?
How do I write *(correct)* parallel programs?

What primitives do I need to implement correct parallel programs?
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?
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.
Parallelism and Synchronization
Parallelism and Synchronization

Each processor core has its own L1 cache
Parallelism and Synchronization

Each processor core has its own L1 cache
Parallelism and Synchronization

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)
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?
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></td>
<td></td>
<td>0</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

Consistency

When a written value will be returned by a read
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)
  $\Rightarrow$ 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 $P_A$ can see $P_2$’s write before $P_1$’s and $P_B$ can see the opposite; their final understanding of state is wrong)
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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CPU B read X</td>
<td></td>
<td></td>
<td></td>
<td></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
Takeaway: Summary of cache coherence

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.
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)
}

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

Very expensive and difficult to maintain 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 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

```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
addiu
write hits

Thread 205
read hits
addiu
write hits

(look familiar?)

Timing-dependent failure ⇒ race condition

- hard to reproduce ⇒ hard to debug
Two threads, one counter

Possible result: lost update!

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

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

```
CSEnter();  # wait
    # 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

```
pthread_mutex_init(&m);
pthread_mutex_lock(&m);
```

```
T1
```

```
hits = hits+1;
```

```
# wait
```

```
T2
```

```
pthread_mutex_lock(&m);
```

```
# wait
```

```
hits = hits+1;
```

```
pthread_mutex_unlock(&m);
```

```
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 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, causing the code to try again.

i.e. use this value 0 in $t0 to try again.
Synchronization in MIPS

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

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

- Succeeds if location not changed since the \textit{LL}
  - Returns 1 in \textit{rt}
- Fails if location is changed
  - Returns 0 in \textit{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>\textit{try: LL \textit{$t0}, 0($s0)}</td>
<td>\textit{try: LL \textit{$t0}, 0($s0)}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\textit{ADDIU \textit{$t0, $t0, 1}}</td>
<td>\textit{ADDIU \textit{$t0, $t0, 1}}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\textit{SC \textit{$t0, 0($s0)}}</td>
<td>\textit{SC \textit{$t0, 0 ($s0)}}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>\textit{BEQZ \textit{$t0, try}}</td>
<td>\textit{BEQZ \textit{$t0, try}}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Mutex from LL and SC

Linked load / Store Conditional

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

\[
\text{mutex_lock(int } *m) \{ \\
\hspace{1em} \text{while(test_and_set(m))}\}\
\]

\[
\text{int test_and_set(int } *m) \{ \\
\hspace{1em} \left\{ \begin{array}{l}
\hspace{1em} \text{old = } *m; \\
\hspace{1em} *m = 1;
\end{array} \right\} \quad \text{LL Atomic} \\
\hspace{1em} \text{return old;}
\}
\]
Mutex from LL and SC

Linked load / Store Conditional

$\texttt{m} = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked$

\texttt{mutex\_lock(int *m) { \\
    \hspace{1em} \texttt{while(test\_and\_set(m)){} \\
    } \\
}}$

\texttt{int test\_and\_set(int *m) { \\
\hspace{1em} \texttt{try: LI $t0, 1} \\
\hspace{2em} \texttt{LL $t1, 0($a0)} \\
\hspace{2em} \texttt{SC $t0, 0($a0)} \\
\hspace{2em} \texttt{BEQZ $t0, try} \\
\hspace{2em} \texttt{MOVE $v0, $t1} \\
\hspace{1em} \texttt{}}$


Mutex from LL and SC

Linked load / Store Conditional

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

\[
\begin{align*}
\text{mutex\_lock}(\text{int } *m) \{ \\
\quad &\text{while(test\_and\_set(m))}\{} \\
\quad &\text{}\}
\end{align*}
\]

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

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

Linked load / Store Conditional

\[
m = 0;
\]

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

mutex_unlock(int *m) {
    SW $zero, 0($a0)
}
### 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>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

```c
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)
}
```

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>
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);
}
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?
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;
}
```
Attempt#1: Producer/Consumer

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;
}
```

```plaintext
head

```

tail
```plaintext

```

[Diagram of a circular buffer with positions 1 to 4, head and tail pointers]
```
### Attempt#2: Protecting an invariant

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

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

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

// consumer: take from list head
```c
char get() {
    pthread_mutex_lock(m);
    while(h == t) {}
    char c = A[h];
    h = (h+1)%n;
    pthread_mutex_unlock(m);
    return c;
}
```
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);`  Circular
  `lock(m2);`  `lock(m1);`  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

```c
char get() {
    acquire(L);
    char c = A[h];
    h = (h+1)%n;
    release(L);
    return c;
}
```
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() {
    acquire(L);
    while (h == t) { };
    char c = A[h];
    h = (h+1)%n;
    release(L);
    return c;
}
```
Attempt#4: 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() {
    do {
        acquire(L);
        empty = (h == t);
        if (!empty) {
            c = A[h];
            h = (h+1)%n;
        }
    } while (empty);
    release(L);
} while (empty);
return c;
}
```
Language-level Synchronization
Condition variables

Use [Hoare] a condition variable to wait for a condition to become true (without holding lock!)

wait(m, c) :
  • atomically release $m$ and sleep, waiting for condition $c$
  • wake up holding $m$ sometime after $c$ was signaled

signal(c) : wake up one thread waiting on $c$
broadcast(c) : wake up all threads waiting on $c$

POSIX (e.g., Linux): pthread_cond_wait,
pthread_cond_signal, pthread_cond_broadcast
Attempt#5: Using a condition variable

wait(m, c) : release m, sleep until c, wake up holding m
signal(c) : wake up one thread waiting on c

cond_t *not_full = ...;
cond_t *not_empty = ...;
mutex_t *m = ...;

void put(char c) {
    lock(m);
    while ((t-h) % n == 1)
        wait(m, not_full);
    A[t] = c;
    t = (t+1) % n;
    unlock(m);
    signal(not_empty);
}

char get() {
    lock(m);
    while (t == h)
        wait(m, not_empty);
    char c = A[h];
    h = (h+1) % n;
    unlock(m);
    signal(not_full);
    return c;
}
Monitors

A Monitor is a concurrency-safe datastructure, with...

- one mutex
- some condition variables
- some operations

All operations on monitor acquire/release mutex

- one thread in the monitor at a time

Ring buffer was a monitor
Java, C#, etc., have built-in support for monitors
Java concurrency

Java objects can be monitors

- “synchronized” keyword locks/releases the mutex
- Has one (!) builtin condition variable
  - o.wait() = wait(o, o)
  - o.notify() = signal(o)
  - o.notifyAll() = broadcast(o)

- Java wait() can be called even when mutex is not held. Mutex not held when awoken by signal(). Useful?
More synchronization mechanisms

Lots of synchronization variations...
(can implement with mutex and condition vars.)

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

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, ...