Shared Memory Architecture

Shared Memory Multiprocessor

**Single global memory space** \(0, \ldots, A - 1\)
- Physically partitioned into \(M\) physical devices
- All CPUs access full memory space via interconnection network

**CPUs communicate via shared memory addresses**
- Standard write / read transactions to shared addresses

**Synchronize memory access**
- Prevent data hazards

```
0       \cdots
CPU     \cdots
N - 1

\begin{array}{c}
\text{Switching} \\
\text{Fabric}
\end{array}
```

```
0, \ldots, (A/M) - 1 \\
0 ... (M - 1)(A/M), \ldots, A - 1
```

**Contemporary Trend**

**Symmetric Multiprocessor (SMP)**
- \(N\) equivalent microprocessors
- Multiple processor cores on single integrated circuit
- Communication network between processors

**Thread Level Parallelism (TLP)**
- Operating system runs in one processor
- OS assigns threads to processors by some scheduling algorithm

**Uniform memory access (UMA)**
- Shared memory accessible by all processors
- Access time to any memory location uniform for all processors
- Symmetric multiprocessor
  - All processors are of equivalent type
  - All processors have symmetric relationship to memory
  - Typically fast bus-based network (SCSI or PCI)

**Non-uniform memory access (NUMA)**
- Memory distributed among processors
  - Each processor physically attached to part of shared memory
  - Cost-effective scaling to large systems
- Access time to memory address depends on distance to processor
  - Most memory accesses are local and fast

**Cache-only memory architecture (COMA)**
- Each processor has part of shared memory as private cache
- Requires that data be migrated to processor requesting it
Open MP for Shared Memory Systems

**Application Program Interface (API) for multiprocessing**
- Supports shared memory applications in C/C++ and Fortran
- Directives for explicit thread-based parallelization
- Simple programming models on shared memory machines

**Fork — Join Model**
- Master thread (consumer thread)
  - Programs initiate as single thread
  - Executes sequentially until parallel construct is encountered
- Fork producer threads
  - Master thread creates team of parallel threads
  - Program statements in parallel construct execute in parallel
- Join producer threads
  - Synchronize and terminate
  - Master thread continues

**Nesting**
- Forks can be defined within parallel sections

Ref: https://computing.llnl.gov/tutorials/openMP/

**General Code Structure**

```c
#include <omp.h>
main () { int var1, var2, var3; /* Serial code */
...
#pragma omp parallel private(var1, var2) shared(var3)
{
 /* Parallel section executed by all threads */
 ...
 /* All threads join master thread and disband */
 }
/* Resume serial code */
...
```

**Selected Options for ParallelPragma**

- **if** (*scalar_expression*)
  - Performs conditional multithreading

- **private**(list)
  - Creates copy of variable types for each thread

- **shared**(list)
  - Single shared copy of variables in list
  - Requires synchronization of memory accesses to avoid hazards

- **reduction**(operator: list)
  - Performs join operation on list of private variables

- **num_threads**(integer-expression)
  - Specifies number of threads to create

"Hello Worlds" Program

```c
#include <omp.h>
main () { int nthreads, tid;
/* Fork team of threads with private variables */
#pragma omp parallel private(tid)
{
 /* Obtain and print thread id */
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
/* Only master thread does this */
if (tid == 0)
{
 nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
/* All threads join master thread and terminate */
}
**Parallel For Directive**

```c
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main ()
{
    int i, chunk;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;
    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    }  /* end of parallel section */
}
```

**Options**

- **static**
  - Loop iterations divided into chunk pieces size
  - Statically assigned to threads
  - Chunk is not specified — iterations are evenly divided

- **dynamic**
  - Loop iterations dynamically scheduled
  - When thread finishes chunk it is dynamically assigned another

- **ordered**
  - Iterations executed as in a serial program

- **nowait**
  - No synchronization at end of parallel loop
  - Threads do not wait for other threads to finish at end of private code

**Parallel For Example**

```c
#include <omp.h>
define N 1000
main ()
{
    int i;
    float a[N], b[N], c[N], d[N];
    for (i=0; i < N; i++)
        a[i] = i * 1.5;
    b[i] = i + 22.35;
    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {#pragma omp section
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
        #pragma omp section
        for (i=0; i < N; i++)
            d[i] = a[i] * b[i];
        } /* end of sections */
    } /* end of parallel section */
}
```

**Sections Directive**

```c
#include <omp.h>
#define N 1000
main ()
{
    int i;
    float a[N], b[N], c[N], d[N];
    for (i=0; i < N; i++)
        a[i] = i * 1.5;
    b[i] = i + 22.35;
    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++)
                c[i] = a[i] + b[i];
            #pragma omp section
            for (i=0; i < N; i++)
                d[i] = a[i] * b[i];
            } /* end of sections */
        } /* end of parallel section */
    }
```
Race Conditions

Race condition
- Data hazard caused by parallel access to shared memory

Example
```c
#pragma omp parallel shared(x) num_threads(2)
{
    x = x + 1;
}
```
- Two threads should increment x independently: $x \leftarrow x + 2$

Interleaved execution sequence for $x = 5$

1. **Thread 1** — R1 $\leftarrow x$ ; CPU1 loads copy of $x = 5$
2. **Thread 2** — R1 $\leftarrow x$ ; CPU2 loads copy of $x = 5$
3. **Thread 1** — R1 $\leftarrow R1 + 1$ ; CPU1 updates R1 $\leftarrow 6$
4. **Thread 2** — R1 $\leftarrow R1 + 1$ ; CPU2 updates R1 $\leftarrow 6$
5. **Thread 1** — x $\leftarrow R1$ ; CPU1 writes x $\leftarrow 6$
6. **Thread 2** — x $\leftarrow R1$ ; CPU2 writes x $\leftarrow 6$

Program completes with result $x \leftarrow 6$ instead of $x \leftarrow 7$

Synchronization

Directives to control access to shared data among threads

- **#pragma omp master**
  - Only master thread (thread 0) performs following {block}

- **#pragma omp critical**
  - Only one thread can execute following {block} at a time
  - Other threads wait for thread to leave critical section before entering

- **#pragma omp barrier**
  - Each thread reaching barrier waits until all threads reach barrier

- **#pragma omp atomic**
  - Memory update in next statement must be completed atomically
  - Mini-critical section for memory write

Preventing Race Condition

Example
```c
#pragma omp parallel shared(x) num_threads(2)
{
    #pragma omp critical
    x = x + 1;
}
```

Execution sequence with critical section for $x = 5$

1. **Thread 1** — R1 $\leftarrow x$ ; CPU1 loads copy of $x = 5$
2. **Thread 2** blocks until thread 1 completes
3. **Thread 1** — R1 $\leftarrow R1 + 1$ ; CPU1 updates R1 $\leftarrow 6$
4. **Thread 1** — x $\leftarrow R1$ ; CPU1 writes x $\leftarrow 6$
5. **Thread 1** completes $\Rightarrow$ thread 2 unblocked
6. **Thread 2** — R1 $\leftarrow x$ ; CPU2 loads copy of $x = 6$
7. **Thread 2** — R1 $\leftarrow R1 + 1$ ; CPU2 updates R1 $\leftarrow 7$
8. **Thread 2** — x $\leftarrow R1$ ; CPU2 writes x $\leftarrow 7$

Program completes with result $x \leftarrow x + 2$

Performance implications — critical section runs sequentially

Reduction

Reduction (operator: list)
- Performs join operation on list of private variables
- Operation can be $+, -, \&, ^, |, &&, ||$

Example
```c
#pragma omp parallel for reduction(+:sum)
for (i=0; i < N; i++)
{
    sum+ = a[i] * b[i];
}
```

- Each thread has private copy of variable sum
- On join (end of parallel construct)
  - Private copies of sum combined by addition ($+$)
  - Result copied into master thread copy of sum
**Bus-Based Symmetric Multiprocessor (SMP)**

**Bootstrap Processor (BSP)**
Chosen by BIOS to perform system startup and run OS

**Application Processors (AP)**
In halt state unless allocated application threads by OS

---

**Cache Organization for Dual Core Processors**

**Pentium D**
Dual core processor
Each core has private L1 D+I cache and L2 D+I cache

**Core Duo, Core 2, i3, i5, i7, ...**
Dual core processor
Each core has private L1 data cache
Cores share L2 cache
No L1 instruction cache
Instructions fetched directly from L2
Employ trace caching instead

---

**Communication Overhead in SMP**

```c
#pragma omp parallel
{
    #pragma omp for
    for (i=0; i<4; i++)
    c[i] = a[i] + b[i];
}
```

**Sequential program requires**

- 4 × load-add-store

**Parallel program requires**

- 4 × update instruction cache over serial bus
- 1 × load-add-store

In most architectures cache updates more expensive than arithmetic
Shared Memory Program — Vector Product

Compute $\sum_{i=0}^{3} a[i] \times b[i]$ from data in shared memory

### Sequential Code

```c
for (i=0; i<4; i++){
    load Ra, a[i]
    load Rb, b[i]
    Ra ← Ra * Rb
    add Racc, Ra
}
store p, Racc
```

### Parallel Code

fork 4 threads with private i=0,1,2,3

```c
load Ra, a[i]
load Rb, b[i]
Ra ← Ra * Rb
store p[i], Ra
```

fork 2 threads with private i=0,2

```c
load Ra, p[0]
load Rb, p[2]
Ra ← Ra + Rb
store p, Ra
```

CC ~ overhead + 4 x 4 + 1

Neglecting overhead

$$S = \frac{17}{12} = 1.42$$

### Capacity Example

**Standard PCI express bus — 16 bytes per cycle at 1 GHz**

**Average data access rate depends on**

- Integer width
- Loads per instruction

<table>
<thead>
<tr>
<th>Instruction/sec</th>
<th>(seconds per CC)</th>
<th>(CC per instruction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\frac{1}{1}$</td>
<td></td>
</tr>
</tbody>
</table>

**Miss rate depends on number of data reads between cache updates**

- Compute dominated $M \sim 0.01$
- Communication dominated $M \sim 0.1$

$$N \leq \frac{RW}{DM}$$

Where:

- $N = \text{CPUs}$
- $D_s = \text{average data access rate (bytes per second)}$
- $M = \text{cache miss rate = inter-CPU access rate}$
- $R_s = \text{transfer rate (transfer cycles per second)}$
- $W = \text{transfer width (bytes per transfer)}$

**Core Duo vs. Pentium M**

**Single-threaded SPEC code performance**

<table>
<thead>
<tr>
<th>Cint</th>
<th>Cfp</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image-url" alt="Graph image" /></td>
<td><img src="image-url" alt="Graph image" /></td>
</tr>
</tbody>
</table>
Multi-Threading Performance

Core Duo vs. Pentium M

Multithreaded code performance

Effect of Threading Model

Data domain decomposition

Divide large data arrays among processors
Works well with shared memory SMP

Functional domain decomposition

Divide tasks among processors
Tends toward load balancing problems on SMP
Works better with work queue systems on message passing systems

Data Hazards in Shared Memory SMP

Three levels of data hazard in shared memory systems

Program level
Concurrent programming of inherently sequential operations
Handled with programmed synchronization directives
Atomic read/write, critical sections, barrier

Cache consistency
Multiple processors writing shared copies of data
Protocols for maintaining valid copies of data values

Hardware level consistency
Instruction-level memory semantics are abstractions
Real hardware operates in more complex manner

General approach to handle hazards
Enforce operational definition for consistency
Enables unambiguous program validation

Sequential Consistency

Strict consistency
Memory operations performed in order intended by programmer
Possible to implement only on single processor systems

Sequential consistency (Lamport 1979)
Clear, consistent, repeatable definition of execution order
Result of any execution identical to result of execution in which:
Operations on all processors executed in specified sequential order
Operations on each processor appear in order specified by program
Specified sequential order
Any well-defined interleaving
Example — round robin

Implications for programmer
Unsynchronized threads assumed to execute interleaved
Cache assumed to enforce write-order consistency
Hardware assumed to enforce read/write-order consistency

Implementing Critical Section with Semaphore

**Semaphore**

Unsigned number with 2 operations

\[ P(s): \begin{cases} s \leftarrow s - 1, & s > 0 \\ wait, & s = 0 \end{cases} \]

\[ V(s): s \leftarrow s + 1 \]

**Binary semaphore**

Mutual exclusion (mutex) or lock

\( s \) initialized to 1

**Critical section**

\[ P(s) /* section begins (s=1) or blocks (s=0) */ \]

\[ x = x + 1; \]

\[ V(s) /* s \leftarrow 1 permits another thread to operate */ \]

**Difficulty**

Requires system-wide atomic semaphore operation

Impractical to disable all system interrupts and interleaving during P and V

---

**Shared Variable Lock**

**Shared variables Flag1 and Flag2 initialized to zero**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
|\[
\text{Flag1} = 1; \\
\text{loop: if (Flag2 == 1) loop} \\
\text{<critical section>} \\
\text{Flag1 = 0;}
\]
|\[
\text{Flag2} = 1; \\
\text{loop: if (Flag1 == 1) loop} \\
\text{<critical section>} \\
\text{Flag2 = 0;}
\]

**Spin-loop**

Loop instruction repeats until condition clears

**Interleaved execution**

Actual execution order \( t_1, t_2, t_3, t_4 \)

Creates deadlock

---

**Modified Shared Variable Lock**

**Shared variables Flag1 and Flag2 initialized to zero**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
|\[
\text{Loop: Flag1 = 1; } \\
\text{if (Flag2 == 1) } \\
\text{if (Flag1 == 1) } \\
\text{Flag2 = 0; loop} \\
\text{<critical section>} \\
\text{Flag1 = 0;}
\]
|\[
\text{Loop: Flag2 = 1; } \\
\text{if (Flag1 == 1) } \\
\text{Flag1 = 0; loop} \\
\text{<critical section>} \\
\text{Flag2 = 0;}
\]

**Interleaved execution**

Order \( t_1, t_2, t_3, t_4 \)

No deadlock but possible livelock if hardware writes not atomic

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
|\[
\text{Loop: Flag1 = 1; } t_1 \\
\text{if (Flag2 == 1) } \\
\text{if (Flag1 == 1) } \\
\text{Flag1 = 0; loop} \\
\text{<critical section>} \\
\text{Flag1 = 0;}
\]
|\[
\text{Loop: Flag2 = 1; } t_2 \\
\text{if (Flag1 == 1) } \\
\text{Flag2 = 0; loop} \\
\text{<critical section>} \\
\text{Flag2 = 0;}
\]

---

**Dekker Algorithm**

**Shared variables Flag1, Flag2 = 0 ; turn = 1**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
|\[
\text{Flag1 = 1; } \\
\text{turn = 1; } \\
\text{loop: if (Flag2 == 1 and turn == 1) loop} \\
\text{<critical section>} \\
\text{Flag1 = 0;}
\]
|\[
\text{Flag2 = 1; } \\
\text{turn = 2; } \\
\text{loop: if (Flag1 == 1 and turn == 2) loop} \\
\text{<critical section>} \\
\text{Flag2 = 0;}
\]

**Interleaved execution**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
|\[
\text{Flag1 = 1; } t_1 \\
\text{turn = 1; } t_3 \\
\text{loop: if (Flag2 == 1 and turn == 1) loop} \\
\text{t4} \\
\text{<critical section>} \\
\text{Flag1 = 0;}
\]
|\[
\text{Flag2 = 1; } t_2 \\
\text{turn = 2; } t_4 \\
\text{loop: if (Flag1 == 1 and turn == 2) loop} \\
\text{t6} \\
\text{<critical section>} \\
\text{Flag2 = 0;}
\]

**No deadlock or livelock**

Generalization to \( n > 2 \) threads — Lamport bakery algorithm
### Machine Language Support

**Atomic instruction primitives in processor ISA**
- Provide hardware-level semaphore $M$ for well-defined atomic memory access
- Enable implementation of atomic constructs at compiler level

<table>
<thead>
<tr>
<th>Instruction primitive</th>
<th>Application</th>
</tr>
</thead>
</table>
| **Test_and_Set** $M$, $R$ | **L1:** Test_and_Set $M$, $R1$ ; spinlock  
  $\text{BNEZ } R1$, L1  
  **<critical section>**  
  Swap $M$, $R1$ |
| **Fetch_and_Add** $M$, $R1$, $R2$ | ADDI $R2$, $R0$, #1  
  $\text{L1:** Fetch_and_Add } M$, $R1$, $R2$  
  $\text{BNEZ } R1$, L1  
  **<critical section>**  
  Swap $M$, $R1$ |
| **Swap** $M$, $R$ | $\text{Regs}[R_{\text{temp}}] \leftarrow Mem[M]$  
  $\text{Mem}[M] \leftarrow \text{Regs}[R]$  
  $\text{Regs}[R] \leftarrow \text{Regs}[R_{\text{temp}}]$ |

**Compare and Swap (CAS)**
- **Swaps** $M$ and $R2$ if $M = R1$
  - **Compare_and_Swap** $M$, $R1$, $R2$  
    - if $(\text{Regs}[R1] == \text{Mem}[M])$
      - $\text{Mem}[M] = \text{Regs}[R2]$  
      - $\text{Regs}[R2] = \text{Regs}[R1]$  
      - $\text{Cflag} \leftarrow 1$
    - else $\text{Cflag} \leftarrow 0$
- **No lock**
  - Non-blocking atomic operation more efficient (M. Herlihy — 1991)

<table>
<thead>
<tr>
<th>Critical Section</th>
</tr>
</thead>
</table>
| **L1:** LW $R1$, $x$ ; load $x$  
  ADDI $R2$, $R1$, #1 ; prepare new value for $x$  
  CAS $x$, $R1$, $R2$ ; if no change in stored $x$, update $x$  
  BEQZ Cflag, L1 ; else start again |

**Load Reserve and Store Conditional**

**Load-reserve**
- Returns current value of memory location
- Associates reservation flag with address
- Flag can be reset by subsequent load-reserve

**Store-conditional**
- Performs write if reservation flag still set
- **Stronger than compare and swap**
  - Prevents store in location that was written after read
  - Even if original value was restored

<table>
<thead>
<tr>
<th>Instruction primitive</th>
<th>Application</th>
</tr>
</thead>
</table>
| **load-reserve** $R$, $M$ | **L1:** load-reserve $R1$, $x$  
  ADDI $R1$, $R1$, #1  
  store-conditional $x$, $R1$  
  BEQZ status, L1 |
| **store-conditional** $M$, $R$ | **L1:** load-reserve $R1$, $x$  
  ADDI $R1$, $R1$, #1  
  store-conditional $x$, $R1$  
  BEQZ status, L1 |

**Hardware Level Consistency without Cache**

**Instruction-level memory semantics**
- ISA defines unambiguous sequential actions
  - LW $R1$, $32(R2)$ $\Rightarrow$ $\text{Regs}[R1] \leftarrow \text{Mem}[\text{Regs}[R2] + 32]$  
  - SW $32(R2)$, $R1$ $\Rightarrow$ $\text{Mem}[\text{Regs}[R2] + 32] \leftarrow \text{Regs}[R1]$ |

**Hardware-level implementation**
- Hardware optimizations alter presumed order of instruction actions
- Instruction behavior meets definition on single processor machines
- Can create hazards in multiprocessors

**Typical hazards**
- Single processor may reorder data reads and writes
- Multiple processors may reorder reads and writes among themselves
  - Multiple processors write to multiple memory devices simultaneously
**Example — Write Buffer with Bypass**

**Write buffer**
- Processor writes updates to memory through FIFO buffer
- Prevents memory write latency from stalling pipeline

**Memory reordering**
- Write buffer permits reads to bypass previous writes
- Prevents memory write queue from stalling subsequent reads

**Forwarding**
- Read hit to address in write buffer updates directly from FIFO
- No data hazards for single processor system

**Violates sequential coherency in multiprocessor system**
- Example — Dekker algorithm with bypassing write buffer
  - Load `other_flag` bypasses
  - Store `my_flag` in both processors
  - Load `other_flag` always returns 0

**Example — Write Overlapping**

**Multiprocessor system with general interconnect network**
- Crossbar or multiple bus network
- Memory partitioned into multiple physical devices
- Permits multiple memory writes per transfer cycle

**Violates sequential coherency in multiprocessor system**
- Example
  - `#pragma omp critical
   \[ x = x + 1; \]
  - Critical section implemented with Dekker algorithm (not CAS)
  - Thread 1
    - Issues write instructions to `x` and `Flag1
      releasing lock
    - Write to `x` delayed in network
    - Write to `Flag1` completes
  - Thread 2 completes
  - Thread 1 write to `x` completes after Thread 2 write to `x`

**Example — Read/Write Reordering**

**Multiprocessor system with general interconnect network**
- Permits multiple memory writes per transfer cycle

**Violates sequential coherency in multiprocessor system**
- Example `#pragma omp critical
\[ x = x + 1; \]
- Critical section implemented with Dekker algorithm (not CAS)
- Thread 1 runs out-of-order but Thread 2 runs in-order

**Memory Fences**

**Memory barrier (membar)**
- Machine-level instruction inserted by programmer or compiler
- Enforces sequential consistency in rescheduling
- Instructions are not moved past a membar

**Example**

```
loop: BNEZ R4, loop
  LW R5, [x]
  ADDI R5, R5, #1
  SW [Flag1], R0
```

```
Processor will not execute load before memory barrier
```

---

**Thread 1 Listing**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI R1, R0, #1</td>
<td></td>
</tr>
<tr>
<td>SW [Flag1], R1</td>
<td></td>
</tr>
<tr>
<td>SW [turn], R1</td>
<td></td>
</tr>
<tr>
<td>LW R2, [Flag2]</td>
<td></td>
</tr>
<tr>
<td>LW R3, [turn]</td>
<td></td>
</tr>
<tr>
<td>BNEZ R4, loop</td>
<td></td>
</tr>
<tr>
<td>LW R5, [x]</td>
<td></td>
</tr>
<tr>
<td>ADDI R5, R5, #1</td>
<td></td>
</tr>
<tr>
<td>SW [x], R5</td>
<td></td>
</tr>
<tr>
<td>SW [Flag1], R0</td>
<td></td>
</tr>
</tbody>
</table>

**Thread 1 Out-of-Order Execution**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R5, [x]</td>
<td></td>
</tr>
<tr>
<td>ADDI R1, R0, #1</td>
<td></td>
</tr>
<tr>
<td>SW [Flag1], R1</td>
<td></td>
</tr>
<tr>
<td>SW [turn], R1</td>
<td></td>
</tr>
<tr>
<td>LW R2, [Flag2]</td>
<td></td>
</tr>
<tr>
<td>LW R3, [turn]</td>
<td></td>
</tr>
<tr>
<td>BNEZ R4, loop</td>
<td></td>
</tr>
<tr>
<td>LW R5, [x]</td>
<td></td>
</tr>
<tr>
<td>ADDI R5, R5, #1</td>
<td></td>
</tr>
<tr>
<td>SW [x], R5</td>
<td></td>
</tr>
<tr>
<td>SW [Flag1], R0</td>
<td></td>
</tr>
</tbody>
</table>

---

**Dynamic rescheduling**
Cache Policies in Single CPU System

**Read miss**
Update cache (copy block from memory)

**Write miss**
Write allocate policy
Update cache
No write allocate
No cache update
Write data directly to memory

**Write to cache**
Write through
Update memory (copy block from cache) on every cache write
Maintains cache-memory consistency
Write back
Update memory when block swapped out
Requires fewer memory bus operations

Cache Consistency and Coherency

**Multiple processors access shared data in main memory**
Each processor copies data blocks to private cache
Caches and main memory become inconsistent

**Example**
CPU-0 reads X to cache from main memory
CPU-1 reads X to cache from main memory
CPU-0 writes to X in cache
Copies of X in CPU-1 cache and main memory not valid

**Strict consistency**
All copies of any data location are identical
Almost impossible to implement

**Sequential consistency**
Operations on all processors executed in specified sequential order
Operations on each processor appear in order specified by program

**Cache coherency**
Implementation of sequential consistency for cache and main memory
Requires processor hardware support for cache coherency management

Cache Inconsistency in Write Back Policy

**Concurrent threads**
`#pragma omp critical`
`x = x + 1;`
Each thread enforces sequential consistency with its private cache
Criticality not enforced on all copies of `x`

**Example**

<table>
<thead>
<tr>
<th>CPU-0</th>
<th>Cache 0</th>
<th>Memory</th>
<th>Cache 1</th>
<th>CPU-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R5, [x]</td>
<td>x = 2</td>
<td>x = 2</td>
<td>x = 2</td>
<td>LW R5, [x]</td>
</tr>
<tr>
<td>ADDI R5, R5, #1</td>
<td>x = 2</td>
<td>x = 2</td>
<td>x = 2</td>
<td>ADDI R5, R5, #1</td>
</tr>
<tr>
<td>SW [x], R5</td>
<td>x = 3</td>
<td>x = 2</td>
<td>x = 2</td>
<td>SW [x], R5</td>
</tr>
<tr>
<td>LW R6, [y]; write back x</td>
<td>y = 10</td>
<td>y = 10</td>
<td>y = 10</td>
<td>LW R6, [y]; write back x</td>
</tr>
</tbody>
</table>

Cache Inconsistency in Write Through Policy

**Concurrent threads**
`#pragma omp critical`
`x = x + 1;`
Each thread enforces sequential consistency with its private cache
Criticality not enforced on all copies of `x`

**Example**

<table>
<thead>
<tr>
<th>CPU-0</th>
<th>Cache 0</th>
<th>Memory</th>
<th>Cache 1</th>
<th>CPU-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R5, [x]</td>
<td>x = 2</td>
<td>x = 2</td>
<td>x = 2</td>
<td>LW R5, [x]</td>
</tr>
<tr>
<td>ADDI R5, R5, #1</td>
<td>x = 2</td>
<td>x = 2</td>
<td>x = 2</td>
<td>ADDI R5, R5, #1</td>
</tr>
<tr>
<td>SW [x], R5</td>
<td>x = 3</td>
<td>x = 3</td>
<td>x = 3</td>
<td>SW [x], R5</td>
</tr>
</tbody>
</table>
Cache Coherence Protocols

**Cache coherence**
Enforce sequential consistency for all caches and main memory
Enables system-wide atomic and critical constructs

**Snoopy cache**
Cache blocks tags with status bits depending on coherence policy
Status depends on access history to data in block
Change in status can initiate write back before usual block eviction

Cache manager
Monitors all addresses written on system bus by all processors
Compares addresses with blocks in cache
Updates state of cached block on address hit

![Diagram showing CPU and Memory connections]

**State Definitions**

**Possible states of data block**
Indicated by status bits in block tag

**Modified**
Unique valid copy of block
Block has been modified since loading

**Owned**
Device is cache is owner of block
Device services requests by other processors for block

**Exclusive**
Unique valid copy of block
Block has not been modified since loading

**Shared**
Block held by multiple caches in system

**Invalid**
Block must be reloaded before next access

**Cache Coherence Policy Types**

**Write update**
Processor W places write address on bus
Snooping processors Sᵢ mark copy of block invalid on address hit
W writes to cache and to bus
Sᵢ update local copies of block and mark updated blocks valid
Creates large bus overhead

**Write invalidate with inquire**
Processor W places write address on bus
Snooping processors Sᵢ mark copy of block invalid on address hit
W writes to local cache — creates unique valid copy of data block
Inquire — Sᵢ place address on bus to access copy of block in W
Lower bus overhead

**Typical implementation in shared memory system**
Write allocate — W loads to cache on write miss
Write invalidate — Sᵢ invalidate cache block on write by W

**Processor Behavior — Modified**

**Processor W cache holds unique valid copy of data block**
Block is dirty
Memory copy is different from W copy
Cache copies in other processors Sᵢ marked Invalid
Sᵢ inquire (request update from this copy) before accessing block

**W can continue to update this copy**
No memory update on cache writes
Memory update on cache swap or inquire

**W responds to inquire**
Snoops memory address from Sᵢ on bus
W updates memory
Marks block invalid on write inquire
Processor Behavior — Shared

**Processor W cache holds one of many valid copies of data line**

- Block is clean
- Memory copy is same as W copy
- Other processors $S_i$ may have copies of block

**W can update this copy**

- Writes update address to memory bus
- Processors $S_i$ mark block invalid
- Updates cache
- W marks block Modified

Processor Behavior — Invalid

**Invalid**

- Block must be reloaded before next read
- State includes blocks tagged invalid and blocks not in cache
- Read and write accesses are cache misses
- Write allocate
  - Write miss at W initiates cache update
  - No write allocate
  - Write miss at W does not initiate cache update
  - Write directly to memory

MSI Protocol State Diagram

- Modified
- Shared
- Invalid

MSI Example

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Read</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>5</td>
<td>Read</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Write</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Write</td>
</tr>
<tr>
<td>8</td>
<td>Write</td>
<td></td>
</tr>
</tbody>
</table>

P1 cache state

- Load on Write
- Miss at P1

P2 cache state

- Load on Write
- Miss at P2
Processor Behavior — Exclusive

Processor W cache holds only valid copy of data line

- Line is clean
- Memory copy is same as W copy
- Cache copies in other processors S_i marked invalid
  - S_i inquire (request update from this copy) before accessing block

W responds to inquire

- Snoops memory address from S_i on bus
- W updates memory
- Marks block invalid on write inquire

Improved performance on private data

- No write back to memory on eviction of exclusive block

### MESI Protocol State Diagram

![MESI Protocol State Diagram]

### MESI Example

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read x</td>
<td>Read x</td>
</tr>
<tr>
<td>2</td>
<td>Read y</td>
<td>Write x</td>
</tr>
<tr>
<td>3</td>
<td>Read z</td>
<td>Replaces x in cache</td>
</tr>
<tr>
<td>4</td>
<td>Evict y without write back</td>
<td>Read w</td>
</tr>
<tr>
<td>5</td>
<td>Evict x with write back</td>
<td>Read y</td>
</tr>
</tbody>
</table>

P1 cache state

Load on Write Miss at P1

Value of shared variable mutex

- Required to enforce criticality
- Each swap causes
  - Read miss and cache update
  - Write to one local cache (write allocate)
  - Invalidation all other caches (write invalidate)

Inefficient movement of data across bus

- Multiple mutex cache misses/loads creates large overhead
- Overhead much larger than read/writes for critical section
Load-Reserve and Store-Conditional with MSI

**Implement critical section**
Use load-reserve/store-conditional
No shared mutex variable

<table>
<thead>
<tr>
<th>R ← 1</th>
<th>L: load-reserve R, x</th>
</tr>
</thead>
<tbody>
<tr>
<td>L: swap mutex, R;</td>
<td>store-conditional x, R</td>
</tr>
<tr>
<td>BNEZ R, L;</td>
<td>BEQZ status, L</td>
</tr>
</tbody>
</table>

**Each processor has private reservation flag**
Flag set on load-reserve
Flag checked on store-conditional attempt
Snooping cache manager clears flag on write to memory location
Requires restart of load and critical section

**Improved overall efficiency**
Overhead created by mutex → multiple reads of critical variable x
Load-reserve/store-conditional eliminates stores to mutex variables

False Sharing

**False sharing**
Multiple threads access distinct address ranges in same cache block
Write to block causes invalidation in other processors
Creates memory thrashing

**Example**
```c
int counter[THREAD_NUM];
int inc_counter ()
{
    counter[my_tid]++;
    return counter[my_tid];
}
```

**Solution**
Avoid small array accessed by multiple threads
Avoid global data or static data variables in same cache block

Cache Coherence and Hardware Optimizations

**Memory hardware optimizations**
Equivalent to memory semantics within processor

**Cache coherence**
Enforces sequential consistency between processors

**Write buffer with read-bypass**
Cache/memory updates follow cache invalidation
Load cannot bypass store on two processors

**Write overlapping**
Writes invalidate other cache copies
Physical writes occur in correct order

**Read/write reordering**
Writes invalidate other cache copies
Rescheduled read invalidated when other CPU writes to variable

Sun T1 Multiprocessor

**Server processor**
Introduced by Sun Microsystems in 2005
Based on UltraSPARC architecture
Optimized for TLP for online transaction processing (OLTP)

**Core**
8 processor cores
Single-issue — not superscalar
In-order execution — no dynamic scheduling
6 stage pipeline — 5 standard stages plus thread selection

**Threads**
4 threads per core
Fine grained multithreading
Switches thread on each clock cycle
Skips over stalled threads
### Sun T1 Organization

**8 processor cores**
- 1 shared FPU
- Fine grain multithreading
- 4 threads per core
- 1.4 GHz clock

**Private L1 cache**
- 16 KB instruction
- 8 KB data
- 64-byte block
- Every L1 block in L2
- L1 miss penalty = 23 CC

**Shared L2 cache**
- 3 MB in 4 banks
- L2 miss penalty = 110 CC
- Distributed directory in L2
- Maintains coherency among L1 caches

---

### Oracle (Sun) T4 Multiprocessor

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads / core</td>
<td>4</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Threads / chip</td>
<td>32</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Pipeline model</td>
<td>In-order</td>
<td>Dynamic scheduling</td>
<td></td>
</tr>
<tr>
<td>Pipeline stages</td>
<td>6</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Shared EUs</td>
<td>1 FPU</td>
<td>1 FPU + 1 Cryptographic</td>
<td></td>
</tr>
<tr>
<td>Clock rate</td>
<td>1.4 GHz</td>
<td>3.0 GHz</td>
<td></td>
</tr>
<tr>
<td>L1 cache</td>
<td>8 x 16 KB</td>
<td>8 x 16 KB I + 8 x 16 KB D</td>
<td></td>
</tr>
<tr>
<td>L2 cache</td>
<td>8 BB</td>
<td>8 x 128 KB</td>
<td></td>
</tr>
<tr>
<td>L3 cache</td>
<td>3 MB</td>
<td>4 MB</td>
<td></td>
</tr>
</tbody>
</table>

**Oracle T4 server**

- 4 SPARC T4 CPUs (256 threads) + 1 TB RAM
- 25% faster than HP 4 Xeon server on OLTP benchmark TPC-H