Shared Memory Multiprocessors

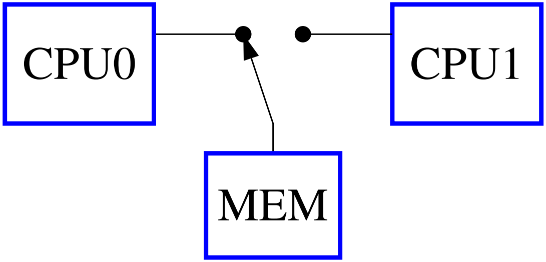
Mark Greenstreet

CpSc 418 – Oct. 4, 2012

Outline:

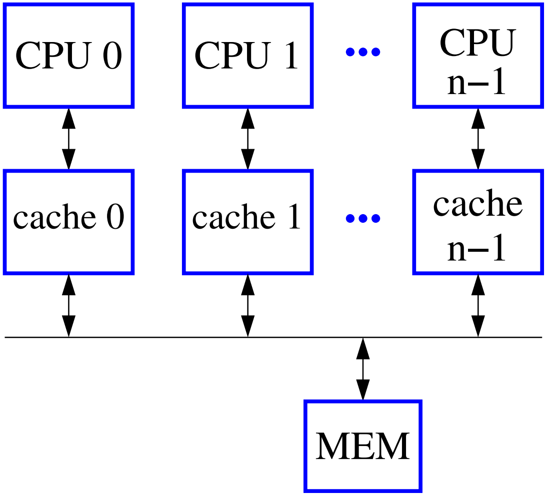
- Shared-Memory Architectures
- Memory Consistency
- Examples

An Ancient Shared-Memory Machine



- Multiple CPU's (typically two) shared a memory
- If both attempted a memory read or write at the same time
  - One is chosen to go first.
  - Then the other does it's operation.
  - That's the role of the switch in the figure.
- By using multiple memory units (partitioned by address), and a switching network, the memory could keep up with the processors.
- But, now that processors are 100's of times faster than memory, this isn't practical.

A Shared-Memory Machine with Caches



- Caches reduce the number of main memory reads and writes.
- But, what happens when a processor does a write?

Today's Story Line

- Shared memory is wonderful:
  - Now, you don't have to bundle up your data structures as messages.
  - Just share a pointer.
- But, what about concurrent accesses?
  - How do I know that you're done building a data structure before I try to use it?
  - What if we have a dynamically changing data structure?
    - ★ How do I make sure that I don't change something when you're in the middle of using it.

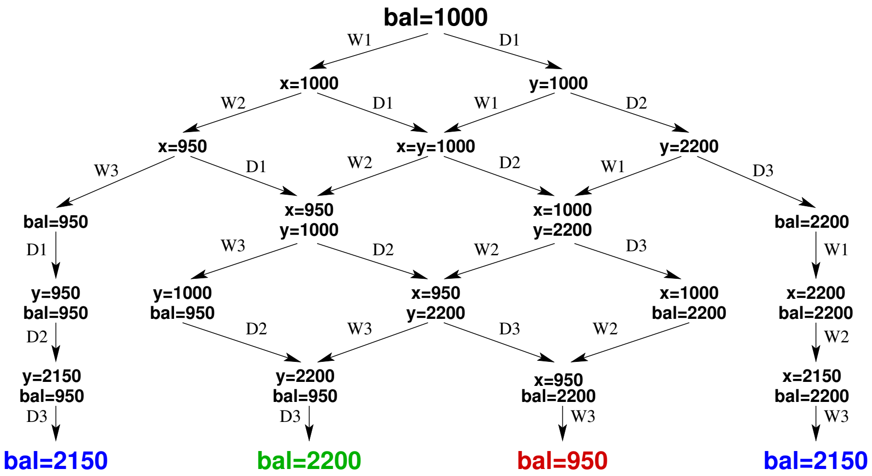
Simple Example: a bank account

	ATM withdrawal		Payroll deposit
	`atm(acct, amt) {`		`pay(acct, amt) {`
W1:	`x = acct.bal;`	D1:	`y = acct.bal;`
W2:	`x = x - amt;`	D2:	`y = y + amt;`
W3:	`acct.bal = x;`	D3:	`acct.bal = y;`
	`}`		`}`

What happens if a withdrawal and deposit happen “at the same time”?

Concurrent withdrawl and deposit

- Given a starting balance of $1,000,
- concurrently withdraw $50 from an ATM while receiving a payroll deposit of $1,200.



Dekker's Algorithm

Problem statement: ensure that at most one thread is in its critical section at any given time.

thread 0:

```
PC_0= 0: while(true) {
PC_0= 1:   non-critical code
PC_0= 2:   flag[0] = true;
PC_0= 3:   while(flag[1]) {
PC_0= 4:     if(turn != 0) {
PC_0= 5:       flag[0] = false;
PC_0= 6:       while(turn != 0);
PC_0= 7:       flag[0] = true;
PC_0= 8:     }
PC_0= 9:   }
PC_0=10:   critical section
PC_0=11:   turn = 1;
PC_0=12:   flag[0] = false;
PC_0=13: }
```

thread 1:

```
PC_1= 0: while(true) {
PC_1= 1:   non-critical code
PC_1= 2:   flag[1] = true;
PC_1= 3:   while(flag[0]) {
PC_1= 4:     if(turn != 1) {
PC_1= 5:       flag[1] = false;
PC_1= 6:       while(turn != 1);
PC_1= 7:       flag[1] = true;
PC_1= 8:     }
PC_1= 9:   }
PC_1=10:   critical section
PC_1=11:   turn = 0;
PC_1=12:   flag[1] = false;
PC_1=13: }
```

See http://en.wikipedia.org/wiki/Dekker's_algorithm.

Dekker's algorithm guarantees mutual exclusion

- Assume initialization: $\texttt{PC}_0 = \texttt{PC}_1 = 0$; $\texttt{flag[0]} = \texttt{flag[1]} = \texttt{false}$; $\texttt{turn} = 0$.
- Invariant:

$$\begin{array}{rcl} I & = & \forall i \in \{0,1\}.\ \texttt{flag[}i\texttt{]} = (\texttt{PC}_i \in \{3,4,5,8,9,10,11\}) \\ & \wedge & \neg((\texttt{PC}_0 = 10) \wedge (\texttt{PC}_1 = 10)) \end{array}$$

  - Assertions about $\texttt{PC}_i$ refer to the state immediately before executing the statement at $\texttt{PC}_i$.
  - Individual program statements are executed atomically, i.e. without interference by actions of other threads.

Proof that I is an invariant (1/2)

$$\begin{array}{rcl} I & = & \forall i \in \{0,1\}.\ \texttt{flag[}i\texttt{]} \ = \ (\texttt{PC}_i \in \{3,4,5,8,9,10,11\}) \\ & \wedge & \neg((\texttt{PC}_0 = 10) \wedge (\texttt{PC}_1 = 10)) \end{array}$$

- I holds initially.
- Statements that don't modify `flag[i]`, $\texttt{PC}_i \notin \{2,5,7,12\}$.
  - Thus, they maintain the clause connecting the value of `flat[i]` to $\texttt{PC}_i$.
  - For example, if $\texttt{PC}_i = 0$,
  -
  - Then I implies that $\texttt{flag[}i\texttt{]} = \texttt{false}$.
  - Executing `while(true) {` sets $\texttt{PC}_i \leftarrow 1$ and leaves $\texttt{flag[}i\texttt{]} = \texttt{false}$.
  - Thus, I continues to hold.
- Similar reasoning applies for statements that do modify `flag[i]`, $\texttt{PC}_i \in \{2,5,7,12\}$.

Proof that I is an invariant (2/2)

$$\begin{array}{rcl} I & = & \forall i \in \{0,1\}.\ \texttt{flag[}i\texttt{]} \ = \ (\texttt{PC}_i \in \{3,4,5,8,9,10,11\}) \\ & \wedge & \neg((\texttt{PC}_0 = 10) \wedge (\texttt{PC}_1 = 10)) \end{array}$$

- Now consider $\neg((\texttt{PC}_0 = 10) \wedge (\texttt{PC}_1 = 10))$.
  - If $\texttt{PC}_0 \leftarrow 10$,
    - Rhen $\texttt{PC}_0$ was 3 which means that $\neg\texttt{flag[1]}$.
    - Because I held before performing the $\texttt{PC}_0 = \textbf{3}\texttt{: while(flag[1]) \{}$ statement, $\texttt{PC}_1 \neq 10$.
    - Thus, $\texttt{PC}_1 \neq 10$ after performing the statement at $\texttt{PC}_0 = 3$, and $\neg((\texttt{PC}_0 = 10) \wedge (\texttt{PC}_1 = 10))$ continues to hold.
  - Similar reasoning applies when $\texttt{PC}_1 \leftarrow 10$.
- Thus, I is maintained by all actions of both threads.
- I is an invariant.
- Also, I guarantees mutual exclusion because I implies $\neg((\texttt{PC}_0 = 10) \wedge (\texttt{PC}_1 = 10))$.

Dekker's algorithm guarantees progress

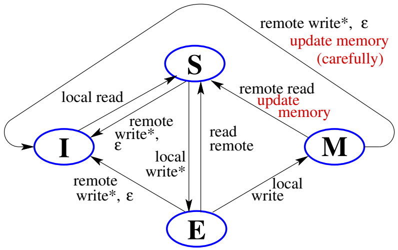
- Assume that every statement eventually terminates.
  - This requires that every time a thread enters its critical section, it eventually leaves.
  - We don't require that the non-critical code terminate.
- We can show a seqence of "eventually" properties that shows that any time a thread tries to enter its critical section it eventually does so. I.e.

$$\mathtt{PC}_i = 2 \leadsto \mathtt{PC}_i = 10$$

- I'll spare you the proof.
  - In class, I was asked about why the algorithm includes the `turn` variable.
  - `turn` is needed to ensure that both threads can make progress.
  - See slide 29 for a "simplified" version without `turn` and why it fails.

Dekker's algorithm with caches

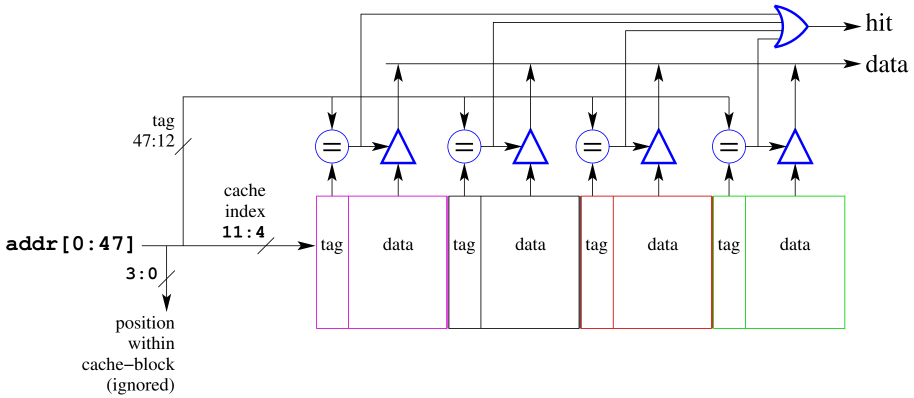
The MESI protocol



I	= invalid
S	= shared
E	= exclusive
M	= modified
write*	= write–through (to memory)
write	= write–back (local–cache only)
ε	= "spontaneous" transition

- Caches can share read-only copies of a cache block.
- When a processor writes a cache block, the first write goes to main memory.
  - The other caches see the write and invalidate their copies.
  - This ensures that writeable blocks are exclusive.

A typical cache



- Only the read-path is shown. Writing is similar.
- This is a 16K-byte, 4-way set-associative cache, with 16 byte cache blocks.

Snooping caches

- Each cache has two copies of the tags.
  - One copy is used for operations by the local processor.
  - The other copy is used to monitor operations on the main memory bus.
    - if another processor attempts to read a block which we have in the `exclusive` or `modified` state, we provide the data (and update main memory).
    - if another processor attempts to write a block that we have, we invalidate our block (updating main memory first if our copy was in the `modified` state.
- Pros and cons:
  - Fairly easy to implement.
  - Doesn't scale to large numbers of processors.

Directory schemes

- Main memory keeps a copy of the data and
  - a bit-vector that records which processors have copies, and
  - a bit to indicate that one processor has a copy and it may be modified.
- A processor accesses main memory as required by the MESI protocol.
  - The memory unit sends messages to the other CPUs to direct them to take actions as needed by the protocol.
  - The ordering of these messages ensures that memory stays consistent.

Sequential Consistency

Memory is said to be sequentially consistent if

- All memory reads and writes from all processors can be arranged into a single, sequential order, such that:
  - The operations for each processor occur in the global ordering in the same order as they did on the processor.
  - Every read gets the value of the preceeding write to the same address.
- Sequential consistency corresponds to what programmers thing "ought" to happen.

MESI Guarantees Sequential Consistency

- First we prove that at most one processor can have the cache block for any particular address in the E or M state.
- Define:

$$\begin{array}{rll} \multicolumn{3}{l}{\mathit{value}(\mathit{addr})} \\ = & \mathit{cache}_i(\mathit{addr}).\mathit{data}, & \text{if } \exists i.\ \mathit{cache}_i(\mathit{addr}).\mathit{state} \in \{\mathtt{E}, \mathtt{M}\} \\ = & \mathit{MEM}(\mathit{addr}), & \text{otherwise} \end{array}$$

- We can show that every *read*(*addr*) gets the value *value*(*addr*), and that
- We *value*(*addr*) gives the value from the most recent write to *addr*.

Dekker's with C-threads

```
typedef struct { % thread parameters
   int id, ntrials;
} dekker_args;

% shared variables
int flag[] = {0,0};
int count[] = {0,0};
int turn = 0;

int dekker_thread(void *void_arg) {
   ...
   for(int i = 0; i < ntrials; i++) {
      do some work;
      acquire the lock;
      critical section (includes test for inteference);
      release lock;
   }
}
```

Work, then lock

```
% do a random amount of "work" before critical region
r = 23*r & 0x3f; % simple pseudo-random, range = {0 ... 63}
for(int j = 0; j < r; j++); % this is "work"?
% acquire the lock
flag[me] = TRUE; % indicate intention to enter critical region
while(flag[!me]) {
   if(turn != me) {
      flag[me] = FALSE;              % give the other thread a chance
      while(turn != me);             % spin waiting for turn
      flag[me] = TRUE;               % try again
   }
}
```

Critical section, then unlock

```
% critical section
for(int j = 0; j < 10; j++) {
   count[me] = j;
   % check_zero reports error and dies if count[!me] != 0
   check_zero(count, !me, i);
}
count[me] = 0;
% release the lock
turn = !me;
flag[me] = 0;
```

Let’s try it

```
% gcc -std=c99 dekker0.c cz.o -o d0
% d0
check_zero failed for trial 8:  a[0] = 1
% d0
check_zero failed for trial 986:  a[1] = 4
% d0
check_zero failed for trial 898:  a[1] = 4
% d0
check_zero failed for trial 10:  a[0] = 1
% ...
```

- What happened?
- Why?

Weaker Consistency

The problem of write-buffers.

Fixing the bug

```
% acquire the lock
flag[me] = TRUE; % indicate intention to enter critical region
__asm__("mfence");
while(flag[!me]) {
   if(turn != me) {
      flag[me] = FALSE;      % give the other thread a chance
      while(turn != me);     % spin waiting for turn
      flag[me] = TRUE;       % try again
   __asm__("mfence");
   }
}
```

- Try again:

```
% d1
ok
% d1
ok
% d1
ok
% ...
```

What's mfence?

- A memory fence.
- Simple version:
  - All loads and stores issued by the processor that executes the `mfence` must complete globally before execution continues beyond the `mfence`.
- `mfence` instructions are expensive
- And in-line assembly code is painful
  - Not portable.
  - Hard to read.
  - Who wants to program in assembly?

Summary

- Shared-Memory Architectures
  - Use cache-coherence protocols to allow each processor to have its own cache while maintaining (almost) the appearance of having one shared memory for all processors.
  - A typical protocol: MESI
  - The protocol can be implemented by snooping or directories.
- Shared-Memory Programming
  - Need to avoid interference between threads.
    - Assertional reasoning (e.g. invariants) are crucial, much more so than in sequential programming.
    - There are too many possible interleavings to handle intuitively.
    - In practice, we don't formally prove complete programs, but we use the ideas of formal reasoning.
  - Real computers don't provide sequential consistency.
    - Use a thread library.

Announcements (1/2)

CS Distinguished Lecture Series:

- Speaker: Thomas Ball, Microsoft Research
- Title: Advances in Automated Theorem Proving.
- When: Thursday, October 11, 2012 at 3:30pm
- Where: Hugh Dempster Pavilion – Room 110
- Hosts: Mark Greenstreet and Alan Hu
- Abstract: In the last decade, advances in satisfiability-modulo-theories (SMT) solvers have powered a new generation of software tools for verification and testing. These tools transform various program analysis problems into the problem of satisfiability of formulas in propositional or first-order logic, where they are discharged by SMT solvers, such as Z3 from Microsoft Research (MSR).
- Vote: YES

Announcements (2/2)

- October 8 (Monday): Thanksgiving.
  - No TA office hours.
  - I might hold a late-afternoon/early-evening office hour. I'll post an announcement to Piazza.
- October 9 (Tuesday):
  - Instructor office hour: 11:30am – 1pm.
  - Lecture topic: Message passing machines.
  - Reading: Lin & Snyder, Chapter 2, through "Observations of Our Six Computers."
  - Homework 2 due at 11:59pm.
- October 11 (Thursday):
  - No Instructor office hours.
  - Lecture: DLS, room DMP 110 (see previous slide)
- October 18 (Thursday): Midterm (in class)

Supplementary Material

Dekker's algorithm without the `turn` variable:

thread 0:

```
PC0= 0: while(true) {
PC0= 1:   non-critical code
PC0= 2:   flag[0] = true;
PC0= 3:   while(flag[1]) {
PC0= 5:     flag[0] = false;
PC0= 7:     flag[0] = true;
PC0= 9:   }
PC0=10:   critical section
PC0=12:   flag[0] = false;
PC0=13: }
```

thread 1:

```
PC1= 0: while(true) {
PC1= 1:   non-critical code
PC1= 2:   flag[1] = true;
PC1= 3:   while(flag[0]) {
PC1= 5:     flag[1] = false;
PC1= 7:     flag[1] = true;
PC1= 9:   }
PC1=10:   critical section
PC1=12:   flag[1] = false;
PC1=13: }
```

I've left the `PC` numbers as in the original version (slide 7).

Analysis of the "no-turn" version

- The "no-turn" version guarantees mutual exclusion.
  - The invariant and proof on slides 8–10 applies for this version as well.
- The "no-turn" version does not guarantee progress.
  - See thec counter-example on the next slide.
  - By repeating lines 4–10 indefinitely
    - Thread 0 never enters its critical region.
    - Thread 1 enters its critical region an unbounded number of times.
- Thread 0 waits forever to enter its critical region.

Counter-example trace for the no-turn algorithm

step	from state				perform
	PC_0	PC_1	`flag[0]`	`flag[1]`	
0	0	0	`false`	`false`	$PC_0 =$ 0: `while(true) {`
1	1	0	`false`	`false`	$PC_0 =$ 1: *non-critical code*
2	2	0	`false`	`false`	$PC_0 =$ 2: `flag[0] = true;`
3	3	0	`true`	`false`	$PC_1 =$ 0: `while(true) {`
4	3	1	`true`	`false`	$PC_1 =$ 1: *non-critical code*
5	3	2	`true`	`false`	$PC_1 =$ 2: `flag[1] = true;`
6	3	3	`true`	`true`	$PC_0 =$ 3: `while(flag[1]) {`
7	5	3	`true`	`true`	$PC_0 =$ 5: `flag[0] = false;`
8	7	3	`false`	`true`	$PC_1 =$ 3: `while(flag[1]) {`
9	7	10	`false`	`true`	$PC_1 =$ 10: critical section
10	7	12	`false`	`true`	$PC_1 =$ 12: `flag[1] = false;`
11	7	0	`false`	`false`	$PC_0 =$ 7: `flag[0] = true;`
12	3	0	`false`	`false`	$PC_1 =$ 0: `while(true) {`
$\geq$13	repeat steps 4–12 indefinitely.				