Shared Memory Multiprocessors

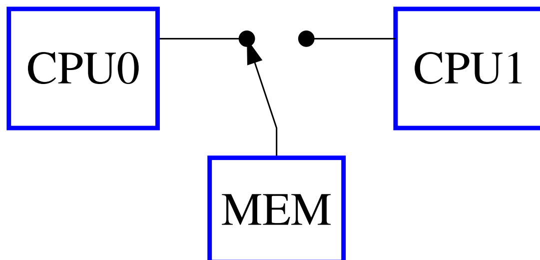
Mark Greenstreet

CpSc 448B – Oct. 4, 2011

Outline:

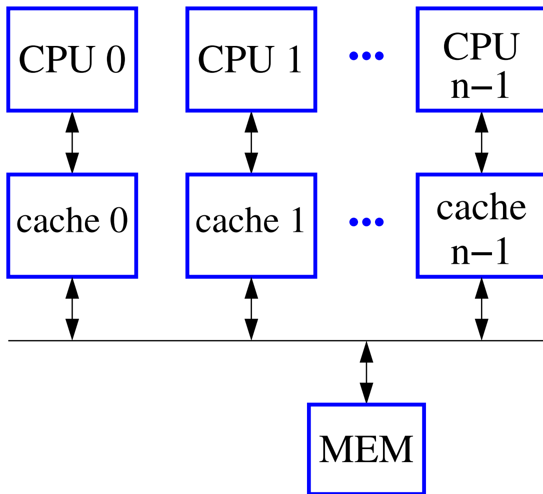
- Shared-Memory Architectures
- Memory Consistency
- Examples

An Ancient Shared-Memory Machine



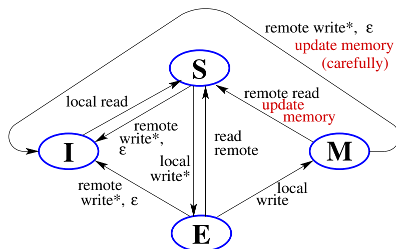
- Multiple CPU's (typically two) shared a memory
- If both attempted a memory read or write at the same time
  - One is chosen to go first.
  - Then the other does it's operation.
  - That's the role of the switch in the figure.
- By using multiple memory units (partitioned by address), and a switching network, the memory could keep up with the processors.
- But, now that processors are 100's of times faster than caches, 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 procesor does a write?

The MESI protocol



I = invalid
S = shared
E = exclusive
M = modified

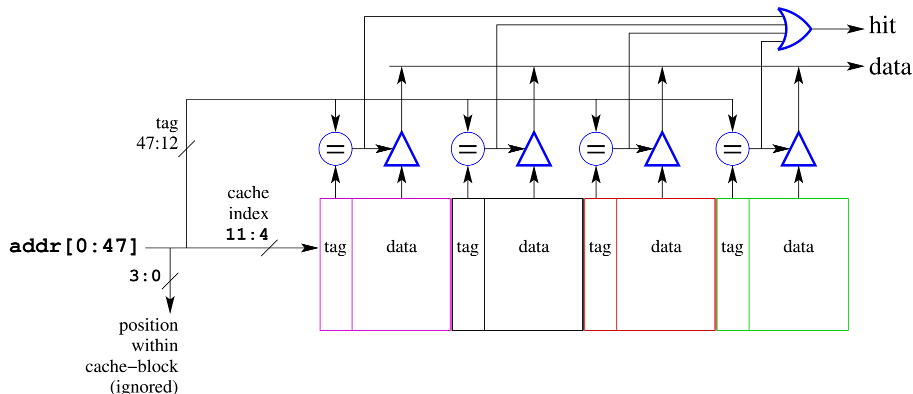
write* = write−through (to memory)

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 block 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 teh `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.

Shared-Memory Machines in practice

Sequential Consistency

MESI Guarantees Sequential Consistency

Dekker's Algorithm

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

thread 0:

```
flag[0] = true;
while(flag[1]) {
    if(turn != 0) {
        flag[0] = false;
        while(turn != 0);       X
        flag[0] = true;
    }
}
critical section
turn = 1;
flag[0] = false;
```

thread 1:

```
flag[1] = true;
while(flag[0]) {
    if(turn != 1) {
        flag[1] = false;
        while(turn != 1);
        flag[1] = true;
    }
}
critical section
turn = 0;
flag[1] = false;
```

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?