# CSE 490/590 Computer Architecture

## Snoopy Caches I

Steve Ko
Computer Sciences and Engineering
University at Buffalo



## Last time…

- Implementations for semaphores
  - Test&set
  - Compare&swap
  - Load-reserve & store-conditional
- Sequential consistency vs. weaker consistencies
  - Agreement between hardware and software
  - For weaker consistency models, hardware provides extra instructions for software to implement stronger guarantees, e.g., memory fences



## Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2.
Initially, both c1 and c2 are 0 *(not busy)*

*Process 1*

```
    …
    c1=1;
L:  if c2=1 then go to L
      < critical section>
    c1=0;
```

*Process 2*

```
    …
    c2=1;
L:  if c1=1 then go to L
      < critical section>
    c2=0;
```

What is wrong?



## Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

*Process 1*

```
    …
L:  c1=1;
    if c2=1 then
        { c1=0; go to L}
      < critical section>
    c1=0
```

*Process 2*

```
    …
L:  c2=1;
    if c1=1 then
        { c2=0; go to L}
      < critical section>
    c2=0
```

- Deadlock is not possible but with a low probability a *livelock* may occur.
- An unlucky process may never get to enter the critical section ⇒ *starvation*



## A Protocol for Mutual Exclusion

***T. Dekker, 1966***

A protocol based on 3 shared variables c1, c2 and turn.
Initially, both c1 and c2 are 0 *(not busy)*

*Process 1*

```
    …
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

*Process 2*

```
    …
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

- turn = *i* ensures that only process *i* can wait
- variables c1 and c2 ensure *mutual exclusion*

*Solution for n processes was given by Dijkstra and is quite tricky!*



## Analysis of Dekker's Algorithm

Scenario 1

*Process 1*

```
    …
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

*Process 2*

```
    …
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

Scenario 2

*Process 1*

```
    …
    c1=1;
    turn = 1;
L: if c2=1 & turn=1
                then go to L
      < critical section>
    c1=0;
```

*Process 2*

```
    …
    c2=1;
    turn = 2;
L: if c1=1 & turn=2
                then go to L
      < critical section>
    c2=0;
```

## N-process Mutual Exclusion

***Lamport's Bakery Algorithm***

*Process i* — Initially num[j] = 0, for all j

Entry Code

```
choosing[i] = 1;
num[i] = max(num[0], …, num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++)  {
    while( choosing[j] );
    while( num[j] &&
           ( ( num[j] < num[i] ) ||
             ( num[j] == num[i] &&  j < i ) ) );
}
```

Exit Code

```
num[i] = 0;
```

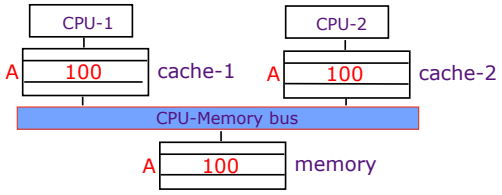


## CSE 490/590 Administrivia

- CSE Graduate Conference on Friday, 4/15 @ 145 Student Union
  - No class
- Keyboards available for pickup at my office
- Updated project 2 with more clarifications & grading criteria



## Memory Coherence in SMPs

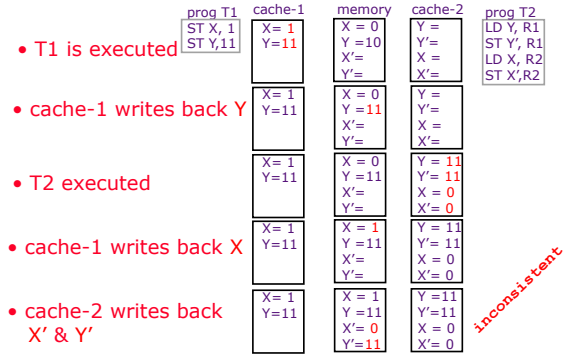



Suppose CPU-1 updates A to 200.
*write-back:* memory and cache-2 have stale values
*write-through:* cache-2 has a stale value

*Do these stale values matter?*
*What is the view of shared memory for programming?*

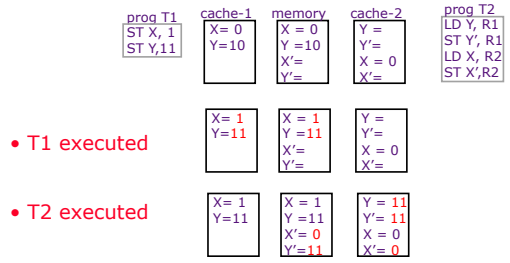


## Write-back Caches & SC






## Write-through Caches & SC




*Write-through caches don't preserve sequential consistency either*



## Cache Coherence vs. Memory Consistency

- A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors
  - i.e., updates are not lost
- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another
  - Equivalently, what values can be seen by a load
- A cache coherence protocol is not enough to ensure sequential consistency
  - But if sequentially consistent, then caches must be coherent
- Combination of cache coherence protocol plus processor memory reorder buffer implements a given machine's memory consistency model

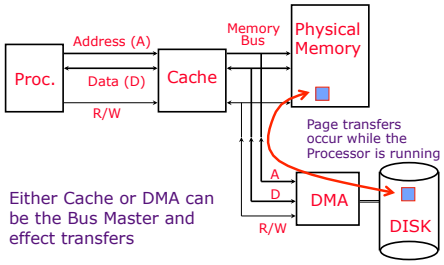
## Maintaining Cache Coherence

Hardware support is required such that
- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

⇒ *cache coherence protocols*



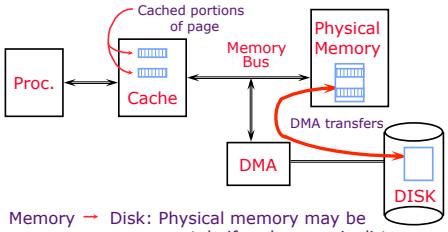
## Warmup: Parallel I/O




(DMA stands for Direct Memory Access, means the I/O device can read/write memory autonomous from the CPU)



## Problems with Parallel I/O




Memory → Disk: Physical memory may be stale if cache copy is dirty

Disk → Memory: Cache may hold stale data and not see memory writes



## Acknowledgements


- These slides heavily contain material developed and copyright by
  - Krste Asanovic (MIT/UCB)
  - David Patterson (UCB)
- And also by:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252