CSE 490/590 Computer Architecture

Synchronization and Consistency II

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Last time…

- Consistency problem
  - Producer-consumer
- Sequential consistency
- Semaphores
- Instructions that can implement semaphores
  - Test&set
  - Fetch&add
  - Swap

Locks or Semaphores

E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

P(s): *if s>0, decrement s by 1, otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P's and V's must be executed atomically, i.e., without
- *interruptions* or
- *interleaved accesses to s* by other processors

```
Process i
    P(s)
        <critical section>
    V(s)
```

initial value of s determines the maximum no. of processes in the critical section

Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design…

Simpler solution:
atomic read-modify-write instructions

Examples: *m is a memory location, R is a register*

```
Test&Set (m), R:
    R ← M[m];
    if R==0 then
        M[m] ← 1;
```

```
Fetch&Add (m), Rv, R:
    R ← M[m];
    M[m] ← R + Rv;
```

```
Swap (m), R:
    Rt ← M[m];
    M[m] ← R;
    R ← Rt;
```


Multiple Consumers Example

using the Test&Set Instruction

```
P:      Test&Set (mutex),Rtemp
        if (Rtemp!=0) goto P
        Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rhead=Rhead+1
        Store (head), Rhead
V:      Store (mutex),0
        process(R)
```

Critical Section

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P's and V's

What if the process stops or is swapped out while in the critical section?

Nonblocking Synchronization

```
Compare&Swap(m), Rt, Rs:
  if (Rt==M[m])
     then  M[m]=Rs;
      Rs=Rt ;
      status ← success;
     else   status ← fail;
```

status is an *implicit argument*

```
try:    Load Rhead, (head)
spin:   Load Rtail, (tail)
        if Rhead==Rtail goto spin
        Load R, (Rhead)
        Rnewhead = Rhead+1
        Compare&Swap(head), Rhead, Rnewhead
        if (status==fail) goto try
        process(R)
```

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (m):
    <flag, adr> ← <1, m>;
    R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs' reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:   Load-reserve R_head, (head)
spin:  Load R_tail, (tail)
       if R_head==R_tail goto spin
       Load R, (R_head)
       R_head = R_head + 1
       Store-conditional (head), R_head
       if (status==fail) goto try
       process(R)
```

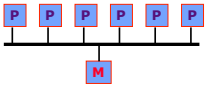

Performance of Locks

Blocking atomic read-modify-write instructions
e.g., Test&Set, Fetch&Add, Swap
vs
Non-blocking atomic read-modify-write instructions
e.g., Compare&Swap,
Load-reserve/Store-conditional
vs
Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:
degree of contention,
caches,
out-of-order execution of Loads and Stores

later ...

Issues in Implementing Sequential Consistency



Implementation of SC is complicated by two issues

- *Out-of-order execution capability*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a ≠ b
Store(a); Load(b)	*yes if* a ≠ b
Store(a); Store(b)	*yes if* a ≠ b

- *Caches*
  Caches can prevent the effect of a store from being seen by other processors

CSE 490/590 Administrivia

- No class on Friday, 4/15
- Keyboards available for pickup at my office
  - Today after 2pm

Memory Fences

Instructions to sequentialize memory accesses

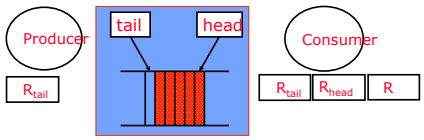
Processors with *relaxed or weak memory models* (i.e., permit Loads and Stores to different addresses to be reordered) need to provide *memory fence* instructions to force the serialization of memory accesses

Examples of processors with relaxed memory models:
Sparc V8 (TSO,PSO): Membar
Sparc V9 (RMO):
Membar #LoadLoad, Membar #LoadStore
Membar #StoreLoad, Membar #StoreStore

PowerPC (WO): Sync, EIEIO

Memory fences are expensive operations, however, one pays the cost of serialization only when it is required

Using Memory Fences



Producer posting Item x:
Load R_{tail}, (tail)
Store (R_{tail}), x
$Membar_{SS}$
$R_{tail}=R_{tail}+1$
Store (tail), R_{tail}

ensures that tail ptr is not updated before x has been stored

Consumer:
Load R_{head}, (head)
spin: Load R_{tail}, (tail)
if $R_{head}==R_{tail}$ goto spin
$Membar_{LL}$
Load R, (R_{head})
$R_{head}=R_{head}+1$
Store (head), R_{head}
process(R)

ensures that R is not loaded before x has been stored

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*

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