Chapter 5 (Cont)

Thread-Level Parallelism

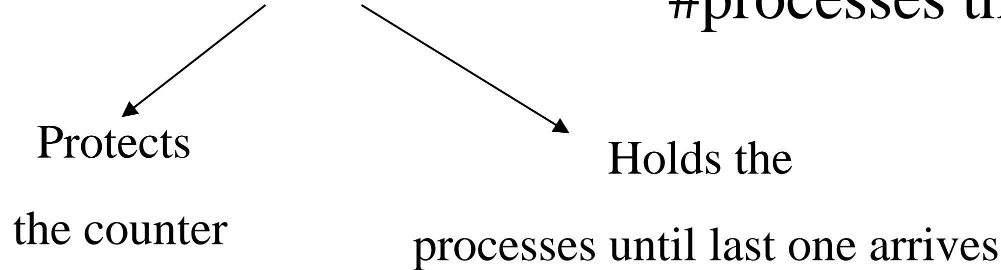
Instructor: Josep Torrellas

CS433

Copyright Josep Torrellas 1999, 2001, 2002, 2013

Barrier Synchronization

- Barrier forces all processes to wait until all processes reach the barrier . Then all processes are released
- Implemented w/2 spin locks + counter that counts #processes that have arrived



Algorithm

```
lock(counterlock)
    if(count == 0) release = 0;  /* First process : reset release */
    count ++ ; /* how many processes have arrived */
unlock(counterlock)
if(count == total) { /* all have arrived */
    count = 0; release = 1; /* release processes */
}
else {  /* more processes to come ; wait */
     spin(release)  /* spin until release = 1  */
}
```

Problem

Problem : Suppose that before the last process leaves the barrier , the first process arrives (case barrier is in a loop) and sets release = 0

$\Rightarrow$ last process never leaves

$\Rightarrow$ Count never reaches total

$\Rightarrow$ barrier spins forever

Solution : Use the sense reversing barrier : each process has a variable (local_sense) , initialized to 1

Final

```
local_sense = ! local_sense   /* toggle local_sense */
lock(counterlock);
    count++ ;
unlock(counterlock);
if(count == total) {
    count = 0; release = local_sense ;
}
else {  /* spin until release is local_sense */
    spin(release = local_sense);
}
/* many bus accesses when processes reach a barrier   */
```

Synchronization for Large Scale Machines

1) Software :

→ delay processes when they fail to acquire a lock. This reduces traffic → reduces network contention

best performance : if delay is increased exponentially

" exponential backoff "

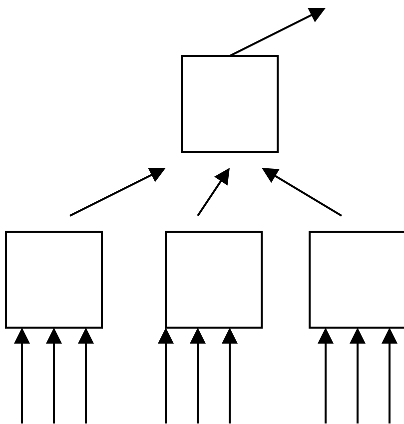
```
          li      R3,1         ; initial delay
lock it : ll      R2,0(R1)
          bnez    R2,lockit    ;retry
          addi    R2,R2,1
          sc      R2,0(R1)
          bnez    R2,gotit
          sll     R3,R3,1      ;increase delay by 2
          pause   R3
          j       lockit
gotit :
```

→ use combining trees for barriers : (reduce contention)

have a tree w/a fan in of k=3 per node

when the kth process arrives at a tree →signal the next level in the tree

→When a process arrives at the root of the tree , all processes are released



```
Struct node { /* a node in the combining tree */
  int counterclock; /*lock for this node */
  int count; /* counter for this node */
  int parent; /* parent in the tree = 0.. P-1 except for root = -1 */
};
struct node tree [0… p-1]; /* the tree of nodes */
int local_sense; /* private per processor */
int release ; /* global release flag */
```

```
/* function to implement barrier */
barrier(int mynode) {
    lock(tree[mynode].counterclock); /* protect count */
    count ++;  /* increment count */
    unlock(tree[mynode].counterclock); /* unlock  */
    if(tree[mynode].count == k) { /* all symbols arrive at mynode */
            if(tree[mynode].parent  >= 0) {
                        barrier(tree[mynode].parent);
            } else {
                                    release = local_sense;
            }
            tree[mynode].count = 0; /*reset for next time */
     }else {
            spin(release = local_sense);  /*wait */
            }
    }

/* code executed by a processor to join barrier */
    local_sense = !local_sense;
barrier(mynode);
```

Combining Trees for Barriers

- Each node combines k processes
  has a separate counter and lock

- when a process reaches the root , it triggers the unfolding of the recursive calls

→ fetch-and -increment for barriers:

each processor sends request, gets a value

last one releases the lock

no need for one of the locks

```
local_sense = !local_sense;
fetch&increment (count);  /*atomic */
if(count == total) {   /* all arrived */
        count = 0;
        release = local_sense;
}
else spin(release=local_sense);
```

Models of Memory Consistency

Problem: a processor updates a variable

different processors see the update at different times

can this result in a program behaving unexpectedly?

P1	P2	Possible scenarios
both processors cache A=B = 0		do1 , not do2
A = 1	B = 1	do2 , not do1
if(B == 0) {	if(A == 0) {	not do1 , not do2
do1	do2	.
}	}	do2 , do1 ??

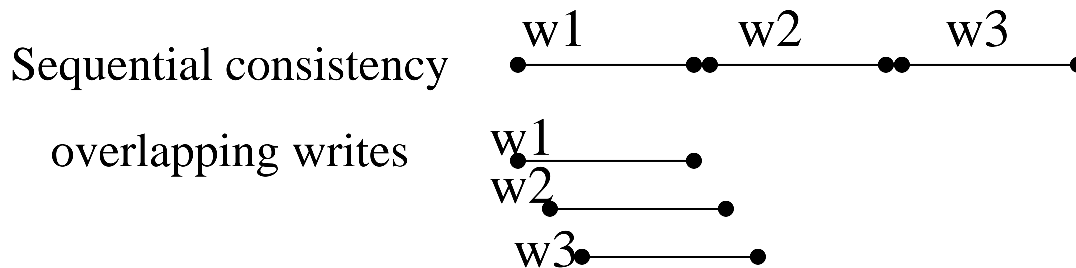
→ if the invalidations get delayed : if it is possible do1 & do2; not what the programmer expected

Model: Sequential Consistency

The result of any execution should be the same as if the accesses executed by each processor were kept in order and the accesses among different processors were interleaved

How to enforce it ? (Sufficient): processor does not issue a memory access until its previous one is finished

Problem: cannot place a write in a write buffer and continue with a read → low performance



The Programmers View

Want: a programming model that is simple to explain and allows for high performance

Programming model : assume that programs are synchronized

Synchronized programs : all accesses are ordered by synch operations

2 accesses to the same var by two different processes and one is a write, are always separated by a pair of synch ops

→ cases where this is not true : a data race ⇒ outcome depends on the speed of each processor

lock	lock	←acquire
rd x	rd x	
wr x	wr x	
unlock	unlock	← release

if it was not synchronized , both rds could read the original value

it is accepted → most pgms are synchronized

We will enforce stalls only at synch points

- Most programs are synchronized : else behavior of pgm is difficult to determine → depends on the speed of each process
- Some pgms are unsynchronized : avoid synchronization cost are willing to accept an inconsistent view of memory e.g. simulated annealing ; still converges even if a read returns an old value
- Restrictions on the ordering of mem operations : fences
  - memory fence (both rd and write)
- Memory fence by proc P :

  •All accesses by P executed before the fence must be completed before the fence

  •No accesses by P are initiated before fence is finished
- In weaker consistency models → synchronization accesses act as fences

Relaxed Models for Memory Consistency

- Allow higher performance
- Still preserve a simple programming model for synchronized programs
- Consider pairs of accesses issued by a single processor

  R $\rightarrow$ R

  R $\rightarrow$ W

  W $\rightarrow$ R

  W $\rightarrow$ W

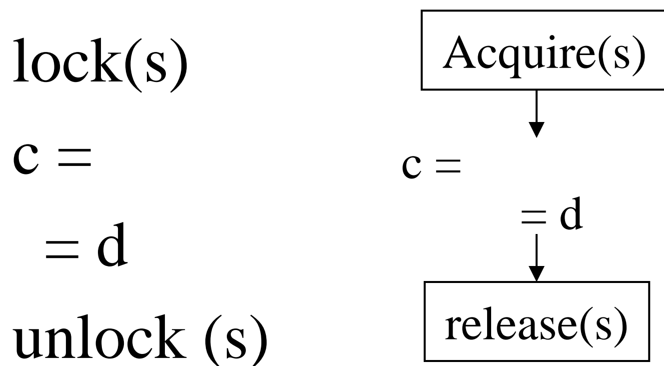
- If pair addresses the same location $\rightarrow$ ordering always preserved in all models
- In sequential consistency : all pairs are preserved

* Relaxing the order : the second access can complete before the first one

e.g.

W $\rightarrow$R : issue and complete the read before the processor receives all the invalidation acknowledgements for the write

<u>release consistency :</u> an example of relaxed consistency model



Release Consistency : (sufficient)

- Accesses within a critical section :
  cannot be issued until acquire is completed
- Release access :
  cannot be issued until accesses in critical section are completed

<u>Note :</u> only need to stall at synchronization points

- before executing any instruction in critical section stall until the acquire is completed
- before executing a release instruction , stall until all accesses in critical section are completed