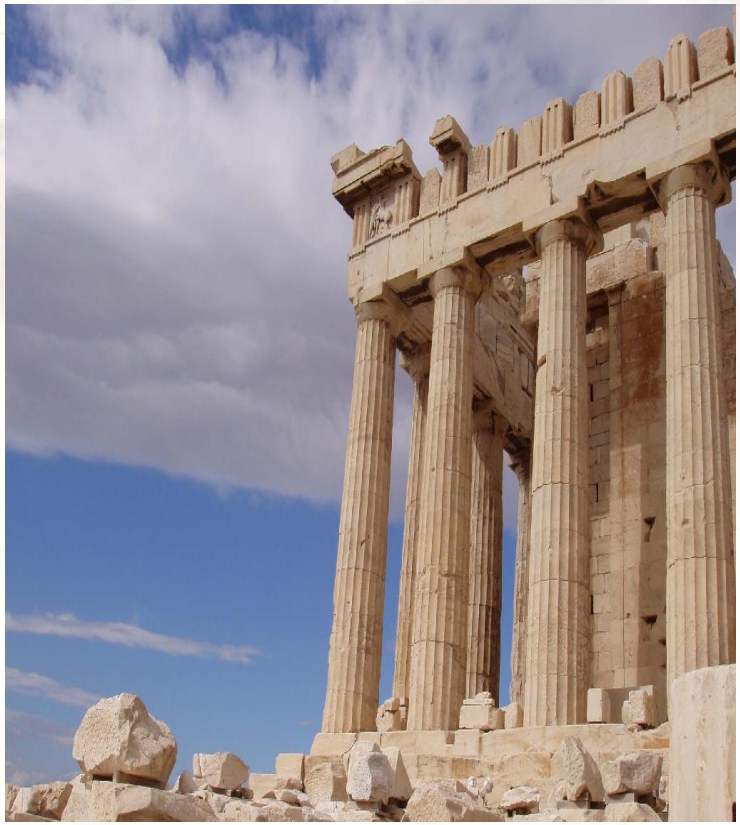
Ch5-3

Synchronization in Multiprocessor



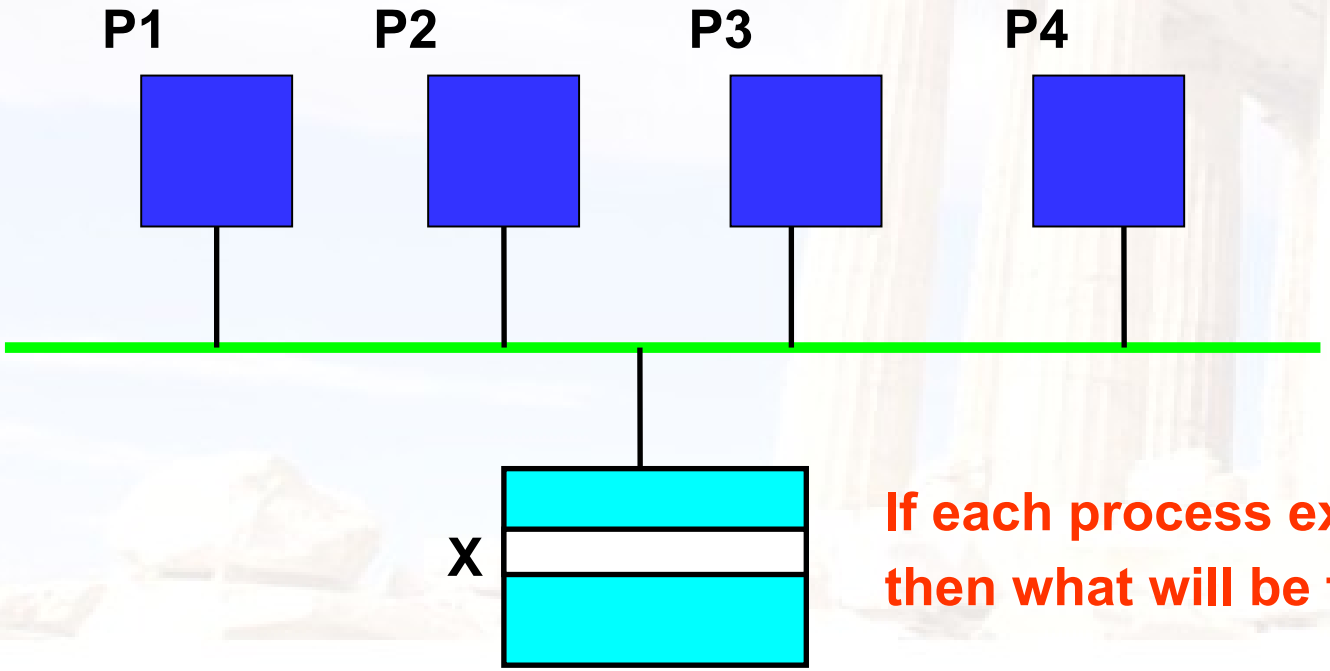
浙江大學
ZHEJIANG UNIVERSITY

Synchronization

- Definition: the coordination of simultaneous threads or processes to complete a task in order to get correct runtime order and avoid unexpected race conditions
  - Forks and Joins: in parallel programming a parallel process may want to wait until several events have occurred.
  - Producer-Consumer: A consumer process must wait until the producer process has produced data.
  - Exclusive use of a resource: Operating system has to ensure that only one process uses a resource at a given time.

Example:

- Why Synchronize? Need to know when it is safe for different processes to use shared data



If each process executes x:=x+1, then what will be final value of x?

X could be 1, 2, 3, 4

P0	P1	P2	P3
R ←x	R ←x	R← x	R ←x
R ← R+1	R ←R+1	R ←R+1	R ←R+1
x ←R	x ←R	x ←R	x ←R

How to implement Synchronization?

- Hardware Primitive: Provide **uninterruptable** ins truction to fetch and update memory (atomic oper ation); or **instruction sequences** that can retriev e and change a value atomically.
- User level synchronization operation using this p rimitive;
- For large scale MPs, synchronization can be a b ottleneck; need techniques to reduce contention and latency of synchronization

Hardware Primitives → Synchronize routine →Contentions → Better Primitives → performance optimization

Try without a Hardware Primitive

{free is a shared boolean, initially true }

Processor 0	Processor 1
while not free **do** nothing;	**while** not free **do** nothing;
free := FALSE;	free := FALSE;
{CS codes};	{CS codes};
free := TRUE	free := TRUE

P0, P1 both enter the Critical Section !

Try again without Hardware Primitive

{**turn0** and **turn1** are shared boolean, initially false}

P0	P1
turn 0 := TRUE;	turn 1 := TRUE
while turn1 **do** nothing;	**while** turn0 **do** nothing;
{CS codes}	{CS codes}
turn0 := FALSE;	turn1 := FALSE

Doesn't work with Deadlock !

Simple solution by Gary Peterson(1979)

Process 0	Process 1
try(0),try(1) : shared variables, initially false.	
{Lock}	{Lock}
try(0):= true ;	try(1) := true;
turn := 0;	turn := 1;
while (turn=0 && try(1))	**while** (turn=1 && try(0))
do nothing;	**do** nothing;
critical section;	*critical section;*
{Unlock}	{Unlock}
try(0) := false	try(1):= false

What if there are many processors ?

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory location
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
- Test-and-set: tests a value and sets it if the value passes the test
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
- Fetch-and-increment: it returns the value of a memory lo cation and atomically increments it

Implement Lock using Hardware Primitives

- **Using Atomic Exchange**

```
            DADDUI   R2, R0, #1
lockit:     EXCH     R2, 0(R1)
            BNEZ     R2,  lockit
```

- **Using Test-and-Set**

```
lockit:     T&S      R2, 0(R1)
            BNEZ     R2,  lockit
```

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction:
  - use 2 instead
- Load linked (load locked) + store conditional
  - Load linked returns the initial value
  - Store conditional returns 1 if it succeeds (no other store to same memory location since proceeding load) and 0 otherwise
  - RISC-V: load reserved/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails
- Advantage: separate read and write of shared data

Build atomic operations using LL&SC

- Example doing atomic swap with LL & SC:

```
try:  mov x3,x4       ;mov exchange value
      lr x2, x1       ;load reserved from
      sc x3,0(x1)     ;store conditional
      bnez x3,try     ;branch store fails
      mov x4,x2       ;put load value in x4?
```

- Example doing fetch & increment with LL & SC:

```
try:  lr x2,x1        ;load reserved 0(x1)
      addi x3,x2,1          ;increment
      sc x3,0(x1)     ;store conditional
      bnez x3,try     ;branch store fails
```

How to implement LL&SC ?

- Link Register: keep track of the memory location address when execute LL.
- Clear when other processor write into the same location
- Match the SC address with the Link Register when SC. if not match or any exception, then failed, set return value=0; otherwise succeed.

- Note: Only Regitser-Register instruction can be insert between LL & SC, why ?

User Level Synchronization—Operation Using the Primitives

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock
-

```
            addi    x2, x0 ,  #1
lockit:     EXCH    x2,0(x1)    ;atomic exchange
            bnez    x2,lockit   ;already locked?
```

- What about MP with cache coherency?
  - Want to spin on cache copy to avoid full memory latency
  - Likely to get cache hits for such variables
- If with cache coherence, lock value can be in cache, otherwise is stored in memory.

Optimized Spin lock

- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic

- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:    li     x2,#1
lockit: lw     x3,0(x1)     ;load var
        bnez   x3,lockit    ;not free=>spin
        EXCH   x2,0(x1)     ;atomic exchange
        bnez   x2,try       ;already locked?
```

```
lockit:  ld x2,0(x1)
         bnez x2,lockit
         addi x2,x0,#1
         EXCH x2,0(x1)
         bnez x2,lockit
```

Spin lock using LL&SC

```
lockit:  LL      x2,0(x1)     ;load linked
         BNEZ    x2, lockit   ;not free=>spin
         Addui   x2, x0, #1     ;locked value
         SC      x2, 0(x1)    ;atomic exchange
         BEQZ    x2, lockit   ;already locked?
```

Cache-coherence steps and bus traffic for P0,P1,P2

	P0	P1	P2	Coherence state of lock	Bus/directory activity
1	Has lock	Spins, test if lock==0	Spins, test if lock==0	Shared	None
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock Variable from P0
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write back fromP0
4		(Waits while bus/directory busy)	Lock=0	Shared	Cache miss for P2 Satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap,returns 0 and set Lock=1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; Lock is exclusive.
7		Swap completes and returns 1,and set Lock =1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; send invalidate and generates write back from P2
8		Spins, test if Lock ==0			None

Advantage of this scheme: reduces memory traffic

Performance analysis of spin lock

- Assume:
  - that 10 processors content one spin lock at the same time.
  - Each bus miss takes 100 clock cycles.
  - No latency if hit in cache.
- Question: How many cycles need to get the lock once for all t he 10 processors ?
- Answer: each n-i+1 iteration
  - i processor read miss get lock = 0
  - i processor write miss try to lock it.
  - Write the lock to release
  - Sum (2i + 1) (i=N,..1) = N^2 + 2N, total 12000 clock cycles

Barrier Synchronization

- A barrier: forces all processes to wait until all the processes reach the barrier and then release all the processes to go forward.

- How to implement ?
  - Shared counter
  - Spin waiting for release signal

Code for barrier

```
lock ( counterlock );                //ensure update atomic
if ( count == 0 ) release = 0;       // first-->reset release
count = count +1;                    // count arrivals
unlock ( counterlock );              // release lock
if ( count == total ){               // all arrived
        count = 0;                   // reset counter
        release =1;                  // release processes
}
else {                               // more to come
        spin(release ==1) ;          // wait for arrivals
}
```

Is there anything wrong ?

Is there anythiny implicit error ?

Code for barrier (updated)

```
{
lock ( counterlock );                  //ensure update atomic
if ( count == 0 ) release = 0;          // first-->reset release
     count = count +1;                  // count arrivals
if ( count == total ){                  // all arrived
         count = 0;                      // reset counter
         release =1;                     // release processes
}
unlock ( counterlock );  // release lock
spin(release ==1) ;                     // wait for arrivals
}
```

Slower processes may be trapped in barrier by the fastest process.

Final result: all trapped some where.

Alternatives to improve

- Count again when leave the barrier
  - Set another shared counter to count down the existing process in barrier
  - Forbid any process reenter the barrier before every process leave the previous instance
  - NO! Extra long latency !
- Sense-reversing barrier
  - Private variable *Local_sense*

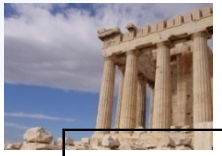
Sense-reversing barrier

```
{
Local_sense = ! Local_sense;  //toggle local_sence
lock ( counterlock );                //ensure update atomic
count = count +1;                   // count arrivals
if ( count == total ){               // all arrived
        count = 0;                  // reset counter
        release = local_sense;     // release processes
}
unlock ( counterlock );              // release lock
spin( release == local_sense) ;  // wait for signal
}
```

Slove the “trap” problem but still poor performance !

Performance analysis of barrier

- Assume:
  - that 10 processors each execute a barrier simultaneou sly.
  - Each bus miss takes 100 clock cycles.
  - No latency if hit in cache.
  - Processors are equal fast, No worry about code outside barrier
  - Lock is implemented using LL&SC
- Question: How many cycles need to arrive the ba rrier then leave?
-



Analysis

Event	Number of times for process *i*	Corresponding source line	Comment
LL counter lock	*i*	Lock(counterlock)	All processes try for lock
Store conditional	*i*	Lock(counterlock)	All processes try for lock
LD count	1	Count=count+1	Successful process
Load linked	*i* -1	Lock(counterlock)	Unsuccessful process;try again
SD count	1	Count=count+1	Miss to get exclusive access
SD counterlock	1	unlock(counterlock)	Miss to get the lock
LD release	2	Spin(release==local_sense)	Read release: misses initially and when finally written

- i^{th} processor will have $3i+4$ bus transactions
- The last processor arrives will have only one bus transaction on LD release

$$\Sigma\ (\ 3i+4\ ) - 1 = \frac{3n^2+11n}{2} - 1$$

20400 cycles

Performance Analysis

- Two main factors
  - Synchronization latency
    - Spin with LL&SC: 2 bus transaction of 1 read &1 write
    - Spin with EXCH: 1 bus transaction if lock is often free
  - Contention due to serialization
    - worse when scale up
- What can do for synchronization on Large scale MP ?
  - Decrease the latency when not much contention
  - Try to minimize the contentions when higher possibility of c omepetition

Software approach → Better hardware Primitives

Review the spin lock again

- Assume:
  - that 10 processors content one spin lock at the same time.
  - Each bus miss takes 100 clock cycles.
  - No latency if hit in cache.
- Question: How many cycles need to get the lock once or all the 10 processors ?
- Answer: each n-i+1 iteration
  - i processor read miss get lock = 0
  - i processor write miss try to lock it.
  - Write the lock to release
  - Sum (2i + 1) (i=N,..1) = N^2 + 2N, total 12000 clock cycles

Unnecessary writes !

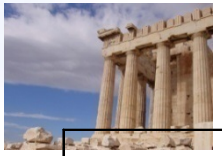
Software implementation exponential back-off

```
        DADDUI  R3,R0,# 1      ;R3=initial delay
lockit:    LL  R2, 0(R2)       ;load linked
        BNEZ R2, lockit        ;not available-spin
        DADDUI   R2, R2, #1    ;get locked value
        SC R2, 0(R1)       ;store conditional
        BNEZ R2, gotit         ;branch if store succeeds
            DSLL  R3, R3, #1        ;increase delay by 2
        PAUSE R3               ;delays by value in R3
        J    lockit
gotit:     use data protected by lock
```

Try to decrease the contentions when bad contention & obtain low latency when small contention

Queuing Lock

- Using array to keep track of the waiting proc esses
- One Process only try to get the lock when it’ s turn come up.
- Every processor using a different address for the lock
- Transfer the lock from one process to anothe r explicitly.



Review the barrier again

Event	Number of times for process *i*	Corresponding source line	Comment
LL counter lock	*i*	Lock(counterlock)	All processes try for lock
Store conditional	*i*	Lock(counterlock)	All processes try for lock
LD count	1	Count=count+1	Successful process
Load linked	*i* -1	Lock(counterlock)	Unsuccessful process;try again
SD count	1	Count=count+1	Miss to get exclusive access
SD counterlock	1	unlock(counterlock)	Miss to get the lock
LD release	2	Spin(release==local_sense)	Read release: misses initially and when finally written

- i^{th} processor will have $3i+4$ bus transactions
- The last processor arrives will have only one bus transaction on LD release

$$\sum (3i+4) - 1 = \frac{3n^2+11n}{2} - 1$$ 20400 cycles

Software Approach: Combining tree barrier

- Use predetermined n-ary tree structure
- When k processes arrive at a child node, it report to its parent node. The processes will be released while arrived at the root.
- Most Kth processors will content for lock at one node.
- Sense reversing technique is used.

Software Approach: Combining tree barrier

```
barrier (int mynode, int local_sense) {
        lock (tree[mynode].counterlock); /* protect count */
        tree[mynode].count=tree[mynode].count+1;
                /* increment count */
        if (tree[mynode].count==k) {/* all arrived at mynode */
                if (tree[mynode].parent >=0) {
                        barrier(tree[mynode].parent);
                } else{
                        release = local_sense;
                };
                tree[mynode].count = 0; /* reset for the next time */
        unlock (tree[mynode].counterlock); /* unlock */
        spin (release==local_sense); /* wait */
};
/* code executed by a processor to join barrier */
local_sense =! local_sense;
barrier (mynode);
```

Hardware Primitive

- Two primitive
  - Queuelock: for lock
  - Fetch&Increment: for barrier
- Difference with previous primitive
  - Same latency but much less serialization

Queuing lock

- Explicitly handing the lock from one waiting process or to the next
- Synchronization is integrated with memory controller or directory controller
- **How does it work ?**
  - Miss is sent to synchronization controller on first miss
  - If the lock is free: returned to the processor
  - Else: create a record of node request;
  - send back the locked value;

  While lock is available, pick one processor and update/invalidate the lock variable in the cache.

Performance of queuing lock

- Question: How long does it take to have 10 processors lock and unlock the varialbe using a Queuing Lock ?
- Answer:
  - n: miss for the first access
  - 1: free the lock by the first processor who get
  - 2: receive the lock, free the lock for each processor of (n-1) .
  - Total: n+1+2(n-1) = 3n-1, 2900 clock cycle

Key issues of implementing Que uing lock

- Distinguish the initial access of the lock
- Queue can be implemented by a bit vector like for t he shared set
- Capability of reclaim the queuing lock

Primitive: Fetch&Increment

- Decreasing serialization
  - For barrier: decrease serialization when collection.
  - For combining tree: decrease serialization on every node.

Barrier using Fetch&Increment

```
{
Local_sense = ! Local_sense;        //toggle local_sence
fetch_and_increment(count);          //atomic update
if ( count == total ){               // all arrived

        count = 0;                   // reset counter
        release = local_sense;       // release processes

 }
else {
spin(release == local_sense) ; // wait for signal
}
```

Performance for improved barrier

- Question: How long does it take to have 10 processors arrive and exit the barrier ?
- Answer:
  - n: fetch&increment of the count
  - n: read miss for the count
  - n: read miss for the release.
  - Total: 3n, 3000 clock cycle

Conclusion

- Efficient use of large scale multiprocessor is a big challenge.
  - Synchronization performance
  - Memory latency
  - Potential load imbalance

Fallacies and Pitfalls

- Measuring performance of multiprocessors by linear speedup versus execution time
- Amdahl's Law doesn't apply to parallel computers
- Linear speedups are needed to make multiprocessors cost-effective
  - Doesn't consider cost of other system components
- Not developing the software to take advantage of, or optimize for, a multiprocessor architecture