Implementing Critical Sections Using Busy Waiting

- In all our solutions today, a process enters a loop until the entry is granted ⇒ busy waiting.
- Problems with busy waiting:
  - Waste of CPU time
  - Potential for extra traffic/communication
  - If a process is switched out of CPU during critical section
    - other processes may have to waste a whole CPU quantum
    - may even deadlock with strictly prioritized scheduling (priority inversion problem)

Basic Hardware Mechanisms for Synchronization

- Test-and-set – atomic exchange
- Fetch-and-op (e.g., increment) – returns value and atomically performs op (e.g., increments it)
- Compare-and-swap – compares the contents of two locations and swaps if identical
- Load-locked/store conditional – pair of instructions – deduce atomicity if second instruction returns correct value
- Transaction support (e.g., Intel's TSX)

Simple Spinlock

type lock = (unlocked, locked)

procedure acquirelock (L : lock)

    while (testandset (L) == locked) // returns old value

procedure releaselock (L : lock)

    lock = unlocked

Synchronization Using Special Instruction: TSL (test-and-set)

```
entry_section:
   TSL R1, LOCK          | copy lock to R1 and set lock to 1
   CMP R1, #0            | was lock zero?
   JNE entry_section     | if it wasn't zero, lock was set, so loop
   RET                   | return; critical section entered

exit_section:
   MOV LOCK, #0          | store 0 into lock
   RET                   | return; out of critical section
```

CAS – the ABA Problem

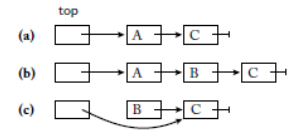


Figure 2.6: The ABA problem in a linked-list stack.

```
1: void push(node** top, node* new):
2:     node* old
3:     repeat
4:         old := *top
5:         new→next := old
6:     until CAS(top, old, new)
```

```
1: node* pop(node** top):
2:     node* old, new
3:     repeat
4:         old := *top
5:         if old = null return null
6:         new := old→next
7:     until CAS(top, old, new)
8:     return old
```

From Shared Memory Synchronization, ML Scott

98

Spinlocks using ll/sc

```
Lock:
    ll reg1, location //load-locked
    bnz reg1, Lock
    sc location, reg2 //store conditional
    beqz lock
    ret
Unlock
    st location, #0 // write 0
    ret
```

99

Using ll/sc for Atomic Exchange

- Swap the contents of R4 with the memory location specified by R1

```
try: mov R3, R4      ; mov exchange value
     ll   R2, 0(R1)  ; load linked
     sc R3, 0(R1)    ; store conditional
     beqz R3, try    ; branch if store fails
     mov  R4, R2     ; put load value in R4
```

100

Spinlock Algorithms

- Test&test&set (w, w/o exponential backoff)
- Ticket lock (w, w/o proportional backoff)
- Array based queue locks
- MCS linked-list based queue locks

101

MCS Lock Acquire

```
mcs_lock_acquire:
   st   %g0, [%o1+4]
   mov  %o1, %g3
   swap [%o0], %g3
   cmp  %g3, 0
   be   .LL4
   mov  1, %g2
   st   %g2, [%o1]
   st   %o1, [%g3+4]
.LL9:
   ld   [%o1], %g2
   cmp  %g2, 0
   bne  .LL9
   nop
.LL4:
   retl
   nop
```

102

MCS Lock Release

```
mcs_lock_release:
   ld   [%o1+4], %g2
   cmp  %g2, 0
   bne  .LL11
   nop
   cas  [%o0], %o1, %g2
   cmp  %g2, %o1
   be   .LL10
   nop
.LL17:
   ld   [%o1+4], %g2
   cmp  %g2, 0
   be   .LL17
   nop
.LL11:
   st   %g0, [%g2]
.LL10:
   retl
   nop
```

103

Which Spinlock Should I Use?

Table 4.1: Tradeoffs among fetch_and_Φ-based spin locks. Symbols are meant to suggest "good" (+), "fair" (∘), and "poor" (−). Space needs are in words, for n threads and j locks, none of which is requested or held.

	TAS (w/ backoff)	ticket	MCS (original)	CLH	MCS ("K42")	CLH
fairness	−	+	+	+	+	+
preemption tolerance	∘	−	−	−	−	−
scalability	∘	∘	+	+	+	+
fast-path overhead	+	+	∘	∘	−	−
interoperability	+	+	−	−	+	+
NRC-NUMA suitability	−	−	+	∘	+	∘
space needs	j	$2j$	j	$2j$	$2j$	$2n + 3j$

From Shared Memory Synchronization, ML Scott

104

A Simple Barrier

mycount: local variable; counter, flag, lock: shared variables
p = number of processors

```
lock(&lock);
if counter == 0
      flag = 0
mycount = ++counter
unlock(&lock);
if (mycount == p) {
      counter = 0
      flag = 1
}
else
      while (flag == 0) {};
```

Will this work?

105

A Sense-Reversing Barrier

```
local_sense=0: local variable; counter, flag, lock: shared variables
p = number of processors;

local_sense = !(local_sense);
lock(&lock)
counter++
if (counter == p) {
        unlock(&lock)
        counter = 0
        flag = local_sense
}
else {
        unlock (&lock)
        while (flag != local_sense) {};
}
```

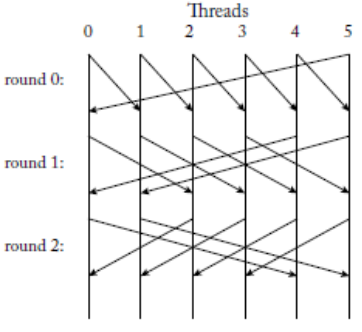
106

Barrier Algorithms

- Centralized sense-reversing barrier
- Software combining tree
- Tournament barrier
- Dissemination barrier
- Combining (static) tree with improved locality

107

Dissemination Barrier



⌜ Log_2 n ⌝rounds

From Shared Memory Synchronization, ML Scott

108

Which Barrier to Use?

Table 5.1: Tradeoffs among leading software barriers. Critical path lengths are in remote memory references (assuming broadcast on a CC-NUMA machine); they may not correspond precisely to wall-clock time. Space needs are in words. Constants a and d in the static tree barrier are arrival fan-in and departure fan-out, respectively. Fuzzy barriers are discussed in Section 5.3.1

	central	dissemination	static tree
space needs			
CC-NUMA	n+1	$n + 2n\lceil\log_2 n\rceil$	$4n + 1$
NRC-NUMA			$(5 + d)n$
critical path length			
CC-NUMA	$n + 1$	$\lceil\log_2 n\rceil$	$\lceil\log_a n\rceil + 1$
NRC-NUMA	∞		$\lceil\log_a n\rceil + \lceil\log_d n\rceil$
total remote refs			
CC-NUMA	$n + 1 .. 2n$	$n\lceil\log_2 n\rceil$	n
NRC-NUMA	∞		$2n - 2$
fuzzy barrier suitability	+	–	–
tolerance of changes in n	+	–	–

From Shared Memory Synchronization, ML Scott

109

Performance Goals

- Low latency, short critical path
- Low traffic
- Scalability
- Low storage cost
- Fairness

110

Non-blocking algorithms

Failure or suspension of any thread cannot cause failure or suspension of another thread (no indefinite delay due to mutual exclusion)

- Operations defined on it do not require mutual exclusion over multiple instructions (use atomic primitives)

- Obstruction-free algorithm
  - One that guarantees that a thread running in isolation will make progress (although livelock is possible)
- Lock-free algorithm
  - Operations guarantee that some process will complete its operation a finite amount of time, even if other processes halt
- Wait-free algorithm
  - Operations can guarantee that EVERY non-faulting process will complete its operation in a finite amount of time

111

Coherence

A multiprocessor memory system is coherent if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the result of the execution and

- it ensures that modifications made by a processor propagate to all copies of the data
- program order is preserved for each process in this hypothetical order
- writes to the same location by different processors are serialized and the value returned by each read is the value written by the last write in the hypothetical order

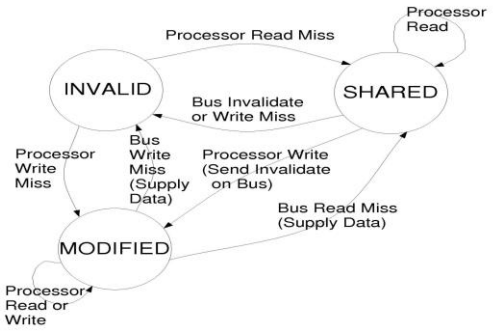
112

Snoop-Based Coherence

- Makes use of a shared broadcast medium to serialize events (all transactions visible to all controllers and in the same order)
  - Write update-based protocol
  - Write invalidate-based (e.g., basic MSI, MESI protocols)
- Cache controller uses a finite state machine (FSM) with a handful of stable states to track the status of each cache line
- Consists of a distributed algorithm represented by a collection of cooperating FSMs

113

A Simple Invalidate-Based Protocol - State Transition Diagram



114

Correctness Requirements

- Need to avoid
  - Deadlock – caused by a cycle of resource dependencies
  - Livelock – activity without forward progress
  - Starvation – extreme form of unfairness where one or more processes do not make forward progress while others do

115

Design Challenges

- Cache controller and tag design
- Non-atomic state transitions
- Serialization
- Cache hierarchies
- Split-transaction buses

116