Module 7: Synchronization

Lecture 14: Scalable Locks and Barriers

The Lecture Contains:

Ticket Lock

- Similar to Bakery algorithm but simpler
- A nice application of fetch & inc
- Basic idea is to come and hold a unique ticket and wait until your turn comes
    - Bakery algorithm failed to offer this uniqueness thereby increasing complexity

```
Shared: ticket = 0, release_count = 0;
Lock: fetch & inc reg1, ticket_addr
Wait: lw reg2, release_count_addr /* while ( release_count != ticket); */
sub reg3, reg2, reg1
bnez reg3, Wait

Unlock: addi reg2, reg2, 0x1 /* release_count ++ */
sw reg2, release_count_addr
```

- Initial fetch & inc generates O(P) traffic on bus-based machines (may be worse in DSM depending on implementation of fetch & inc)
- But the waiting algorithm still suffers from 0.5P 2 messages asymptotically
    - Researchers have proposed proportional backoff i.e. in the wait loop put a delay proportional to the difference between ticket value and last read release_count
- Latency and storage-wise better than Bakery
- Traffic-wise better than TTS and Bakery (I leave it to you to analyze the traffic of Bakery)
- Guaranteed fairness: the ticket value induces a FIFO queue

Previous Next

Array-based Lock

- Solves the $O(P^2)$ traffic problem
- The idea is to have a bit vector (essentially a character array if boolean type is not supported)
- Each processor comes and takes the next free index into the array via fetch & inc
- Then each processor loops on its index location until it becomes set
- On unlock a processor is responsible to set the next index location if someone is waiting
- Initial fetch & inc still needs O(P) traffic, but the wait loop now needs O(1) traffic
- Disadvantage: storage overhead is O(P)
- Performance concerns
  - Avoid false sharing: allocate each array location on a different cache line
  - Assume a cache line size of 128 bytes and a character array: allocate an array of size 128P bytes and use every 128 the position in the array
  - For distributed shared memory the location a processor loops on may not be in its local memory: on acquire it must take a remote miss; allocate P pages and let each processor loop on one bit in a page? Too much wastage; better solution: MCS lock (Mellor- Crummey & Scott)
- Correctness concerns
  - Make sure to handle corner cases such as determining if someone is waiting on the next location (this must be an atomic operation) while unlocking
  - Remember to reset your index location to zero while unlocking

Previous Next

RISC Processors

- All these atomic instructions deviate from the RISC line
  - Instruction needs a load as well as a store
- Also, it would be great if we can offer a few simple instructions with which we can build most of the atomic primitives
  - Note that it is impossible to build atomic fetch & inc with xchg instruction
- MIPS, Alpha and IBM processors support a pair of instructions: LL and SC
  - Load linked and store conditional

LL/SC

- Load linked behaves just like a normal load with some extra tricks
  - Puts the loaded value in destination register as usual
  - Sets a load_linked bit residing in cache controller to 1
  - Puts the address in a special lock_address register residing in the cache controller
- Store conditional is a special store
  - sc reg , addr stores value in reg to addr only if load_linked bit is set; also it copies the value in load_linked bit to reg and resets load_linked bit
- Any intervening “operation” (e.g., bus transaction or cache replacement) to the cache line containing the address in lock_address register clears the load_linked bit so that subsequent sc fails

Previous Next

Locks With LL/SC

- Test & set

```
Lock:   LL r1, lock_addr        /* Normal read miss/ BusRead */
        addi r2, r0, 0x1
        SC r2, lock_addr        /* Possibly upgrade miss */
        beqz r2, Lock           /* Check if SC succeeded */
        bnez r1, Lock           /* Check if someone is in CS */
```

- LL/SC is best-suited for test & test & set locks

```
Lock:   LL r1, lock_addr
        bnez r1, Lock
        addi r1, r0, 0x1
        SC r1, lock_addr
        beqz r1, Lock
```

Fetch & op with LL/SC

- Fetch & inc

```
Try:    LL r1, addr
        addi r1, r1, 0x1
        SC r1, addr
        beqz r1, Try
```

- Compare & swap: Compare with r1, swap r2 and memory location (here we keep on trying until comparison passes)

```
Try:    LL r3, addr
        sub r4, r3, r1
        bnez r4, Try
        add r4, r2, r0
        SC r4, addr
        beqz r4, Try
        add r2, r3, r0
```

Previous Next

Point-to-point Synch.

- Normally done in software with flags

  P0: A = 1; flag = 1;
  P1: while (!flag); print A;

- Some old machines supported full/empty bits in memory
  - Each memory location is augmented with a full/empty bit
  - Producer writes the location only if bit is reset
  - Consumer reads location if bit is set and resets it
  - Lot less flexible: one producer-one consumer sharing only (one producer-many consumers is very popular); all accesses to a memory location become synchronized (unless compiler flags some accesses as special)
- Possible optimization for shared memory
  - Allocate flag and data structures (if small) guarded by flag in same cache line e.g., flag and A in above example

Barrier

- High-level classification of barriers
  - Hardware and software barriers
- Will focus on two types of software barriers
  - Centralized barrier: every processor polls a single count
  - Distributed tree barrier: shows much better scalability
- Performance goals of a barrier implementation
  - Low latency: After all processors have arrived at the barrier, they should be able to leave quickly
  - Low traffic: Minimize bus transaction and contention
  - Scalability: Latency and traffic should scale slowly with the number of processors
  - Low storage: Barrier state should not be big
  - Fairness: Preserve some strict order of barrier exit (could be FIFO according to arrival order); a particular processor should not always be the last one to exit

Previous Next

Centralized Barrier

```
struct bar_type {
int counter;
struct lock_type lock;
int flag = 0;
} bar_name ;
BARINIT ( bar_name ) {
LOCKINIT( bar_name.lock );
bar_name.counter = 0;
}
```

```
BARRIER ( bar_name , P) {
int my_count ;
LOCK ( bar_name.lock );
if (! bar_name.counter ) {
bar_name.flag = 0; /* first one */
}
my_count = ++ bar_name.counter ;
UNLOCK ( bar_name.lock );
if ( my_count == P) {
bar_name.counter = 0;
bar_name.flag = 1; /* last one */
}
else {
while (! bar_name.flag );
}
}
```

Sense Reversal

- The last implementation fails to work for two consecutive barrier invocations
    - Need to prevent a process from entering a barrier instance until all have left the previous instance
    - Reverse the sense of a barrier i.e. every other barrier will have the same sense: basically attach parity or sense to a barrier

```
BARRIER ( bar_name , P) {
local sense = ! local_sense ; /* this is private per processor */
LOCK ( bar_name.lock );
bar_name.counter ++;
if ( bar_name.counter == P) {
UNLOCK ( bar_name.lock );
bar_name.counter = 0;
bar_name.flag = local_sense ;
}
else {
UNLOCK ( bar_name.lock );
while ( bar_name.flag != local_sense );
}
}
```

Previous Next

Centralized Barrier

- How fast is it?
    - Assume that the program is perfectly balanced and hence all processors arrive at the barrier at the same time
    - Latency is proportional to P due to the critical section (assume that the lock algorithm exhibits at most O(P) latency)
    - The amount of traffic of acquire section (the CS) depends on the lock algorithm; after everyone has settled in the waiting loop the last processor will generate a BusRdX during release (flag write) and others will subsequently generate BusRd before releasing: O(P)
    - Scalability turns out to be low partly due to the critical section and partly due to O(P) traffic of release
    - No fairness in terms of who exits first

Tree Barrier

- Does not need a lock, only uses flags
    - Arrange the processors logically in a binary tree (higher degree also possible)
    - Two siblings tell each other of arrival via simple flags (i.e. one waits on a flag while the other sets it on arrival)
    - One of them moves up the tree to participate in the next level of the barrier
    - Introduces concurrency in the barrier algorithm since independent subtrees can proceed in parallel
    - Takes log(P) steps to complete the acquire
    - A fixed processor starts a downward pass of release waking up other processors that in turn set other flags
    - Shows much better scalability compared to centralized barriers in DSM multiprocessors; the advantage in small bus-based systems is not much, since all transactions are any way serialized on the bus; in fact the additional log (P) delay may hurt performance in bus-based SMPs

Previous Next

Tree Barrier

```
TreeBarrier ( pid , P) {
unsigned int i , mask;
for ( i = 0, mask = 1; (mask &
pid ) != 0; ++ i , mask <<= 1) {
while (!flag[ pid ][ i ]);
flag[ pid ][ i ] = 0;
}
if ( pid < (P - 1)) {
flag[ pid + mask][ i ] = 1;
while (!flag[ pid ][MAX- 1]);
flag[ pid ][MAX - 1] = 0;
}
for (mask >>= 1; mask > 0; mask >>= 1) {
flag[ pid - mask][MAX-1] = 1;
}
```

- Convince yourself that this works
- Take 8 processors and arrange them on leaves of a tree of depth 3
- You will find that only odd nodes move up at every level during acquire (implemented in the first for loop)
- The even nodes just set the flags (the first statement in the if condition): they bail out of the first loop with mask=1
- The release is initiated by the last processor in the last for loop; only odd nodes execute this loop (7 wakes up 3, 5, 6; 5 wakes up 4; 3 wakes up 1, 2; 1 wakes up 0)

- Each processor will need at most log (P) + 1 flags
- Avoid false sharing: allocate each processor's flags on a separate chunk of cache lines
- With some memory wastage (possibly worth it) allocate each processor's flags on a separate page and map that page locally in that processor's physical memory
  - Avoid remote misses in DSM multiprocessor
  - Does not matter in bus-based SMPs

Previous Next