

eting in "General"

06 05:27 UTC

e Kalpesh

Organised by

Hemangee Kalpesh
Kapoor

Channel

General



Ex: using LL-SC do atomic-swap and fetch-&-increment

Atomic Swap : R4 <=> [R1]

```
TRY: MOV R3, R4      // move exchange value
      LL R2, O(R1)    // load linked
      SC R3, O(R1)    // store conditional
      BEQZ R3, TRY    // branch if store fails (R3 == 0)
      MOV R4, R2      // put load value in R4
```

Fetch-and-increment

```
TRY: LL R2, O(R1)      // load linked
      DADDUI R3, R2, #1 // increment (OK if reg-reg)
      SC R3, O(R1)    // store conditional
      BEQZ R3, TRY    // branch if store fails (R3 == 0)
```

User-level synchronisation using these primitives

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

Spin Locks

```
                DADDUI R2, R0, #1
Lockit:  EXCH  R2, O(R1)    // atomic exchange
                BNEZ   R2, Lockit // already locked
```

- This is used for processors and multi-processor where all data is in memory, i.e. no cache coherence problem
- What about multi-proc with cache coherency?
 - Want to spin on cached copy to avoid memory latency
 - Likely to get cache hits for such variables
 - Locality makes same processor ask for lock again and again

Spinning on cached copy

- Problem: exchange includes a write, which invalidates all other copies. This also generates considerable bus traffic. If more processes want lock => more write misses, etc.
- Solution
 - Start by repeatedly reading (local cached copy) of the variable to see if lock is free. When free try to exchange
 - Here it will race with other processes to acquire lock
 - All procs read old value and store 1 into lock-var
 - Single winner sees '0' and others see '1'
 - When done, the winner releases lock by storing '0' in the lock-var

```
Lockit: LD      R2, 0(R1)    // read lock
        BNEZ    R2, lockit  // not free, so spin
        DADDUI  R2, R0, #1   // load value=1
        EXCH    R2, 0(R1)    // swap
        BNEZ    R2, Lockit  // branch if lock
                               //was not '0'
```

Spinning on cached copy

- Steps
- (1) Processor storing '1' in lock-var invalidates all the processor's cached copies
- (2) therefore, other processors must fetch new lock value from memory. They find that lock is locked, so again test + spin (in local cache)
- (3) After lock release => store '0' => caches copies again invalidated
- (4) Next will get cache miss again => one proc will see '0' first and acquire lock => making lock-var=1
 - Steps from (1) repeat ...





Same using LL-SC pair

```

Lockit:  LL      R2, 0(R1)
          BNEZ   R2, lockit
          DADDUI R2, R0, #1
          SC      R2, 0(R1)
          BEQZ   R2, Lockit
  
```


Implementing Locks

Cache-based
Synchronisation
Wr-invalidate algo
Lock P0 -> (P1 || P2)
-> P2 -> P1 spins

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	<u>Has lock</u>	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	<u>Shared</u>	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	<u>Set lock to 0</u>	<u>(Invalidate received)</u>	<u>(Invalidate received)</u>	<u>Exclusive (P0)</u>	Write invalidate of lock variable from P0.
3		<u>Cache miss</u>	<u>Cache miss</u>	<u>Shared</u>	Bus/directory services P2 cache miss; write-back from <u>P0</u> ; state shared.
4		(Waits while bus/directory busy)	Lock = 0 test 	Shared	Cache miss for P2 satisfied
5		Lock = 0 	<u>Executes swap, gets cache miss</u>	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	<u>Exclusive (P2)</u>	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		<u>Swap completes and returns 1, and sets lock = 1</u>	<u>Enter critical section</u>	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		<u>Spins, testing if lock = 0</u>			None

Point-to-point synchronisation

- Synchronisation between Producer and Consumer
- **Software Algorithm:** uses 'flag' to synchronise

P1	P2
<code>a = f(x);</code>	<code>while (flag == 0)</code>
<code>Flag = 1;</code>	<code>b = g(a)</code>

- If we know that the variable 'a' is initialised to a certain value (say 0), which will be changed to a new value we are interested in by the production event, then we can use 'a' itself as the synchronisation flag:

P1	P2
<code>a=f(x) ;</code>	<code>while (a == 0);</code>
	<code>B= g(a);</code>

Point-to-point synchronisation

- **Hardware Support: Full-Empty Bits**
- Special flag value extended (in research machines) with extra-bit = full-empty bit
- When producer writes to flag it makes bit = full
- When consumer reads from flag it makes bit = empty
- Atomicity maintained by manipulation of read/write to full-empty bit
- Here the program becomes
- Not used (in commercial machines) due to complexity and cost

P1

`a=f(x) ; //set a`

P2

`b = g(a) ; //use a`

Problems with above method

- Less flexible
- Not possible for following reasons:
 - Single producer – multiple consumers not possible
 - Producer wanting to update value several times before telling consumer
 - Which read/write instructions should use bit full/empty?
 - If all use then -> performance problem
 - Some special instr then -> need compiler, language support needed
 - COSTLY. Therefore NOT USED
 - complexity and cost
 - Single producer + multiple consumer cannot be handled

Global (Barrier) Event Synchronisation

- Centralised Software Barrier
- Shared counter
 - To maintain number of processes arrived at Barrier
 - Incremented by each arriving process
 - Increment must be mutually exclusive
- Process increments count
 - If $\text{count} == p$ then it is last process
 - Else busy-wait till Barrier flag is ON
- The last process sets Barrier flag and releases all $(p-1)$ waiting processes

Simple Centralised Barrier algo

```

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 == 0) {
        bar_name.flag = 0; /* first one */
    }

    my_count = bar_name.counter++;

    UNLOCK (bar_name.lock);

    if (my_count == P) { // last one to arrive
        bar_name.counter = 0; //reset count,
        bar_name.flag = 1; } // set flag
    else {
        while (bar_name.flag == 0); // busy wait
    }
}

```

Centralised Barrier with Sense Reversal

- Problem in above barrier, if barrier operation is use consecutively using same bar variable. EX:

some computation

BARRIER(bar1, p);

Some more computation

BARRIER(bar1, p);

- Sample execution leading to the problem. Steps =

1. P1 || P2 || P3 || P4

2. BARRIER (1st)

3. (P1 last, so releases all. P1 makes counter=0; flag=1;) || (P2, P3, P4 wait for flag to be 1)

4. P2, P3 see flag=1 [[P4 has not seen flag=1 as it may be in waiting queue, i.e. not scheduled by OS]]

5. therefore P1, P2, P3 do more computation and again wait for BARRIER

6. P1, P2, P3 has made flag=0 (Reset) but P4 waits for flag=1 (its old instance)

7. when P1, P2, P3 are done they wait for flag to be '1' which will be done by the last processes (here P4)

=> BUT P4 is waiting for flag to be 1 (from earlier instance of P1)

Therefore BARRIER will never be resolved

Solution

- (1) Prevent processes to enter new instance of barrier until all have exited the previous of same barrier
- (2) Use another counter and do not reset flag in new instance until counter has turned to p
 - This new counter counts processes that leave barrier
 - But having more counters incurs latency and contention
 - Current setup requires to reset flag when count=p
- (3) Better solution: Do not reset flag
 - Make processes to wait for a new value of flag for every instance
 - e.g. wait for flag=1 then wait for flag to become '0' then again '1', etc.
 - Maximum we can have processes in two barriers:
 - One old pending and one new on-going
 - We need flag in 2 senses only: '0' and '1'
 - Therefore called Toggle ==> called sense reversal

