Synchronization in Shared Memory multiprocessors

Extract from EE557 slides by Prof. Dubois

Reference: EE557 Textbook Sec. 7.5
SHARED-MEMORY COMMUNICATION

- IMPLICITELY VIA MEMORY

![Diagram of shared memory with threads T1 and T2 accessing a shared variable](https://example.com/diagram.png)

- PROCESSORS SHARE SOME MEMORY
- COMMUNICATION IS IMPLICIT THROUGH LOADS AND STORES
  - NEED TO SYNCHRONIZE
  - NEED TO KNOW HOW THE HARDWARE INTERLEAVES ACCESSES FROM DIFFERENT PROCESSORS

NO ASSUMPTION ON THE RELATIVE SPEED OF PROCESSORS
SYNCHRONIZATION

- NEED FOR "MUTUAL EXCLUSION"
- ASSUME THE FOLLOWING STATEMENTS ARE EXECUTED BY 2 THREADS, T1 AND T2, ON A

\[
\begin{align*}
T1 & : A & \leftarrow & A + 1 \\
T2 & : A & \leftarrow & A + 1 \\
\end{align*}
\]

- THE PROGRAMMER'S EXPECTATION IS THAT, WHATEVER THE ORDER OF EXECUTION OF THE TWO STATEMENTS IS, THE FINAL RESULT WILL BE THAT A IS INCREMENTED BY 2
- HOWEVER PROGRAM STATEMENTS ARE NOT EXECUTED IN AN ATOMIC FASHION.
- COMPILED CODE ON A RISC MACHINE WILL INCLUDE SEVERAL INSTRUCTIONS
- A POSSIBLE INTERLEAVING IS:

\[
\begin{align*}
R1 & : r1 & \leftarrow & A \\
M1 & : r1 & \leftarrow & r1 + 1 \\
W1 & : A & \leftarrow & r1 \\
R2 & : r1 & \leftarrow & A \\
M2 & : r1 & \leftarrow & r1 + 1 \\
W2 & : A & \leftarrow & r1 \\
\end{align*}
\]

- AT THE END A HAS BEEN INCREMENTED BY 1 (NOT 2 AS EXPECTED)
MUTUAL EXCLUSION

WE MUST MAKE PROGRAM STATEMENTS APPEAR ATOMIC

- **CRITICAL SECTIONS**
  - PROVIDED BY LOCK AND UNLOCK PRIMITIVES FRAMING THE STATEMENT(S)
  - MODIFICATIONS ARE "RELEASED" ATOMICALLY AT THE END OF THE CRITICAL SECTION

- **SO THE CODE SHOULD BE:**

  T1
  ```
  lock(La)  
  A<- A+1 
  unlock(La)
  ```

  T2
  ```
  lock(La)  
  A<- A+1 
  unlock(La)
  ```

  /acquire

  /release

HONOR SYSTEM

DEKKER'S ALGORITHMS FOR LOCKING

- **ASSUME A AND B ARE BOTH 0 INITIALLY**

  T1
  ```
  A:=1
  while(B==1);
  <critical section>
  A:=0
  ```

  T2
  ```
  B:=1
  while(A==1);
  <critical section>
  B:=0
  ```

  /acquire

  /release

- **AT MOST ONE PROCESS CAN BE IN THE CRITICAL SECTION AT ANY ONE TIME.**
- **DEADLOCK IS POSSIBLE**
- **COMPLEX (TO SOLVE DEADLOCK AND TO SYNCHRONIZE MORE THAN 2 THREADS)**

USE HARDWARE PRIMITIVES TO SYNCHRONIZE

Excluded for EE457

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved
BARRIER SYNCHRONIZATION

GLOBAL SYNCHRONIZATION AMONG ALL THREADS
• ALL THREADS MUST REACH THE BARRIER BEFORE ANY THREAD IS ALLOWED TO EXECUTE BEYOND THE BARRIER

```
P1
...
BAR := BAR+1;
while (BAR < 2):

P2
...
BAR := BAR +1;
while (BAR < 2):
```

NOTE: NEED A CRITICAL SECTION TO INCREMENT BAR

POINT-TO-POINT SYNCHRONIZATION

```
T1

while (FLAG==0):
print A

T2

A = 1;
FLAG = 1;
/release
/acquire

SIGNAL SENT BY T1 TO T2 THROUGH FLAG (PRODUCER/CONSUMER SYNCHRONIZATION)
• NOTE: NO NEED FOR CRITICAL SECTIONS TO UPDATE AND READ FLAG
```

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved
SYNCHRONIZATION

• **ACQUIRE METHOD**
  - ACQUIRE ACCESS RIGHT TO GET THROUGH THE SYNCHRONIZATION (ENTER CRITICAL SECTION, GO PAST EVENT...)

• **RELEASE METHOD**
  - ENABLE OTHER PROCESSORS TO ACQUIRE THE RIGHT TO GET THROUGH THE SYNCHRONIZATION

• **WAITING ALGORITHM**
  - **BLOCKING**
    - WAITING PROCESSES ARE DESCHEDULED
      - HAS HIGH OVERHEAD
      - ALLOWS PROCESSOR TO WORK ON SOMETHING ELSE
  - **BUSY WAITING**
    - WAITING PROCESSES REPEATEDLY TEST A LOCATION UNTIL IT CHANGES VALUE
    - HAS LOW OVERHEAD BUT HOLDS THE PROCESSOR
    - MAY HAVE HIGH MEMORY/NETWORK TRAFFIC
  - **BUSY-WAITING IS BETTER WHEN**
    - SCHEDULING OVERHEAD IS LARGER THAN EXPECTED WAIT TIME
LOCKS

- **HARDWARE LOCKS**
  - Bus lines can act as barrier
    - When a processor reaches a barrier it drives the line
  - Lock registers
    - Shared registers or flip-flops
    - Set register to acquire the lock
  - Inflexible
    - Not good for general purpose use (limited hardware resources)

- **ISA Support:** Most modern machines use a form of atomic read-modify-write instruction
  - IBM 370: Atomic compare and swap
  - X86: Any instruction can be prefixed with a lock
  - SPARC: Swap
  - MIPS, PowerPC: Support from pairs of instructions
    - Load-locked, store-conditional

These basic mechanisms are used to build software locks.
NAIVE SOFTWARE LOCK

Lock:
LW R2, lock
BNEZR2, Lock
SW R1, lock /
R1 = 1
RET

Unlock:
SW R0, lock
RET

• PROBLEM: LOCK IS NOT ATOMIC--TWO THREADS CAN ACQUIRE THE LOCK AT THE SAME TIME

• SOLUTION: ATOMIC READ/WRITE OR SWAP INSTRUCTION
  • ATOMICALLY READ THE VALUE OF THE LOCATION AND SET IT TO ANOTHER VALUE

• SIMPLEST ONE: TEST_AND_SET (T&S)
  • T&S R1, lock
    • READ lock IN R1
    • WRITE 1 IN lock
    • SUCCESS IF VALUE READ IN R1 IS 0
    • FAILURE IF IT IS 1

T&S MUST BE ATOMIC

READ-SET-and-CHECK what you read
SOFTWARE LOCKS

Lock: T&S R1, lock
      BNEZ R1, Lock
      RET

Unlock: SW R0, lock
       /R₀ = ∅ like $∅$ in MIPS
       RET

• OTHER R/M/W ATOMIC OPERATIONS
  • SWAP R1, MEM_LOC: EXCHANGE THE CONTENT OF R1 AND MEM_LOC
  • FETCH&OP
    • EXAMPLE: F&A (R1, MEM_LOC, CONST), WHERE CONST IS A SMALL VALUE (SPECIAL CASE: 1).
      – FETCH MEM_LOC IN R1, THEN ADD CONST TO MEM_LOC.
  • COMPARE&SWAP
    • CAS (R1, R2, MEM_LOC)
      – COMPARE MEM_LOC TO R1 AND IF THEY ARE EQUAL SWAP R2 AND MEM_LOC
REDUCING THE FREQUENCY OF T&S

• **T&S WITH BACKOFF**
  - INCREASE THE DELAY UNTIL THE NEXT TRIAL AFTER EVERY FAILURE
    - E.G., EXPONENTIAL BACKOFF
      - BACKOFF BY $k \times c^i$ AT THE $i$th TRIAL

• **TEST AND TEST&SET LOCK**
  - TEST WITH ORDINARY LOADS
  - WHEN VALUE CHANGES TO 0, TRY TO OBTAIN LOCK WITH T&S
  - WORKS WELL WITH CACHE

---

**Important**

Once they find that the lock has been released, they all race to get the block in "MODIFIED" state. One of them will succeed and will set the lock to 1.

Other processors also get the block one by one in the "MODIFIED" state but they find the lock has been already set, Then they do again busy wait using ordinary loads in the "SHARED" state.

Processors waiting to get the lock will be spinning on their local cached copy of the block in "SHARED" state first as explained on pages 390-391 of CAQA by H&P 5th edition.
Unlike in multi-cycle processors, it is difficult to implement the atomic operation instructions such as Test-and-Set or Compare-and-Swap in pipelined processors as we cannot do a read and a write operation on a memory location as part of one instruction. And we already know that the following multiple instruction sequence does not guarantee mutual exclusion.

**NAIVE SOFTWARE LOCK**

Can Mutual exclusion be achieved through cache coherency protocol?
An intermediate idea, which is "not so good"

Let us choose to get help from the cache coherency protocol. Let us propose and implement three instructions called "HLW", "JRH", and "SWRH". HLW stands for Hold and Load and will inform the SCU (Snoopy Control Unit) to get the Block in Modified state, do load-word, and then hold the block in "Modified Held-up state" meaning it should not give away the block even if a SCU in another core asks for it. Just tell him to wait. The SWRH stands for Store Word and Release Hold. This releases the Held-up state (and not the lock itself). However, if the lock is already locked, you should release the Held-up state so that the original core who locked it can get the block and release the lock. So you do "JRH" (JRH = Just release the Held-up State) and will inform the SCU not to be selfish any more and give the block to others if they ask. This avoids a dead-lock!

**Proposed Software Lock**

**Lock:**
- LW R2, lock
- BNEZ R2, Lock
- SW R1, lock
- RET

**Lock2:**
- JRH lock
- J Lock
Multiple single-threaded cores: proposed HLW, SWRH, and JRH provide mutual exclusion!

Multiple multi-threaded cores: proposed HLW, SWRH, and JRH fail to provide mutual exclusion!
The above intermediate idea has an important flaw. It provides mutual exclusion between threads running on different cores but does not provide mutual exclusion between multiple threads running within a single core. Moreover it is not desirable to make another SCU requesting for the block to wait for a long time on the bus. There may be several lock operations going on simultaneously and this design lowers efficiency a lot.

So then what is the answer?

The "LL" and the "SC" instruction pair
MIPS II instructions Load Linked (LL) and Store Conditional (SC)\textsuperscript{1}, in conjunction with the cache coherency mechanism and protocol, provide synchronization support for R4000 processors. The two instructions work very much like their simple counterparts load and store. The LL instruction, in addition to doing a simple load, has the side effect of setting a user transparent bit called the load link bit (LLbit). The LLbit forms a breakable link between the LL instruction and a subsequent SC instruction. The SC performs a simple store if and only if the LLbit is set when the store is executed. If the LLbit is not set, then the store will fail to execute. The success or failure of the SC is indicated in the target register of the store after the execution of the instruction. The target register is loaded with 1 in case of a successful store or it is loaded with 0 if the store was unsuccessful. The LLbit is reset upon occurrence of any event that even has potential to modify the lock-variable (like semaphore or counter) while the sequence of code between LL and SC is being executed. The most obvious case where the link will be broken is when an invalidate occurs to the cache line which was the subject of the load. In this case, some other processor successfully completed a store to that shared line.
In general, the link will be broken if following events occur while the sequence of code between LL and SC is being executed:

1. External Update to the cache line containing the lock-variable.
2. External Invalidate to the cache line containing the lock-variable.
3. Intervention or Snoop invalidating cache the line containing the lock-variable.

4. Upon completion of an ERET (return from exception)

The most important features of the LL and SC primitives are:

1. They provide a mechanism for generating all of the common synchronization primitives including test-and-set, counters, sequencers, etc. with no additional overhead.
2. They operate in a fashion so that bus traffic is generated only when the state of the cache line changes; locked words stay in the cache until another processor takes ownership of that cache line.
Load Linked Word \textbf{LL}

Format: \texttt{LL \ rt, offset(base)}

Purpose: To load a word from memory for an atomic read-modify-write.

Description: \( \texttt{rt} \leftarrow \text{memory[base+offset]} \)

The LL and SC instructions provide primitives to implement atomic Read-Modify-Write (RMW) operations for cached memory locations.

There is one active RMW sequence per processor. When an LL is executed it starts the active RMW sequence replacing any other sequence that was active.

The RMW sequence is completed by a subsequent SC instruction that either completes the RMW sequence atomically and succeeds, or does not and fails. See the description of SC for a list of events and conditions that cause the SC to fail and an example instruction sequence using LL and SC.

Executing LL on one processor does not cause an action that, by itself, would cause an SC for the same block to fail on another processor.

An execution of LL does not have to be followed by execution of SC; a program is free to abandon the RMW sequence without attempting a write.
LL and SC extract from **MIPS IV Instruction Set**

**SC**

**Store Conditional Word**

**Format:** SC rt, offset(base)  
**Purpose:** To store a word to memory to complete an atomic read-modify-write.

**Description:** if (atomic_update) then memory[base+offset] ← rt, rt ← 1 else rt ← 0

The LL and SC instructions provide primitives to implement atomic Read-Modify-Write (RMW) operations for cached memory locations.

The SC completes the RMW sequence begun by the preceding LL instruction executed on the processor. If it would complete the RMW sequence atomically, then the least-significant 32-bit word of GPR rt is stored into memory at the location specified by the aligned effective address and a one, indicating success, is written into GPR rt. Otherwise, memory is not modified and a zero, indicating failure, is written into GPR rt.

If any of the following events occurs between the execution of LL and SC, the SC will fail:

- A coherent store is completed by another processor or coherent I/O module into the block of physical memory containing the word. The size and alignment of the block is implementation dependent. It is at least one word and is at most the minimum page size.

- An exception occurs on the processor executing the LL/SC.
  An implementation may detect “an exception” in one of three ways:
  1) Detect exceptions and fail when an exception occurs.
  2) Fail after the return-from-interrupt instruction (RFE or ERET) is executed.
  3) Do both 1 and 2.
LOAD-LOCKED or LOAD-LINKED (LL)

- LOAD LOCKED or LOAD LINKED (LL)
- LL READS lock IN REGISTER Rx
- STORE CONDITIONAL (SC)
- TRIES TO STORE 1 IN lock
- IF SC SUCCEEDS THE SEQUENCE LL-SC WAS ATOMIC
- IF SC FAILS, IT DOES NOT WRITE TO MEMORY; RATHER IT SETS R1 TO 0
- SC CAN FAIL IF IT DETECTS INTERVENING WRITES TO lock SINCE LL
- IT TRIES TO GET THE BUS, BUT ANOTHER SC SUCCEEDS FIRST
- SC CAN FAIL IF IT TRIES TO GET THE BUS, BUT ANOTHER SC SUCCEEDS FIRST
- TWO ADVANTAGES:
  - EASIER TO IMPLEMENT IN A PIPELINE
  - FLEXIBILITY

T&S(Rx,lock):
ADDI R2,R0,1
LL Rx,lock
SC R2,lock
BEQZ R2,T&S
RET

There is no Test and Set instruction in MIPS. Instead we have two instructions LL and SC.

From previous slide:
Unlock: SW R0,lock
RET

Lock:
BEGIN
SW R1,lock
T&S R1,lock
BNEZ R1,lock
RET
BNEZ R1,lock
RET
**LINKED LOAD-LOCKED AND STORE CONDITIONAL**

- **Fancier atomic ops can be implemented by adding code between LL and SC, but**
  - Keep the code short so that SC is likely to succeed
  - Avoid instructions that cannot be undone (e.g., stores, instructions causing exceptions)

**Implementation**

- LL-bit is set when LL is executed
- Bus interface snoops update or invalidate signals and resets LL-bit
- SC tests LL-bit, and fails if bit is reset
- One or multiple LL-bits
  - Could have a single LL bit for all addresses
  - Could attach address to LL-bit
  - Support multiple LL-bits for multiple different addresses

Please refer to the LL Link Register to hold address besides the LL bit stated in pages 390-391 of CAQA by H&P 5th edition.
From the previous to previous slide (slide 238):

**SC CAN FAIL IF**

- IT DETECTS INTERVENING WRITES TO lock SINCE LL
- IT TRIES TO GET THE BUS, BUT ANOTHER SC SUCCEEDS FIRST

The above is rewritten below with additional explanation:

SC can fail if

-- It detects intervening writes to the lock (as revealed by a match of address with the link register) since the last LL

-- It tries to get to the bus and announce "BusUpGr" to elivate the block from the SHARED state to the MODIFIED state, but the SCU (Snoopy Control Unit) from another core succeeds in making the announcement

**Example:**

One LL-bit and one LL link address register

![Diagram showing address translation and link registers]
The 4-threaded core has a 4-entry LL Array. Each entry has two fields:
-- a 1-bit field for LL-bit
-- a 28-bit field for LL-Address-register to hold the block address.
Here a cache block contains 4 words. Hence block address = 28 bits.

Execution of an intervening SC by another thread (whether in the same core or another core in a multi-core system) causes resetting of the LL bit if address matches and hence SC of the original thread (which did LL and set the LL-bit) will fail.
From an earlier slide with some variable names changed for clarity

T&S(Rx,lock):
ADDI R2,R0,1
LL Rx,lock
SC R2,lock
BEQZ R2, T&S
RET

Lock0:
LW R1,lock
BNEZ R1,Lock0
T&S Rx,lock
BNEZ Rx,Lock0
RET

Unlock0:
SW R0,lock
RET

After LL, it is not checking Rx immediately. It insists on completing SC successfully first and then checks if Rx returned by LL was a zero (using BNEZ Rx, Lock0)!

Slight improvement in coding in EE560 CMP project

The EE560 CMP project consists of designing a Chip-Multiprocessor with 4 cores, with each core containing 4 threads. The L2 (MfM) and the 4 cores are connected via a Bus. Bus-based Cache coherency protocol (MOOESI protocol with a Owner_Dirty and a Owner_Clean) is implemented. The above lines are coded as inline code with no function calls and returns. Only SC and SW require the cache block to be in M (Modified) state. LW and LL do not require M state. So the LW in the above code is removed. Since there is no LW, we test the lock with LL to see if the lock is in the unlocked state before attempting to do SC.

Lock0:
LL $6, LOCK($0);  Load the lock from Main Memory
BNE $6, $0, Lock0;  If lock is 1, load it again
ADDI $6, $0, 1;
SC $6, LOCK($0);  Try to set the lock by SC
BEQ $6, $0, Lock0;  If SC fails then start all over

; Critical section of the code here

Unlock0: SW $0, LOCK($0);  No need to do LL or SC as only one thread tries to unlock
LL and SC can be used directly to perform counter incrementation such as barrier counter without a separate lock!

Simple-minded coding with lock:

1. Establish lock as shown on the previous page.

2. Then enter the critical section below to read BAR (the barrier count), increment, and write it back.

```
LW $8, BAR($0);
ADDI $8, $8, 1;
SW $8, BAR($0);
```

3. Release the lock as shown on the previous page.

The TROJAN way of coding without a lock:

```
B0: LL $8, BAR($0); Read the BAR
    ADDI $8, $8, 1; Increment the reg.
    SC $8, BAR($0); Try to write to the BAR
    BEQ $8, $0, B0; Start allover if SC fails
    ; nothing to lock or unlock!
```
In the absence of the LL and SC, one would be trying to write the "Naive" code on the right side. If many threads found the lock in the unlocked state simultaneously, they all finish the first two instructions and will be waiting to do the third, and will eventually succeed doing the third instruction one after another in all three cases (a), (b) and (c) above. Each will get the block in Modified state (one after another), and many will be writing a 1 on the top of a 1, not knowing that the lock was already set.
Illustrating the Case b with a sequence of steps or events.

Between interacting threads running on different cores:

<table>
<thead>
<tr>
<th>Core_0:</th>
<th>Core_1:</th>
<th>Core_2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock0:</td>
<td>Lock0:</td>
<td>Lock0:</td>
</tr>
<tr>
<td>LW R2, lock</td>
<td>LW R2, lock</td>
<td>LW R2, lock</td>
</tr>
<tr>
<td>BNEZ R2, Lock0</td>
<td>BNEZ R2, Lock0</td>
<td>BNEZ R2, Lock0</td>
</tr>
</tbody>
</table>

Say all three cores have the block consisting of the lock in S (SHARED) state.

All three SCU's run to the Bus to announce BusUpGr. Say Core_0 wins. Core_0 executes SW in M state.

<table>
<thead>
<tr>
<th>Core_1:</th>
<th>Core_2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW R1, lock</td>
<td>SW R1, lock</td>
</tr>
</tbody>
</table>

Core_1 and Core_2 try to say on the bus Bus RdX. Say Core_1 succeeds. Core_1 executes SW.

Core_2 says Bus RdX. It succeeds. Core_2 executes SW.

Now all three cores simultaneously think that they have the lock:(

Example from EE560:
Color counting by the 16 threads in the four cores (4 * 4 = 16)

0111_1011
0th_1KB

0111_1011
1st_1KB

16KB

0111_1011
15th_1KB

Count_of_C00
Count_of_C7B
Count_of_CFF

Lock_C00
Lock_C7B
Lock_CFF

0111_1011 = Color 7B = C7B

16K items, each painted in one of the 256 colors, Color_00 to Color_FF.
Count how many times each color is used by updating the counters Count_of_C00 to Count_of_CFF.
Locks Lock_C00 to Lock_CFF are used to avoid RMW races.
Consider the EE560 4-core processor, each core with 4 threads. All together there are 16 threads.

Color Counting job: Suppose there are 16K items, each painted in one of the 256 colors, Color_00 to Color_FF. There is a 16KB array specifying which item is painted with which color. Our job is to find the number (count) of items painted with each color (Count_of_C00 to Count_of_CFF).

Byte-0 indicates the color of item-0. And if the content of Byte-0 is 0111_1011, the item-0 is painted with Color_7B. So we need to increment Count_of_C7B. Assume that the Count_of_C00 to Count_of_CFF are all 32-bit words and can hold any value from 0000_0000 to 0001_0000 hex.

0000_4000

We divide this job among the 16 threads by allocating one-16th of the job to each. We provide each a 1KB of the 16KB and inspect their respective array bytes and increment the common shared variables, Count_of_C00 to Count_of_CFF.

You immediately notice that there is a RMW race among the 16 Threads. Potentially, if the first byte of their respective 1KB arrays happens to be the same namely 0111_1011, then all of them wish to increment the color counter Count_of_C7B. They should do this one at a time without causing any RMW (Read Modify Write) race between any two threads with in the same core or any two threads across the cores. To avoid RMW race, we do need locks. Here we need 256 binary locks: Lock_C00 to Lock_CFF.
Locks Lock_C00 to Lock_CFF are brought into the private L1 caches and are exchanged from one L1 cache to another L1 cache.
This is much faster compared to a design where the locks are in a non-cacheable area (like in the L2 cache or in the MM) and every thread in every core has to access the locks from there taking a lot of clocks. If lock is in "locked" state, threads or cores waiting for it to be released, will be "spinning" around these locks in a "busy wait loop" causing too much traffic on the interconnection network (a bus for EE457).
All this makes it very inefficient. Hence allowing locks to be cached is the way to go.

Here let us consider implementing locks using the MIPS LL and SC instructions.
LL (load locked or load linked)
SC (store conditional)
Description of these instructions can be read in section 2.11 on "Parallelism and Synchronization Instructions" of the EE457 textbook and also in section 7.5 on "Synchronization" in the EE557 textbook.
Assembly code is simplified here for EE457 showing a specific case of just 1 color (Color_7B), one color counter (Count_of_C7B) and one lock (Lock_C7B). Code is implemented in two ways: (A) with locks and (B) without locks

(A) With Locks:

7B_Lock: LL $6, Lock_C7B($0); Load the lock
       BNE $6, $0, 7B_Lock; If lock is already a 1, load it again
       ADDI $6, $0, 1; Prepare to write a 1 into the lock.
       SC $6, Lock_C7B($0); Try to set the lock by SC
       BEQ $6, $0, 7B_Lock; If SC fails, then start allover

    ; Critical section of the code here
       LW $8, Count_of_C7B($0);
       ADDI $8, $8, 1;
       SW $8, Count_of_C7B($0);

7B_Unlock: SW $0, Lock_C7B($0); No need to do LL or SC as only
            ; one thread tries to unlock

(B) Without locks (see how short and simple this code is):

7B_Inc:  LL $6, Count_of_C7B($0); Load the count itself
       ADDI $6, $6, 1; Increment the the count in $6.
       SC $6, Count_of_C7B($0); Try to store the incremented Count
       BEQ $6, $0, 7B_Inc; If SC fails, then start allover