Lock Implementations

- atomic instructions
- load locked / store conditional
- FALSE sharing
- testAndSet lock
- testAndTestAndSet lock
- Ticket lock
- Array Based Queuing lock
- MCS lock
- Reader Writer lock

Lock Implementations

- consider a spinlock implementation based on an IA32 logical shift right instruction [shr]

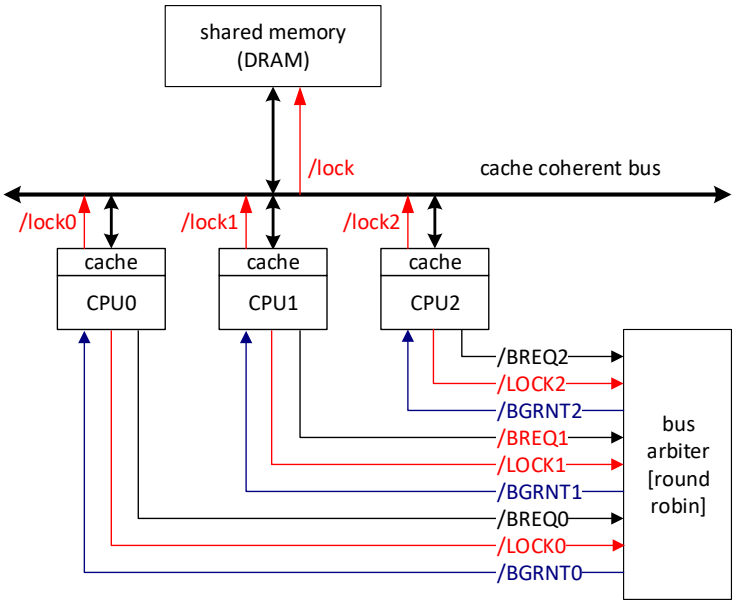

```
;
; simple spin lock (NB: 1 == free, 0 == taken)
;
wait    shr     lock, 1     ; lock in memory
        jnc     wait        ; jump no carry (retry if C == 0)
        ret                 ; return

free    mov     lock, 1     ; lock = 1 (free)
        ret                 ; return
```

if lock free and "shr lock, 1" is executed, lock becomes taken and the carry flag is set

atomically sets lock and returns 1 in carry flag if lock has been acquired

- shr performs a read modify write operation [RMW]
- works in a single CPU system, but not in a multiprocessor
- why? because of the way the CPU accesses memory during a RMW operation

Bus Arbiter



- ONLY one CPU can access shared memory at a time
- CPUs given access to bus and shared memory, one at a time, by bus arbiter

- if CPU wishes to access shared memory, it asserts its bus request signal [/BREQn]
- arbiter grants bus access to one CPU at a time, normally on a bus cycle by bus cycle basis, by asserting its bus grant signal [/BGRNTn – other CPUs wait their turn]
- arbiter normally grants access to bus in a fair manner [round robin]

Atomic Instructions

- atomic RMW memory accesses [read cycle followed by a write cycle] must NOT be interleaved with memory accesses made by other CPUs
- CPUs have special atomic instructions which perform atomic RMW memory accesses
- if bus cycles are arbitrated on a bus cycle by bus cycle basis then

  a CPU could read a lock and find it free; on the next bus cycle another CPU could also read the lock and find it free before the first CPU has been given a bus cycle to set the lock resulting in the lock being allocated to both CPUs

- IA32/x64 CPUs asserts a /LOCK signal [external pin on chip] to inform bus arbiter + external hardware that it is performing an atomic RMW memory access
- bus arbiter must simply lock CPU onto bus while the /LOCK signal is asserted
- atomic RMW cycles often local to a cache thanks to the cache coherency protocol

IA32/x64 Atomic Instructions

- XCHG [exchange] instruction generates an atomic read-modify-write memory access
- use variant which exchanges a register with a memory location

```
;
; testAndSet lock [NB: 0 = free, 1 = taken]
;
wait    mov     eax, 1          ; eax = 1
        xchg    eax, lock       ; exchange eax and lock in memory [atomic]
        test    eax, eax        ; test eax [result of xchg]
        jne     wait            ; re-try if unsuccessful
        ret                     ; return

free    mov     lock, 0         ; clear lock
        ret
```

- XCHG asserts /LOCK when executed, hence atomic

IA32/x64 Atomic Instructions

- a selection of other IA32/x64 instructions can perform atomic RMW cycles if preceded with a LOCK prefix instruction

- bts [bit test and set], btr [bit test and reset], btc [bit test and complement], xadd [exchange and add], cmpxchg [compare and exchange], cmpxchg8b, cmpxchg16b, inc, dec, not, neg, add, adc, sub, sbb, and, or & xor

- only makes sense if one of the operands is a memory location

- consider the *exchange and add* instruction xadd [useful for implementing a ticket lock]

```
lock                    ; lock prefix
xadd    reg, mem        ; tmp = reg + mem, reg = mem, mem = tmp
                        ; [eg set reg = 1 to implement a ticket lock in mem]
```

- without a LOCK prefix, XADD is executed non atomically

Windows Example

- can mix assembly language and C++, BUT…
- x64 VC++ compiler doesn't support an inline assembler, so for Win32/x64 portability use the intrinsics defined in intrin.h

```
LONG __cdecl InterlockedExchange(
  _Inout_  LONG volatile *Target,
  _In_     LONG Value
);
```

- example

```
volatile long lock = 0;                    // declare volatile and initialise lock

while (InterlockedExchange(&lock, 1));     // acquire testAndSet lock
```

- NB: even though long and int are both 32 bit signed integers, types NOT equivalent

Windows Example…

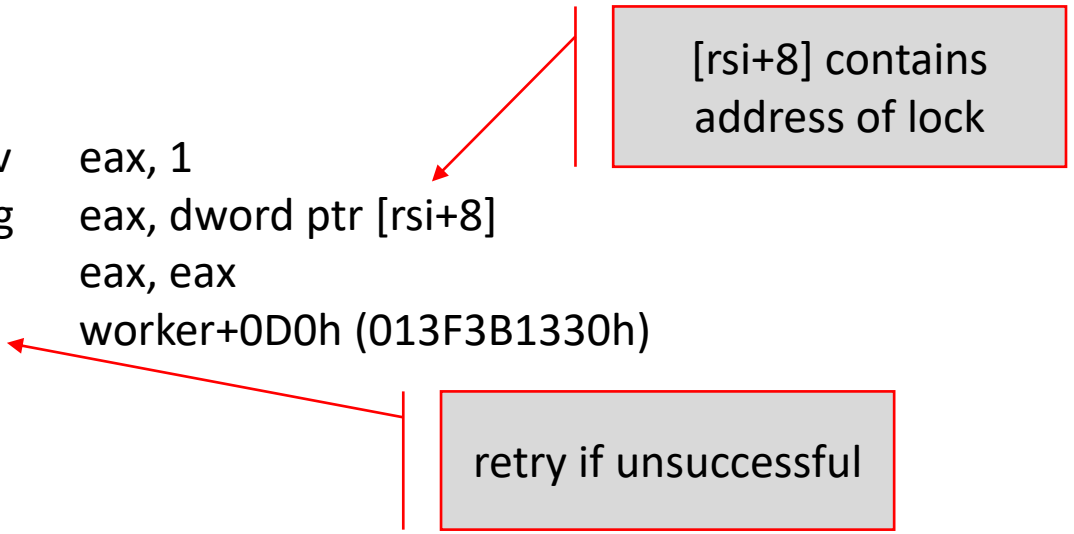
- x64 Release code for

```
while (InterlockedExchange(&lock, 1));
```

obtained using Visual Studio Debugger

- VC++ compiler generates in line code rather than a function call

```
000000013F3B1330  mov     eax, 1
000000013F3B1335  xchg    eax, dword ptr [rsi+8]
000000013F3B1338  test    eax, eax
000000013F3B133A  jne     worker+0D0h (013F3B1330h)
```



Load Locked / Store Conditional Instructions

- alternative approach for performing atomic RMW accesses to memory
- a forerunner of transactional memory
- the execution of a load locked [LL] followed by a store conditional [SC] instruction is used to perform an atomic RMW access to memory
- first used by MIPS CPU [ll/sc]
- also used by Alpha [ldq_l/stq_c], IBM Power PC [lwarx/stwcx] and ARM [ldrex/strex] CPUs

Typical LL/SC Implementation

- each CPU has a lockFlag [LF] and a lockPhysicalAddressRegister [LPAR] used by the LL and SC instructions

- LL Rn, va ; load locked (32 or 64 bit)

```
lockFlag = 1
lockPhysicalAddressRegister = physicalAddress(va)
Rn = [va]
```

- SC Rn, va ; store conditional (32 or 64 bit)

```
if (lockFlag == 1)          ; check lock flag
    [va] = Rn               ; conditional store if lockFlag == 1
Rn = lockFlag               ; used to indicate if store occurred
lockFlag = 0                ; clear lock flag
```

Typical LL/SC Implementation…

- where is the magic?
- if the per CPU lockFlag is *still* set when an associated SC is executed, the store occurs otherwise NO store takes place [conditional store]
- what clears the lockFlag?
- a CPU will clear its lockFlag if any other CPU writes to the physical memory address contained in its lockPhysicalAddressRegister

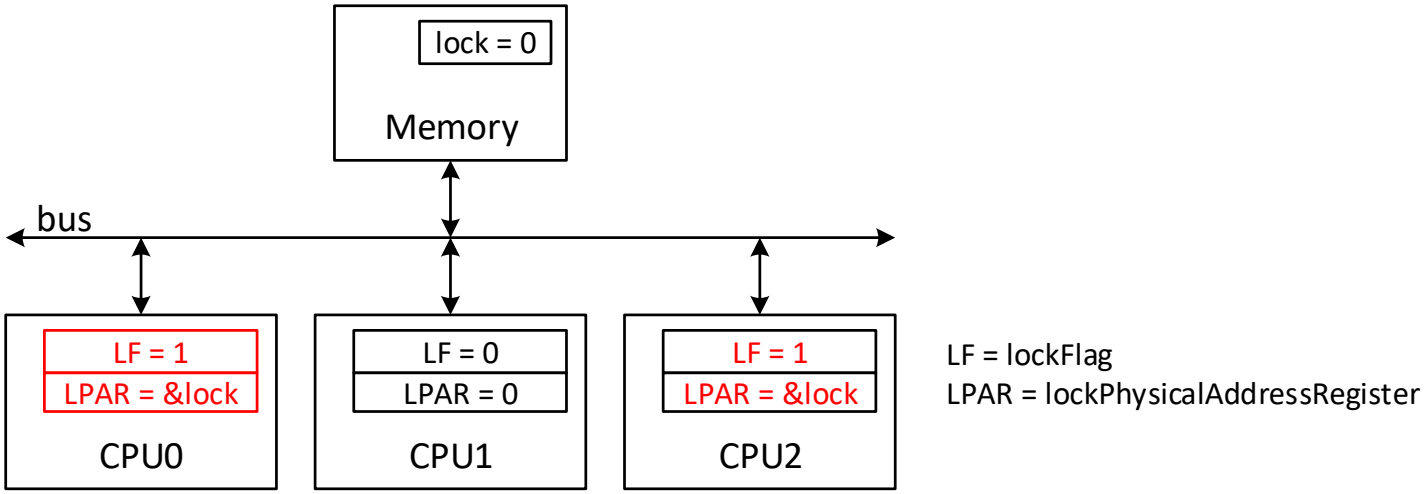
Typical LL/SC Implementation of a TestAndSet Lock

```
ACQUIRE   LL     R1, lock        ; read lock
          BNE    ACQUIRE         ; retry if already set
          MOV    R1, #1          ; r1 = 1
          SC     R1, lock        ; store conditional (R1 = 1 if successful)
          BEQ    ACQUIRE         ; retry if unsuccessful
          MB                     ; memory barrier

          <update shared data structure>

          MB                     ; memory barrier
          MOV    R1, #0          ; clear...
          MOV    lock, R1        ; lock
```

- MB: memory barrier is equivalent to an IA32/x64 memory fence
- if SC is executed successfully, it means that the CPU has performed an atomic RMW testAndSet operation on the lock (no other CPU has written to the lock between the LL and SC instructions)

Typical LL/SC Implementation from a Hardware Perspective

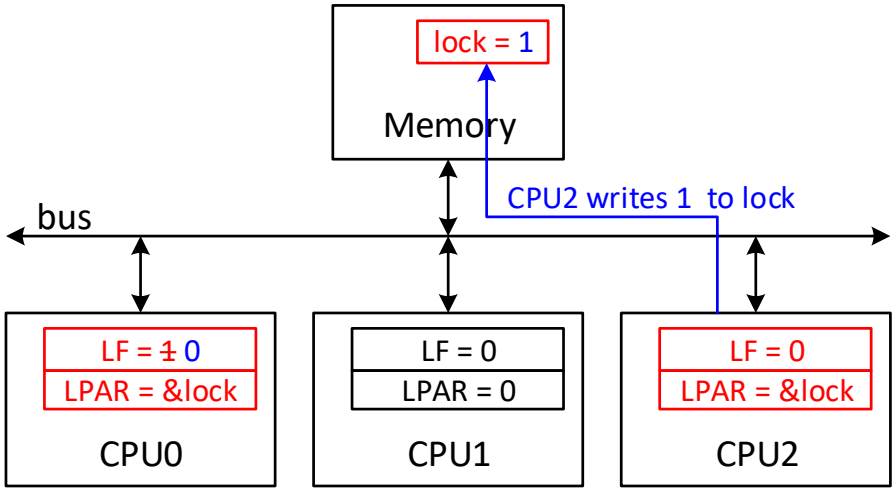
- assume two CPUs try to acquire a testAndSet lock
- imagine CPU0 executes a *LL ra, lock* first, followed by CPU2



- CPU0 and CPU2 lockFlag and LPAR are both set
- as both CPUs find lock free, both CPUs will eventually execute their corresponding SC instructions

Typical LL/SC Implementation from a Hardware Perspective...

- assume CPU2 executes its SC first [gets access to bus first to write 1 to lock]
- CPU2 successfully executes its SC instruction thus setting and acquiring lock



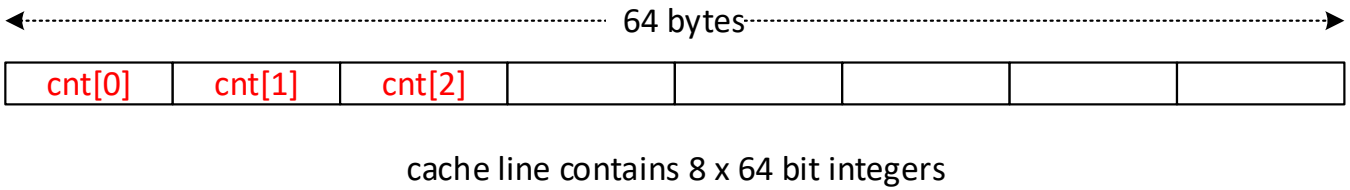
- CPU0 observes a CPU write to the memory address in its LPAR and clears its lockFlag
- when CPU0 executes its SC instruction, it will fail as its lockFlag is 0
- CPU0 must try to obtain lock again [retry]
- a write to the lock is ONLY performed when the lock is obtained which reduces bus traffic and cache line invalidations

Memory Barriers

- memory barriers can be implicit [eg. Intel XCHG instruction] or explicit
- in the LL/SC example, memory barriers are explicit [MB instruction]
- memory barriers guarantee that **ALL** memory accesses are complete [written to the first level cache] **AND** there is NO read ahead before execution proceeds beyond the barrier
- the MB at the start of the critical section prevents any memory accesses within the critical section being made before the lock is acquired - for example pre-fetched reads could potentials return stale data
- the MB at the end of the critical section prevents any memory accesses within the critical section being delayed past the clearing of the lock which prevents the next lock holder from accessing stale data [in case writes are re-ordered by the particular CPU implementation]

Cache Lines – FALSE SHARING

- multi byte caches lines [modern Intel CPUs have 64 byte cache lines]
- 8 x 64 bit integers fit into a 64 byte cache line



cache line contains 8 x 64 bit integers

- consider a global array cnt and a multithreaded program where each thread increments its own cnt[thread]

```
UINT64 cnt[MAXTHREAD];          // UINT64 defined as unsigned long long
```

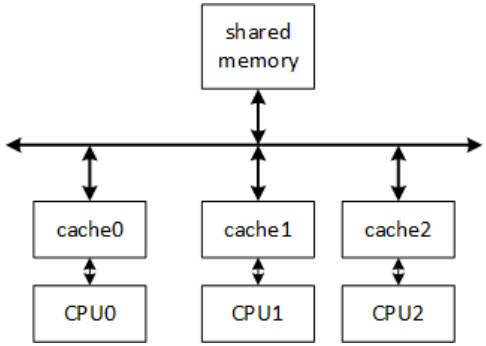
- even though each thread increments its own counter, which is NOT shared with the other threads, the cache line itself is shared with the other threads which results in FALSE sharing

Caches Lines – FALSE SHARING …

- assume MESI cache coherency protocol [used by Intel CPUs]
- assume, initially, that each cache doesn't have a copy of the cache line containing the counters
- assume threads increment their counters non atomically [OK since each counter NOT shared]

- one possible interleave…
- CPU0 reads cache line [Exclusive] and increments counter locally in cache [Modified]
- CPU1 reads cache line, CPU0 intervenes to supply modified cache line [now Shared] and CPU1 writes through to update counter and get ownership of cache line [Exclusive]
- each time a CPU increments its counter it generates zero, one or two bus cycles [depending on the interleave]

- cache line *ping pongs* between caches resulting in considerable cache coherency bus traffic with a consequential loss of throughput



Cost of FALSE SHARING

- synthetic benchmark
- malbec (8 cores 16 threads)
- 32-bit OR 64-bit counter per thread

- FALSESHARING = 0
  - each counter in its own cache line

- FALSESHARING = 1
  - 8 or 16 counters per cache line

- each thread increments its own counter as quickly as possible
  - non-atomically OR
  - atomically (using InterlockedIncrement)

Cost of FALSE SHARING …

- total number of increments per second (ops/s)
- 32-bit integers

	FALSESHARING	1 thread	16 threads	
non-atomic increment	0	339,120,000	5,709,570,000	16.83
non-atomic increment	1	340,950,000	792,760,000	2.32
InterlockedIncrement	0	97,150,000	1,019,440,000	10.49
InterlockedIncrement	1	96,860,000	43,030,000	0.44

Cost of FALSE SHARING …

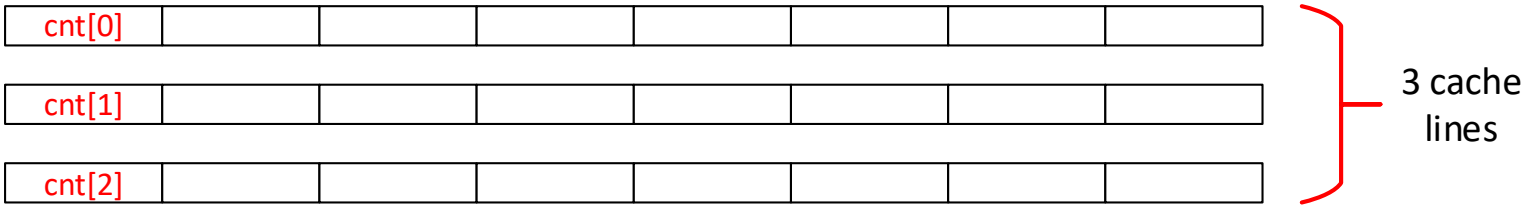
- total number of increments per second (ops/s)
- 64 bit integer

	FALSE SHARING	1 thread	16 threads	
non-atomic increment	0	340,350,000	5,710,260,000	16.77
non-atomic increment	1	338,580,000	1,782,710,000	5.26
InterlockedIncrement	0	97,310,000	1,022,650,000	10.51
InterlockedIncrement	1	97,150,000	77,530,000	0.80

- 64 bit integer FALSE SHARING faster than 32 bit integer because there is less sharing
- 16 x 64bit integers require 2 cache lines, 16 x 32bit integers all fit into a single cache line

Caches Lines

- to avoid FALSE sharing, can allocate each counter in its own cache line



- first approach - use Microsoft specific declaration modifier

```
__declspec(align(64)) UINT64 cnt0;     // cnt0 aligned on a 64 byte boundary
__declspec(align(64)) UINT64 cnt1;     // cnt1 aligned on a 64 byte boundary
__declspec(align(64)) UINT64 cnt2;     // cnt2 aligned on a 64 byte boundary
UINT64 cnt3;                           // ??
```

- align(n) – n must be a constant so can't be determined at runtime
- what about cnt3 declared after cnt2?
- is cnt3 stored in the same cache line as cnt0, cnt1 or cnt2?
- NOT with VC++

Cache Lines…

- second approach - use _aligned_malloc() and _aligned_free()
- can determine cache **lineSz** at runtime using CPUID instruction

```
UINT64 *cnt;                                          // -> counters

cnt = (UINT64*) _aligned_malloc(3*lineSz, lineSz);    // allocate each counter in own cache line
                                                      // allocate 3*lineSz bytes
…                                                     // align on a lineSz byte boundary

cnt[i*lineSz/sizeof(UINT64)]++;                       // increment cnt[i]
                                                      // lineSz/sizeof(UINT64)  integers per cache line

…

_aligned_free(cnt);                                   // free  memory
```

How to allocate C++ objects aligned on a cache line

- may also need to allocate objects in their own cache line(s) to avoid false sharing
- one straightforward approach is to use a class to override new and delete

```
//
// derive from ALIGNEDMA for aligned memory allocation
//
class ALIGNEDMA {

public:

    void* operator new(size_t);          // override new
    void operator delete(void*);         // override delete

};
```

- C++ magic!

How to allocate objects aligned to a cache line …

```
//
// new
//
void* ALIGNEDMA::operator new(size_t sz) {          // aligned memory allocator

    sz = (sz + lineSz - 1) / lineSz * lineSz;       // make sz a multiple of lineSz
    return _aligned_malloc(sz, lineSz);             // allocate on a lineSz boundary
}

//
// delete
//
void ALIGNEDMA::operator delete(void *p) {
    _aligned_free(p);                               // free object
}
```

- NB: size_t is an unsigned integer which can hold an address
- NB: variable lineSz must be initialised to the cache line size

How to allocate Objects aligned to a cache line …

- derive object from ALIGNEDMA
- each object will then be allocated on a cache line boundary, in its own cache line(s)

```
class Object : public ALIGNEDMA {
public:
   volatile int x;
   volatile int y;
};
```

- when a new Object created …

```
Object *o = new Object();
```

- the ALIGNEDMA new function is called to allocate memory for the Object so it will be aligned on a cache line boundary

Intel IA32/x64 Test and Set Lock

- consider the following code for a test and set lock

CPU0	shared data	CPU1
wait mov eax, 1	*obtain lock*	wait mov eax, 1
xchg eax, lock		xchg eax, lock
test eax, eax		test eax, eax
jne wait		jne wait
<update shared data>	*update shared data*	<update shared data>
mov lock, 0	*release lock*	mov lock, 0

Intel IA32/x64 Test and Set Lock

- why does the testAndSet code work on an IA32/x64 CPU?
  - XCHG and LOCK prefixed instructions are serialising instructions - there is an implicit MFENCE which guarantees that **ALL** pending writes are written to the first level cache **AND** there is NO read ahead before execution proceeds past the instruction
  - writes are a made to memory in program order so that when the lock is cleared [mov lock, 0] and written to the first level cache, ALL previous writes to the shared data structure have also been written to the first level cache
  - lock is obtained using a serialising instruction [xchg eax, lock] which prevents read ahead so that data in the shared data structure will not be read until the lock is obtained
  - serialising instructions reduce CPU performance because the CPU has to wait until the write buffer is flushed to the first level cache AND it prevents read ahead

TestAndSet Lock in C/C++

- declaration of lock

```
volatile long lock = 0;                    // lock stored in shared memory
```

- to acquire lock

```
while (InterlockedExchange(&lock, 1));     // wait for lock [0:free 1:taken]
```

- to release lock

```
lock = 0;                                  // clear lock
```

- if InterlockedExchange [XCHG] is used to obtain a lock, performance is poor when there is **contention** for the lock

- again need to consider the operation of the MESI cache coherency protocol

TestAndSet Lock…

- ALL waiting CPUs repeatedly execute an XCHG instruction trying to get hold of lock
- if the lock is contested, the memory accesses made by the XCHG instruction don't benefit from having a cache since the shared cache lines are continually overwritten [even if the lock is a 1, it is overwritten with a 1] which invalidates the entries in the other caches.
- typically, the read part of the XCHG generates bus traffic [INVALID -> SHARED] and so does the write part [SHARED -> EXCLUSIVE].
- the XCHG read and write will probably generate bus cycles [depends on the exact interleaving with other CPUs]
- a write update cache coherency protocol allows the reads to be local cache reads [eg. Firefly]
- the lock is overwritten even if there is NO chance of obtaining the lock
- why isn't there an instruction which conditionally writes a 1 if the value read is 0 [eg. conditional testAndSet as per LL/SC] ?

TestAndTestAndSet Lock

- designed to take advantage of underlying cache behaviour
- to acquire lock [optimistic version]

```
while (InterlockedExchange(&lock, 1))        // try for lock
    while (lock == 1)                        // wait until lock free
        _mm_pause();                         // instrinsic see next slide
```

- to acquire lock [pessimistic version]

```
do {
    while (lock == 1)                        // wait until lock free
        _mm_pause();                         // intrinsic see next slide
} while (InterlockedExchange(&lock, 1));     // try for lock
```

- optimistic version assumes lock is going to be free

IA32 TestAndTestAndSet Lock…

- 7.11.2 PAUSE Instruction

The PAUSE instruction can improves the performance of processors supporting Hyper-Threading Technology when executing "spin-wait loops" and other routines where one thread is accessing a shared lock or semaphore in a tight polling loop. When executing a spin-wait loop, the processor can suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation and flushes the core processor's pipeline. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation and prevent the pipeline flush. In addition, the PAUSE instruction de-pipelines the spin-wait loop to prevent it from consuming execution resources excessively. (See Section 7.11.6.1, "Use the PAUSE Instruction in Spin-Wait Loops," for more information about using the PAUSE instruction with IA-32 processors supporting Hyper-Threading Technology.)

TestAndTestAndSet Lock…

- the advantage is that the test of the lock [lock == 1] is executed entirely within the cache and the *xchg* instruction is only executed when the lock is known to be free and there is a chance of acquiring the lock
- the cached lock variable will be invalidated or updated when the lock is released and only then is an attempt made to obtain the lock by executing an *xchg* instruction
- if the release of the lock invalidates the other shared caches lines then $O(n^2)$ bus cycles may be generated [where n is number of CPUs waiting for lock].
- there will be enough bus activity to interfere with the CPU in the critical section as well as the other CPUs not involved with the lock
- if the lock is held for a long time the impact may not be important, but for short critical sections the lock will be released before the last spurt of activity has subsided resulting in continued bus saturation

TestAndSet Lock with Exponential Back Off

- don't continuously try to acquire lock, delay between attempts

  to acquire lock:

```
d = 1;                                          // initialise back off delay
while (InterlockedExchange(&lock, 1)) {         // if unsuccessful…
    delay(d);                                   // delay d time units
    d *= 2;                                     // exponential back off
}
```

- testAndTestAndSet lock NOT necessary when using a back off scheme
- the longer the CPU has being waiting for the lock, the longer it will have to wait before it attempts to acquire the lock again, possibility of starvation
- apparently works well in practice

Ticket Lock with Proportional Back Off

```
class TicketLock {
public:
    volatile long ticket;                                   // initialise to 0
    volatile long nowServing;                               // initialise to 0
};

void acquire(TicketLock *lock) {                            // acquire lock
    int myTicket = InterlockedExchangeAdd(&lock->ticket, 1); // get ticket [atomic]
    while (myTicket != lock->nowServing)                    // if not our turn…
        delay(myticket - lock->nowServing);                 // delay relative to…
 }                                                          // position in Q

void release(TicketLock *lock) {                            // release lock
    lock->nowServing++;                                     // give lock to next thread
}                                                           // NB: not atomic
```

Ticket Lock with Proportional Back Off…

- think of waiting in a Q in a Bakery, Tourist Office, ISS help desk, A&E, …
- deterministic, ONLY 1 atomic instruction executed per lock acquisition
- FAIR, locks granted in order of request which eliminates the possibility of starvation
- back off proportional to position in Q, if the time in critical section is constant, delay can be calculated such that the subsequent test of lock->nowServing will *just* succeed
- still polls a **shared** location [lock->nowServing] which can cause *considerable* bus traffic when lock released with an invalidate cache coherency protocol [eg. MESI]
- delay not necessary with a write-update protocol [eg. Firefly]
- what happens if ticket overflows? OR number of threads > number of CPUs?
- can check algorithm correctness using Spin 

Array Based Queuing Lock [Anderson]

- locked passed to next thread via an array slot **shared ONLY** between the two threads

```
#define N …                                              // number of threads 0..N-1

struct lock {                                            // shared data structure
    volatile long slot[N][cacheLineSz/sizeof(long)];     // each slot stored in a different cache line
    volatile long nextSlot;                              // initially 0
};

void acquire (struct lock *L, long &mySlot) {            // mySlot is a thread local variable
    mySlot = InterlockedExchangeAdd(&L->nextSlot, 1);    // get next slot
    mySlot = mySlot % N;                                 // make sure mySlot in range 0..N-1
    if (mySlot == N - 1)                                 // check for wrap around and handle by ...
        InterlockedAdd(&L->nextSlot, -N);                // subtracting N from nextSlot
    while (L->slot[mySlot][0] == 0);                     // wait for turn
}

void release (struct lock *L, long mySlot) {
    L->slot[mySlot][0] = 0;                              // initialise slot[mySlot] for next time
    L->slot[(mySlot+1) % N][0] = 1;                      // pass lock to next thread
}
```

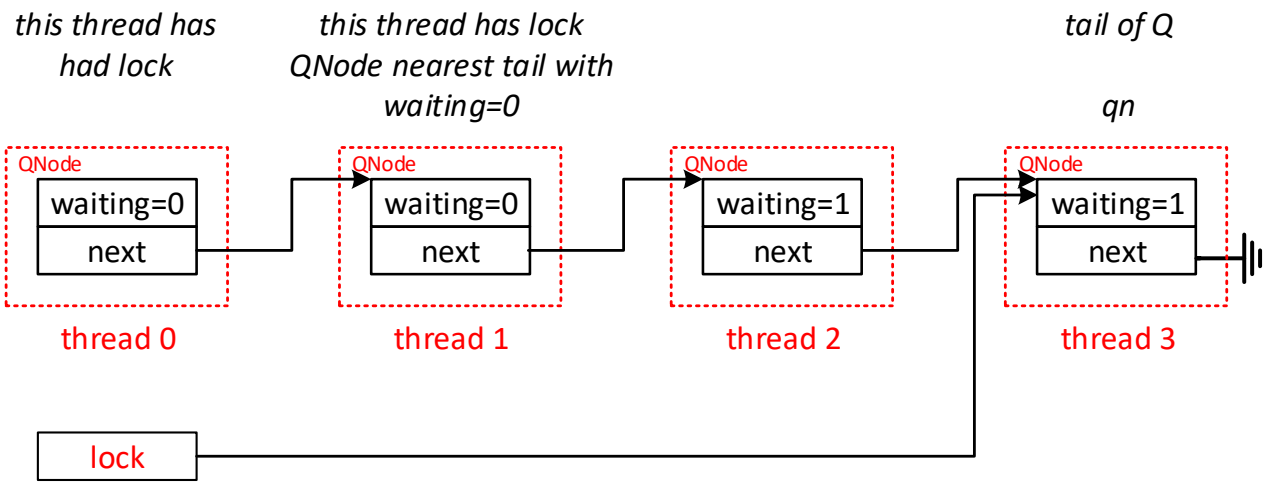
Array Based Queuing Lock...

- locked passed to next thread via an array slot **shared ONLY** between the two threads
  - if thread0 gets slot[0], thread2 gets slot[1] and thread5 gets slot[2]
  - thread0 passes lock to thread2 by setting slot[1] and thread2 passes lock to thread5 by setting slot[2] and so on…
- threads no longer polling a **single shared** location as per ticket lock
- reduces bus traffic for a write-invalidate cache coherency protocol [eg. MESI]
- deterministic, ONLY 1 (occasionally 2) atomic instructions executed per lock acquisition
- struct lock {...} initialised to 0, except slot[0] which must be initialised to 1
- slots recycled
- can check algorithm using Spin 

- WEAKNESS - need to know max number of threads

MCS Lock [Mellor-Crummey and Scott]

- *lockless* queue of waiting threads



- each thread has its own QNode which is linked into a Q of QNodes waiting for lock
- a global variable *lock* points to tail of Q
- acquire lock by adding a thread's QNode [*qn*] to tail and waiting until qn->waiting == 0
- pass lock to next thread by setting qn->next->waiting = 0 [if not at tail of Q]
- number of threads NOT hardwired into algorithm

MCS Lock…

- before looking at the code for the MCS lock need to discuss
  - the Compare and Swap [CAS] instruction and ...
  - thread local storage

Compare and Swap [CAS]

- pseudo C++ version of an atomic CAS

```
atomic long CAS(long *a, long e, long n) { // memory address, expected value, new value
    long r = *a;                           // read contents of memory address
    if (r == e)                            // compare with expected value and if equal…
        *a = n;                            // update memory with new value
    return r;                              // success if e returned
}
```

- returns expected value if exchange took place
- CAS can be mapped to the IA32/x64 compare and exchange instruction

```
cmpxchg mem, reg                   // if (eax == mem)
                                   //     ZF = 1, mem = reg
                                   // else
                                   //     ZF = 0, eax = mem
```

can test eax == expected value to check if swap took place

Compare and Swap…

- make use of following instrinsic defined in intrin.h

  long InterlockedCompareExchange(long volatile *a, long n, long e)

  NB: different parameter order than previous CAS definition

- for convenience can always define

  #define CAS(a, e, n) InterlockedCompareExchange(a, n, e)

MCS Lock…

- derive QNode from ALIGNEDMA so each QNode is allocated in its own cache line, aligned on a cache line boundary

```
class QNode : public ALIGNEDMA {
public:
        volatile int waiting;
        volatile QNode *next;
};
```

- each thread must allocate is own QNode [at start up]

```
QNode *qn = new QNode();
```

Windows Thread Local Storage [Tls]

- allocate next available Tls index

```
DWORD tlsIndex = TlsAlloc();      // get a global tlsIndex that all threads can use
```

- set value stored at tlsIndex

```
QNode *qn = new QNode();      // at start of worker function
TlsSetValue(tlsIndex, qn);    // each worker thread gets its own QNode
```

- get value stored at tlsIndex

```
volatile QNode *qn = (QNode*) TlsGetValue(tlsIndex);
```

- TlsGetValue used by acquire() and release() to get a pointer to the per thread QNode
- NB there is a linux equivalent (see helper.h and helper.cpp)
- alternatively, C++11 compliant compilers, can use

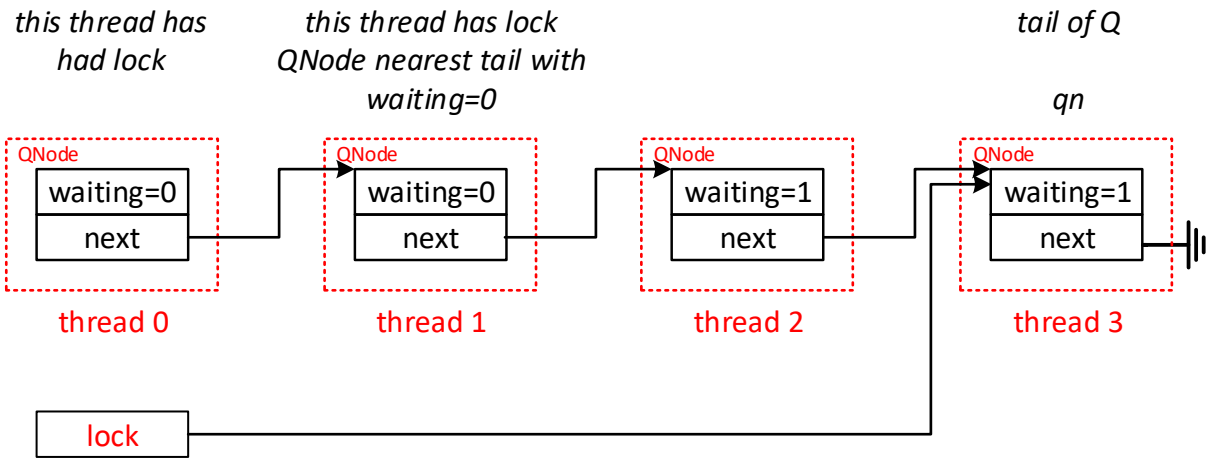
```
int thread_local v;                // there will be a thread local variable v in EACH thread
```

MCS Lock acquire

```
void acquire(QNode **lock) {
    volatile QNode *qn = (QNode*) TlsGetValue(tlsIndex);
    qn->next = NULL;
    volatile QNode *pred = (QNode*) InterlockedExchangePointer((PVOID*) lock, (PVOID) qn);
    if (pred == NULL)
        return;        // have lock
    qn->waiting = 1;
    pred->next = qn;
    while (qn->waiting);
}
```

MCS Lock acquire

- each thread has its own local QNode which can be added to the list of threads waiting for lock

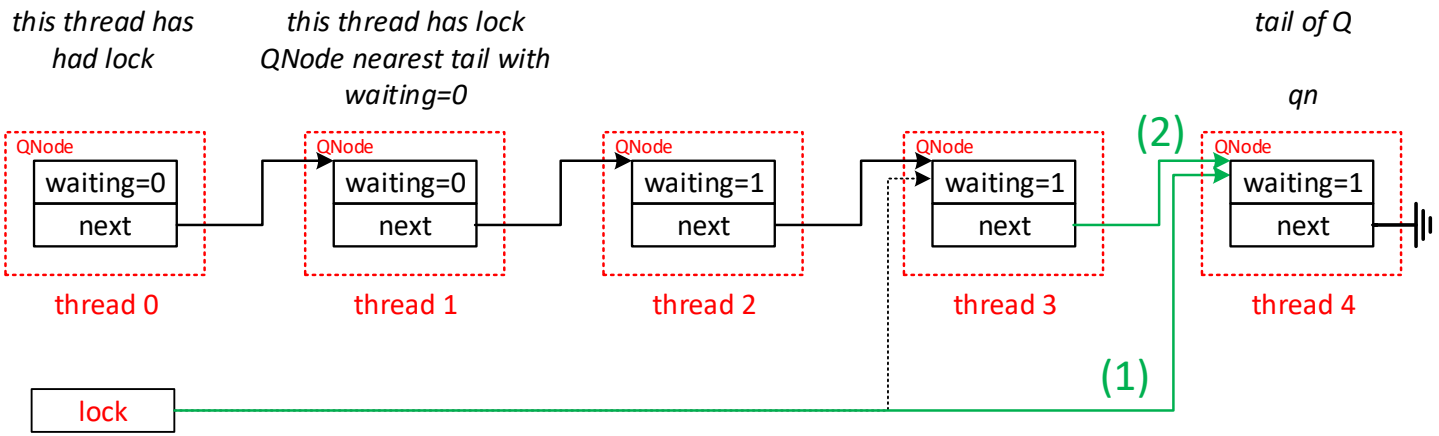


- lock points to tail of Q
- next points to next QNode in Q
- waiting == 1 indicates thread is waiting for lock [QNodes that have already obtained lock remain on Q]

MCS Lock acquire

- thread local QNode [qn] added to Q by executing

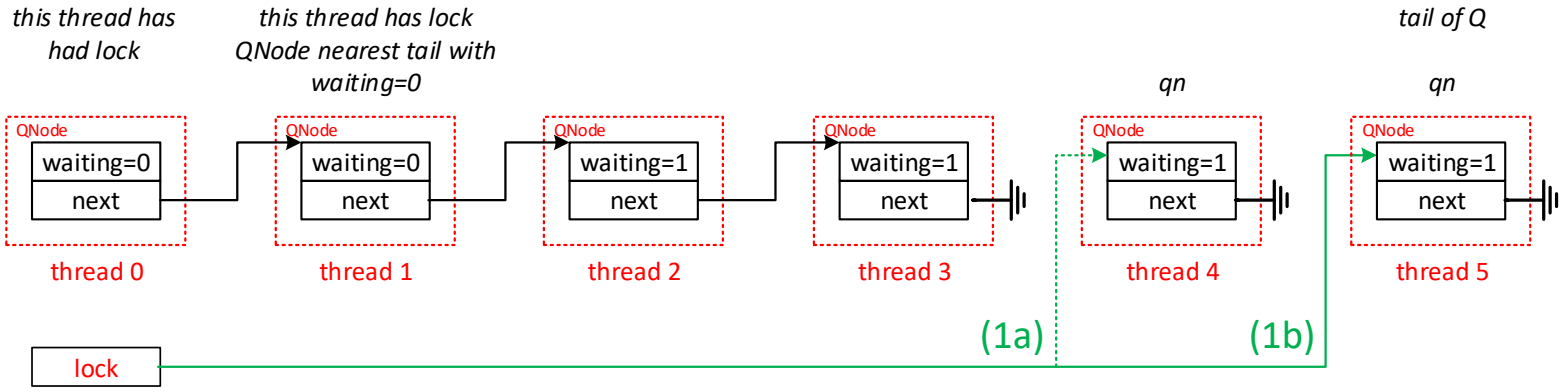
  pred = InterlockedExchangePointer(lock, qn) // atomic {pred = lock, lock = qn} (1)



- if pred == NULL [previous value of lock == NULL] , then at "head" of Q so have lock otherwise…
- set qn->waiting = 1 and…
- link previous tail of Q [pred] to qn by setting pred->next = qn (2) and then wait until qn->waiting == 0

MCS Lock acquire

- what can happen if two or more threads try to acquire lock “simultaneously”?



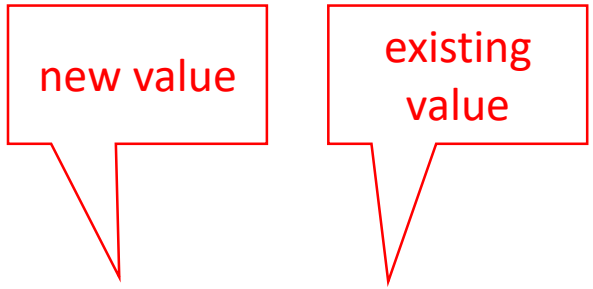
- n threads can execute InterlockedExchangePointer(…) (1a and 1b) before any thread has an opportunity to link its QNode into Q

- the thread local QNode will eventually be linked into Q when thread executes *pred->next = qn*

- this situation needs to be taken into consideration by the code for releasing lock

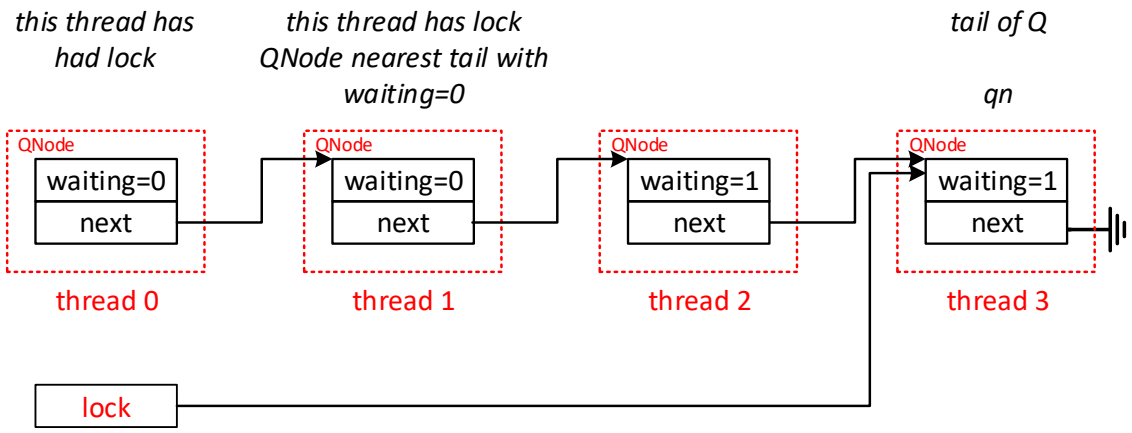
MCS Lock…

```
void release(QNode **lock) {
    volatile QNode *qn = (QNode*) TlsGetValue(tlsIndex);
    volatile QNode *succ;
    if (!(succ = qn->next)) {
        if (InterlockedCompareExchangePointer((PVOID*)lock, NULL, (PVOID) qn) == qn)
            return;
        while ((succ = qn->next) == NULL);      // changed from do … while()
    }
    succ->waiting = 0;                          // simple case
}
```



MCS Lock release...

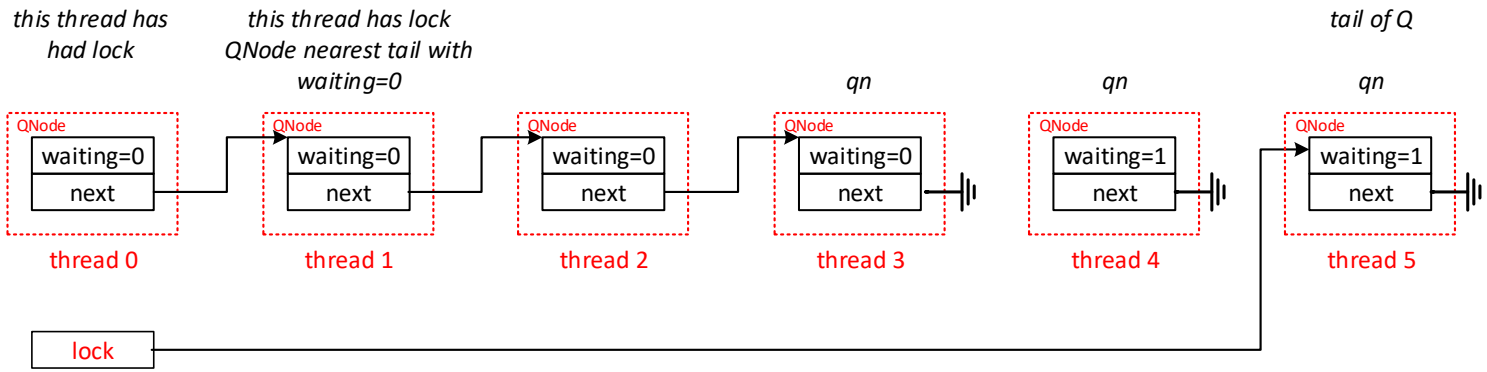
- if (qn->next != NULL), set qn->next->waiting = 0 which passes lock to next thread in Q



- if (qn->next == NULL) use InterlockedCompareExchangePointer(lock, NULL, qn) to atomically set lock = 0 if lock == qn [NB: returns original value of lock]
- if lock was qn, can return as there are no more threads waiting for lock as at tail of Q
- why can InterlockedCompareExchangePointer(lock, NULL, qn) fail?

MCS Lock release...

- there must be QNodes that are not yet attached to Q [i.e. lock != qn]



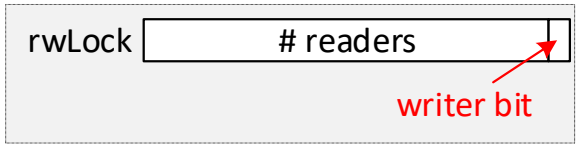
- so wait until qn->next is not NULL
- since succ is assigned qn->next in *wait* loop, succ will eventually point to next QNode [i.e. waits until thread has had an opportunity to link its QNode into Q]
- then set succ->waiting = 0 to pass lock to next thread
- NB: there is NO explicit removal of QNodes from Q
- can check algorithm correctness using Spin

Reader Writer Lock

- so far have only examined data structures protected by locks
- locks inhibit parallelism
- reader writer locks first suggested by Courtois et al. [1971]
- relax mutual exclusion constraint
- allow multiple concurrent readers or a single writer
- reader preference or writer preference
- following code is a reader preference reader writer lock with exponential back off for writers
- can result in indefinite postponement and even starvation of non preferred threads
- possible to construct a fair lock in which readers wait for any earlier writer, and writers wait for any earlier read or write request [eg. queued reader writer lock]

Reader Writer Lock...

```
const WA_flag = 1;
const RC_inc = 2;

volatile int rwLock = 0;

void writerAcquire() {
    int delay = ....
    while (!CAS(&rwLock, 0, WA_flag)) {
        pause(delay);
        delay = ...
    }
}

void writerRelease() {
    FAA(&rwLock, -WA_flag);
}
```




```
void readerAcquire() {
    FAA(&rwLock, RC_inc);
    while (rwLock & WA_flag);
    loadFence();
}

void readerRelease() {
    FAA(&rwLock, -RC_inc);
}
```

- pseudo C/C++
- CAS returns 1 if successful
- FAA - fetch and add

Learning Outcomes

- you are now able to:
  - explain the operation of atomic instructions and their role in implementing locks
  - explain the operation of load locked / store conditional instructions and their role in implementing locks
  - evaluate the cost of false sharing data between threads
  - explain the importance of memory ordering and the function of serializing instructions
  - implement and evaluate numerous locks [eg. TAS, TATAS, Ticket, Array and MCS]
  - implement and evaluate a Reader Writer lock