Readers-Writers Problem

- Multiple threads may access data
  - Readers – will only observe, not modify data
  - Writers – will change the data

- Goal: allow multiple readers or one single writer
  - Thus, lock can be shared amongst concurrent readers

- Can implement with other primitives
  - Keep integer \( i \) – \# or readers or -1 if held by writer
  - Protect \( i \) with mutex
  - Sleep on condition variable when can’t get lock
Implementing shared locks

```c
struct sharedlk {
    int i;
    mutex_t m;
    cond_t c;
};

void AcquireExclusive (sharedlk *sl) {
    lock (sl->m);
    while (sl->i) { wait (sl->m, sl->c); }  
    sl->i = -1;  
    unlock (sl->m);
}

void AcquireShared (sharedlk *sl) {
    lock (sl->m);
    while (sl->i < 0) { wait (sl->m, sl->c); }  
    sl->i++;  
    unlock (sl->m);
}
```
shared locks (continued)

```c
void ReleaseShared (sharedlk *sl) {
    lock (sl->m);
    if (!--sl->i) signal (sl->c);
    unlock (sl->m);
}

void ReleaseExclusive (sharedlk *sl) {
    lock (sl->m);
    sl->i = 0;
    broadcast (sl->c);
    unlock (sl->m);
}
```

- **Note:** Must deal with starvation
struct var {
    int lock;
    int val;
};

void atomic_inc (var *v) {
    while (test_and_set (&v->lock))
    ;
    v->val++;
    v->lock = 0;
}

void atomic_dec (var *v) {
    while (test_and_set (&v->lock))
    ;
    v->val--;
    v->lock = 0;
}
Relaxed consistency model

• Suppose no sequential consistency
  - Recall alpha test_and_set had mb instruction

• What happens if we omit mb?
  - Hardware could violate program order

<table>
<thead>
<tr>
<th>PROGRAM ORDER</th>
<th>VIEW ON OTHER CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write: v-&gt;lock = 1;</td>
<td>v-&gt;lock = 1;</td>
</tr>
<tr>
<td>read: v-&gt;val;</td>
<td></td>
</tr>
<tr>
<td>write: v-&gt;val = read_val + 1;</td>
<td>v-&gt;lock = 0;</td>
</tr>
<tr>
<td>write: v-&gt;lock = 0;</td>
<td>/* danger */</td>
</tr>
<tr>
<td></td>
<td>v-&gt;val = read_val + 1;</td>
</tr>
</tbody>
</table>

• If atomic_dec called where danger, bad val results

• mb in test_and_set preserves program order
  - All ops before mb in program order appear before on all CPUs
  - All ops after mb in program order appear after on all CPUs
Cache coherence

- Performance requires caches
- Sequential consistency requires cache coherence
- Bus-based approaches
  - "Snoopy" protocols, each CPU listens to memory bus
  - Use write through and invalidate when you see a write
  - Or have ownership scheme (e.g., Pentium MESI bits)
  - Bus-based schemes limit scalability
- Cache-Only Memory Architecture (COMA)
  - Each CPU has local RAM, treated as cache
  - Cache lines migrate around based on access
  - Data lives only in cache
cc-NUMA

- Previous slide had “dance hall” architectures
  - Any CPU can “dance with” any memory equally

- An alternative: Non-Uniform Memory Access
  - Each CPU has fast access to some “close” memory
  - Slower to access memory that is farther away
  - Use a directory to keep track of who is caching what

- Originally for machines with many CPUs
  - Now AMD Opterons are kind of like this

- cc-NUMA = cache-coherent NUMA
  - Can also have non-cache-coherent NUMA, though uncommon
  - BBN Butterfly 1 has no cache at all
  - Cray T3D has local/global memory
NUMA and spinlocks

• Test-and-set spinlock has several advantages
  - Simple to implement and understand
  - One memory location for arbitrarily many CPUs

• But also has disadvantages
  - Lots of traffic over memory bus
  - Not necessarily fair (same CPU acquires lock many times)
  - Even less fair on a NUMA machine
  - Allegedly Google had fairness problems even on Opterons

• Idea 1: Avoid spinlocks altogether
• Idea 2: Reduce bus traffic of spinlocks
  - Design lock that spins only on local memory
  - Also gives better fairness
Eliminating locks

• One use of locks is to coordinate multiple updates of single piece of state

• How to remove locks here?
  - Factor state so each variable only has a single writer (Assuming sequential consistency)

• Producer/consumer example revisited
  - Assume one producer, one consumer
  - Why do we need count written by both? To detect buffer full/empty
  - Have producer write in, consumer write out
  - Use in/out to detect buffer state
void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */
        while (((in + 1) % BUFFER_SIZE) == out)
            ;  // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (in == out)
            ;  // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in nextConsumed */
    }
}
Non-blocking synchronization

- Design algorithm to avoid critical sections
  - Any threads can make progress if other threads are preempted
  - Which wouldn’t be the case if preempted thread held a lock

- Requires atomic instructions available on some CPUs
  - E.g., ATOMIC_COMPARE_AND_SWAP: CAS (mem, old, new)
    - If *mem == old, then set *mem = new and return true

- Can implement many common data structures
  - Stacks, queues, even hash tables

- Can implement any algorithm on right hardware
  - Need operation such as ATOMIC_COMPARE_AND_SWAP
    (has property called consensus number = ∞
    – See “Wait Free Synchronization” by Herlihy)
  - Rarely used in practice because inefficient (lots of retries),
    though entire cache kernel written w/o locks using double C&S
Example: stack

struct item {
    /* data */
    struct item *next;
};
typedef struct item *stack_t;

void atomic_push (stack_t *stack, item *i) {
    do {
        i->next = *stack;
    } while (!CAS (stack, i->next, i));
}

item *atomic_pop (stack_t stack) {
    item *i;
    do {
        i = *stack;
    } while (!CAS (stack, i, i->next));
    return i;
}
Benign races

- Can also eliminate locks with race conditions
- Sometimes “cheating” buys efficiency…
- Care more about speed than accuracy

```java
hits++; // each time someone accesses web site

if (!initialized) {
    lock (m);
    if (!initialized) {
        initialize ();
        initialized = 1;
    }
    unlock (m);
}
```

- Know you can get away with race
Read-copy update [McKenney]

- Some data is read way more often than written
- Routing tables
  - Consulted for each packet that is forwarded
- Data maps in system with 100+ disks
  - Updated when disk fails, maybe every $10^{10}$ operations
- Optimize for the common case of reading w/o lock
  - E.g., global variable: `routing_table *rt;`
  - Call `lookup (rt, route);` with no locking
- Update by making copy, swapping pointer
  - E.g., `routing_table *nrt = copy_routing_table (rt);`
  - Update `nrt`
  - Set global `rt = nrt` when done updating
  - All `lookup` calls see consistent old or new table
Garbage collection

• When can you free memory of old routing table?
  - When you are guaranteed no one is using it—how to determine

• Definitions:
  - temporary variable – short-used (e.g., local) variable
  - permanent variable – long lived data (e.g., global rt pointer)
  - quiescent state – when all a thread’s temporary variables dead
  - quiescent period – time during which every thread has been in quiescent state at least once

• Free old copy of updated data after quiescent period
  - How to determine when quiescent period has gone by?
  - E.g., keep count of syscalls/context switches on each CPU
  - Can’t hold a lock across context switch or user mode
MCS lock

- Lock designed by Melloc-Crummey and Scott
  - Goal: reduce bus traffic on cc machines

- Each CPU has a *qnode* structure in local memory

```c
typedef struct qnode {
    struct qnode *next;
    bool locked;
} qnode;
```

- Local can mean local memory in NUMA machine
- Or just its own cache line that gets cached in exclusive mode

- A lock is just a pointer to a *qnode*

```c
typedef qnode *lock;
```

- Lock list of CPUs holding or waiting for lock
- While waiting, just spin on local locked flag
MCS Acquire

```c
acquire (lock *L, qnode *I) {
    I->next = NULL;
    qnode *predecessor = I;
    ATOMIC_SWAP (predecessor, *L);
    if (predecessor != NULL) {
        I->locked = true;
        predecessor->next = I;
        while (I->locked)
            ;
    }
}
```

- **If unlocked**, \( L \) is **NULL**
- **If locked**, no waiters, \( L \) is owner’s qnode
- **If waiters**, \( *L \) is tail of waiter list:
MCS Release w. C&S

release (lock *L, qnode *I) {
    if (!I->next)
        if (ATOMIC_COMPARE_AND_SWAP (*L, I, NULL))
            return;
    while (!I->next)
        ;
    I->next->locked = false;
}

• If I->next NULL and *L == I
    - No one else is waiting for lock, OK to set *L = NULL
• If I->next NULL and *L != I
    - Another thread is in the middle of acquire
    - Just wait for I->next to be non-NULL
• If I->next is non-NULL
    - I->next oldest waiter, wake up w. I->next->locked = false
MCS Release w/o C&S

• What to do if no atomic compare & swap?
• Be optimistic–read *L w. two ATOMIC_SWAPS:
  1. Atomically swap NULL into *L
     - If old value of *L was I, no waiters and we are done
  2. Atomically swap old *L value back into *L
     - If *L unchanged, same effect as ATOMICCOMPARE_AND_SWAP
• Otherwise, we have to clean up the mess
  - Some “userper” attempted to acquire lock between 1 and 2
  - Because *L was NULL, the userper succeeded
    (May be followed by zero or more waiters)
  - Stick old list of waiters on to end of new last waiter
MCS Release w/o C&S code

release (lock *L, qnode *I) {
    if (I->next)
        I->next->locked = false;
    else {
        qnode *old_tail = NULL;
        ATOMIC_SWAP (*L, old_tail);
        if (old_tail == I)
            return;

        qnode *userper = old_tail;
        ATOMIC_SWAP (*L, userper);
        while (I->next == NULL)
            ;
        if (userper != NULL)
            userper->next = I->next;
        else
            I->next->locked = false;
    }
}
Kernel support for synchronization

- Locks must interact with scheduler
  - For processes or kernel threads, must go into kernel (expensive)
  - Common case is you can acquire lock—how to optimize?

- Idea: only go into kernel if you can’t get lock

```c
struct lock {
    int busy;
    thread *waiters;
};

void acquire (lock *lk) {
    while (test_and_set (&lk->busy)) { /* 1 */
        atomic_push (&lk->waiters, self); /* 2 */
        sleep ();
    }
}

void release (lock *lk) {
    lk->busy = 0;
    wakeup (atomic_pop (&lk->waiters));
}
```
Race condition

- Unfortunately, previous slide not safe
  - What happens if release called between lines 1 and 2?
  - wakeup called on NULL, so acquire blocks

- *futex* abstraction solves the problem
  - Ask kernel to sleep only if memory location hasn’t changed

- void futex (int *uaddr, FUTEX_WAIT, int val...);
  - Go to sleep only if *uaddr == val
  - Extra arguments allow timeouts, etc.

- void futex (int *uaddr, FUTEX_WAKE, int val...);
  - Wake up at least val threads sleeping on uaddr

- uaddr is translated down to offset in VM object
  - So works on memory mapped file at different virtual addresses in different processes
Transactions

• Another paradigm for handling concurrency
  - Often provided by databases, but some OSes use them
  - *Vino* OS used to abort after failures
  - OS support for transactional memory now hot research topic

• A *transaction* $T$ is a collection of actions with
  - *Atomicity* – all or none of actions happen
  - *Consistency* – $T$ leaves data in valid state
  - *Isolation* – $T$’s actions all appear to happen before or after every other transaction $T'$
  - *Durability* – $T$’s effects will survive reboots

• Transactions typically executed concurrently
  - But *isolation* means must *appear* not to
  - Must roll-back transactions that use others’ state
  - Means you have to record all changes to undo them
The deadlock problem

mutex_t m1, m2;

void p1 (void *ignored) {
    lock (m1);
    lock (m2);
    /* critical section */
    unlock (m2);
    unlock (m1);
}

void p2 (void *ignored) {
    lock (m2);
    lock (m1);
    /* critical section */
    unlock (m1);
    unlock (m2);
}

• This program can cease to make progress – how?
• Can you have deadlock w/o mutexes?
More deadlocks

• Same problem with condition variables
  - Suppose resource 1 managed by $c_1$, resource 2 by $c_2$
  - A has 1, waits on $c_2$, B has 2, waits on $c_1$

• Or have combined mutex/condition variable deadlock:
  - lock (a); lock (b); while (!ready) wait (b, c);
    unlock (b); unlock (a);
  - lock (a); lock (b); ready = true; signal (c);
    unlock (b); unlock (a);

• One lesson: Dangerous to hold locks when crossing abstraction barriers!
  - I.e., lock (a) then call function that uses condition variable
Deadlocks w/o computers

- Real issue is *resources* & how required
- E.g., bridge only allows traffic in one direction
  - Each section of a bridge can be viewed as a resource.
  - If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
  - Several cars may have to be backed up if a deadlock occurs.
  - Starvation is possible.
Deadlock conditions

1. Limited access (mutual exclusion):
   - Resource can only be shared with finite users.

2. No preemption:
   - once resource granted, cannot be taken away.

3. Multiple independent requests (hold and wait):
   - don’t ask all at once (wait for next resource while holding current one)

4. Circularity in graph of requests
   - All of 1–4 necessary for deadlock to occur
   - Two approaches to dealing with deadlock:
     - pro-active: prevention
     - reactive: detection + corrective action
Prevent by eliminating one condition

1. Limited access (mutual exclusion):
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies

2. No preemption:
   - Threads: threads have copy of registers = no lock
   - Physical memory: virtualized with VM, can take physical page away and give to another process!

3. Multiple independent requests (hold and wait):
   - Wait on all resources at once (must know in advance)

4. Circularity in graph of requests
   - Single lock for entire system: (problems?)
   - Partial ordering of resources (next)
Resource-allocation graph

- View system as graph
  - Processes and Resources are nodes
  - Resource Requests and Assignments are edges

- Process: 

- Resource w. 4 instances: 

- $P_i$ requesting $R_j$: 

- $P_i$ holding instance of $R_j$: 

Example resource allocation graph

Diagram showing resource allocation with nodes labeled $P_1$, $P_2$, $P_3$, and $R_1$, $R_2$, $R_3$, $R_4$. Arrows indicate the allocation of resources.
Graph with deadlock
Is this deadlock?
Cycles and deadlock

• If graph has no cycles $\implies$ no deadlock
• If graph contains a cycle
  - Definitely deadlock if only one instance per resource
  - Otherwise, maybe deadlock, maybe not
• Prevent deadlock w. partial order on resources
  - E.g., always acquire mutex $m_1$ before $m_2$
  - Usually design locking discipline for application this way
Prevention

- Determine safe states based on possible resource allocation
- Conservatively prohibits non-deadlocked states
Claim edges

- Dotted line is claim edge
  - Signifies process may request resource
Example: unsafe state

- Note cycle in graph
  - $P_1$ might request $R_2$ before relinquishing $R_1$
  - Would cause deadlock
Detecting deadlock

- Static approaches (hard)
- Program grinds to a halt
- Threads package can keep track of locks held:

---

Resource-Allocation Graph  Corresponding wait-for graph
Fixing & debugging deadlocks

• Reboot system (windows approach)
• Examine hung process with debugger
• Threads package can deduce partial order
  - For each lock acquired, order with other locks held
  - If cycle occurs, abort with error
  - Detects potential deadlocks even if they do not occur
• Or with transactions, can just tolerate
  - Just abort a transaction when deadlock detected
  - Safe, though inefficient if it happens often
Detecting data races

- Static methods (hard)
- Debugging painful—race might occur rarely
- Instrumentation—modify program to trap memory accesses
- Lockset algorithm (eraser) particularly effective:
  - For each global memory location, keep a “lockset”
  - On each access, remove any locks not currently held
  - If lockset becomes empty, abort: No mutex protects data
  - Catches potential races even if they don’t occur