Implementing Atomic Exchange

- Can not be implemented just by CPU
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time

Implemented by Memory Bus
- memory bus synchronizes every CPUs access to memory
- the two parts of the exchange (read + write) are coupled on bus
- bus ensures that no other memory transaction can intervene
- this instruction is much slower. Higher overhead than normal read or write

Spinlock

- A Spinlock is
  - a lock where waiter spins on looping memory reads until lock is acquired
  - also called “busy waiting” lock

Simple implementation using Atomic Exchange
- spin on atomic memory operation
- that attempts to acquire lock while
- atomically reading its old value

Implementing Spinlocks

- Spin first on fast normal read, then try slow atomic exchange
- when lock appears free use exchange to try to grab it
- if exchange fails then go back to normal read

Blocking Locks

- If a thread may wait a long time
  - it should block so that other threads can run
  - it will then unblock when it becomes runnable (lock available or event notification)

Blocking locks for mutual exclusion
- If lock is held, locker puts itself on wait queue and blocks
- when lock is unlocked, unlocker restarts one thread on wait queue

Blocking locks for event notification
- waiting thread puts itself on a wait queue and blocks
- notifying thread restarts one thread on wait queue (or perhaps all)

Implementing blocking locks presents a problem
- lock structure includes a wait queue and a few other things
- data structure is shared by multiple threads; lock operations are critical sections
- mutual exclusion can be provided by blocking locks (they aren’t implemented yet)
- and so, we need to use spinlocks to implement blocking locks (this gets tricky)

The Importance of Mutual Exclusion

- Shared data
  - data structure that could be accessed by multiple threads
  - typically concurrent access to shared data is a bug

Critical Sections
- sections of code that access shared data

Race Condition
- simultaneous access to critical section by multiple threads
- conflicting operations on shared data structure are arbitrarily interleaved
- unpredictable (non-deterministic) program behaviour — usually a bug (a serious bug)

A mechanism implemented in software (with some special hardware support)
- to ensure critical sections are executed by one thread at a time
- though reading and writing should be handled differently (more later)

For example
- consider the implementation of a shared stack by a linked list...

Synchronization

- Simple Locks
  - companion
  - must synchronize across multiple CPUs
  - accessing memory (variables) shared by multiple threads
  - cannot free transfers among threads (and unit notified by another thread)

- Lock semantics
  - a lock is either held by a thread or available
  - at most one thread can hold a lock at a time
  - a thread attempting to acquire a lock that is already held is forced to wait

- Lock Primitives
  - lock acquire lock, if necessary
  - unlock release lock, allowing another thread to acquire if waiting
  - using locks for the shared stack

Atomic Memory Exchange Instruction

- We need a new instruction
  - to atomically read and write a memory location
  - with no intervening access to that memory location from any other thread allowed

Atomicity
- is a general property in systems
- where a group of operations are performed as a single, indivisible unit

The Atomic Memory Exchange
- one type of atomic memory instruction (there are other types)
- group a load and store together atomically
- exchanging the value of a register and a memory location

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic exchange</td>
<td>r[1] ← m[r[0]]</td>
<td>xchg int, int</td>
</tr>
</tbody>
</table>

Implementing Simple Locks

- Here’s a first cut
  - use a shared global variable for synchronization
  - lock loops until the variable is 0 and then sets it to 1
  - unlock sets the variable to 0

- Sequential test works

- Concurrent test doesn’t always work

- We now have a race in the lock code

- The race exists even at the machine-code level
  - two instructions acquire lock: one to read it free, one to set it held
  - but read by another thread and interpose between these two

- The bug
  - push and pop are critical sections on the shared stack
  - they run in parallel so their operations are arbitrarily interleaved
  - sometimes, this interleaving corrupts the data structure

- What is wrong?

- Implementing Spinlocks
  - use normal read in loop until lock appears free
  - try slow atomic exchange

- The bug
  - push and pop are critical sections on the shared stack
  - they run in parallel so their operations are arbitrarily interleaved
  - sometimes, this interleaving corrupts the data structure

- What is wrong?

- Implementing Atomic Exchange
  - can not be implemented just by CPU
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time

- Implemented by Memory Bus
  - memory bus synchronizes every CPUs access to memory
  - the two parts of the exchange (read + write) are coupled on bus
  - bus ensures that no other memory transaction can intervene
  - this instruction is much slower. Higher overhead than normal read or write

- Implementing Atomic Exchange
  - can not be implemented just by CPU
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time

- Implemented by Memory Bus
  - memory bus synchronizes every CPUs access to memory
  - the two parts of the exchange (read + write) are coupled on bus
  - bus ensures that no other memory transaction can intervene
  - this instruction is much slower. Higher overhead than normal read or write

- Implementing Atomic Exchange
  - can not be implemented just by CPU
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time

- Implemented by Memory Bus
  - memory bus synchronizes every CPUs access to memory
  - the two parts of the exchange (read + write) are coupled on bus
  - bus ensures that no other memory transaction can intervene
  - this instruction is much slower. Higher overhead than normal read or write

- Implementing Atomic Exchange
  - can not be implemented just by CPU
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time

- Implemented by Memory Bus
  - memory bus synchronizes every CPUs access to memory
  - the two parts of the exchange (read + write) are coupled on bus
  - bus ensures that no other memory transaction can intervene
  - this instruction is much slower. Higher overhead than normal read or write

- Implementing Atomic Exchange
  - can not be implemented just by CPU
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time

- Implemented by Memory Bus
  - memory bus synchronizes every CPUs access to memory
  - the two parts of the exchange (read + write) are coupled on bus
  - bus ensures that no other memory transaction can intervene
  - this instruction is much slower. Higher overhead than normal read or write

Reading

- Companion
  - 6 (Synchronization)

- Text
  - shared variables in a thread program, synchronizing threads with semaphores, using threads for parallelism, other concurrency issues
  - READ: 12.4-12.6, parts of 12.7
  - READ: 13.4-13.5, (no equivalent to 12.6), parts of 13.7
### Implementing a Blocking Lock

**Spinlocks**
- *Pros and Cons*:
  - Uncontended locking has lower overhead.
  - Contending for lock has high cost.
- *Use when*:
  - Critical section is small.
  - Contention expected to be minimal.
- *When implementing*:
  - Lock may be held for some time.
  - When event wait is long.

**Blocking Locks**
- *Pros and Cons*:
  - Uncontended locking has higher overhead.
  - Contending for lock has no cost.
- *Use when*:
  - Lock may be held for some time.
  - When contention is high.
  - When event wait may be long.

### Spinlock guard

```c
struct ...  spinlock_unlock (&->spinlock);
```

```c
......

waiter_thread->state = TS_RUNNABLE;
ready_queue_enqueue (waiter_thread);
```

### Monitor automatically exited before block on wait

- *after* waiter blocks, it exits monitor to allow other threads to ... a blocking lock.

### Condition Variables

- *Mechanism to transfer control back and forth between threads*:
  - Uses monitors.
  - CV can only be accessed when monitor lock is held.

### Using Conditions

- *Basic formulation*:
  - One thread enters monitor and may wait for a condition to be established.

### Wait and Notify Semantics

- *Monitor automatically exits block on wait*:
  - Before waiter blocks.

### Using Condition Variables for Disk Read

- *Blocking read*:
  - *call async read as before*.
  - *but now block on condition variable that is given to completion routine*.

### Monitors and Conditions

- *Mutual exclusion plus inter-thread synchronization*:
  - *abstraction supporting explicit locking*.
  - *Condition Variable*:
    - *Threads to synchronize with each other*.
    - *does not release monitor*.
    - *waiter does not run until notifier exits monitor*.
    - *a third thread can intervene and enter monitor before waiter*.
    - *wait blocks until a subsequent signal operation on the variable*.
    - *notify all* unblocks all waiters and continues to hold monitor.
    - *Condition Variable*:
      - *can only be accessed from inside of a monitor*.
      - *with monitor lock held*.

### Drunking Beer Example

- *Beer pitcher is shared data structure with these operations*:
  - *Pour from pitcher into glass*.
  - *Refill pitcher*.

### Monitors

- *Provides mutual exclusion with blocking lock*:
  - *enter lock*.
  - *exit unlock*.

### Blocking Lock Example Scenario

```
void refill (int n) {
  monitor {
    for (int i=0; i<n; i++) {
      glasses++;
      notify;
    }}
```

### Confusion about loop function

- *busywait*:
  - *only inside spinlock*.
  - *thread blocked inside loop body, not busywaiting*.
  - *when finishing thread is not blocking*.

### Locks and Loops

- *Blocking vs Busy Waiting*:
- *Shared Queue Example*:
  - *Unsynchronized Code*
Adding Mutual Exclusion

```c
void enqueue (uthread_queue_t* queue, uthread_t* thread) {
  uthread_monitor_enter (&queue->monitor);
  if (queue->head==0)
    queue->head=thread;
  else
    thread=0;
  uthread_monitor_exit (&queue->monitor);
  return thread;
}
```

### Some Questions About Example

- Why is dequeue a while loop to check for non-empty?
- Why must condition variable be associated with specific monitor?
- Why can’t we use condition variable outside of monitor?

### Implementing Condition Variables

- Some key observations
  - wait, notify and notify_all are called while monitor is held
  - the monitor must be held when they return
- Implementation
  - in the lab
    - look carefully at the implementations of monitor enter and exit
    - understand how these are similar to wait and notify
  - use this code as a guide
  - you also have the code for semaphores, which you might also find helpful

### Using Semaphores to Drink Beer

- Explicit locking not required when using semaphores since atomicity built-in
- Use semaphore to store glasses head by pitcher
- set initial value of empty when creating it
- use this code as a guide

### Semaphore

- Introduced by Edsger Dijkstra for the THE System circa 1968
- recall that he also introduced the "process" (aka "thread") for this system
- wait is analogous to a semaphore in C
- synchronize primitive provide by UNIX to applications
- **A Semaphore**
  - an atomic counter that can never be less than 0
  - attempting to cause non-negative blocks calling thread
    - P(s) (to decrement s (prolong for probe to variangen in Dutch))
    - atomically blocks until s > 0 then decrements s
    - V(s) (increment s (verhogen in Dutch))
    - atomically increases s unblocking threads waiting in P as appropriate

### Using Semaphores

- good building block for implementing many other things
  - monitors
  - initial value of semaphore is 1
  - lock is P(s)
  - unlock is V(s)
- Implementing Condition Variables
  - this is the warm beer problem
  - it took until 2003 before we actually got this right
  - for further reading
    - "Semaphore condition variables."
  - rendezvous: two threads wait for each other before continuing
  - barriers: all threads must arrive at barrier before any can continue

### Lock-Free Atomic Stack in Java

- Recall the problem with concurrent stack
  - a pop could intervene between two steps of push, corrupting linked list
- we solved this problem using locks to ensure mutual exclusion
- now... solve without locks, using atomic compare-and-set of top

### Synchronization in Java (5)

- Monitors using the Lock interface
  - a few variants allow interruptibility, just trying lock...
  - multiple-reader single writer locks

### Other ways to use Semaphores

- Asynchronous Operations
  - create outstanding request semaphore
  - async_read: P(outstanding_request)
  - completion interrupt: V(outstanding_request)
- Rendezvous
  - two threads wait for each other before continuing
  - create a semaphore for each thread initialized to 0

### Synchronization in Java

- Reader-Writer Monitors
  - if we classify critical sections as
    - reader: if only reads the shared data
    - writer: if only writes the shared data
  - Reader-Writer Monitors
    - monitor state is head-for-reading
    - thread A calls monitor_enter() and blocks waiting for monitor to be free
    - thread B calls monitor_enter_read_only(): what do we do?
- Disallowing new readers while writer is waiting
  - is the fair thing to do
  - but, a group of readers can access monitor concurrently
- Policy question
  - monitor state is head-for-reading
  - thread A calls monitor_enter() and blocks waiting for monitor to be free
  - if thread is waiting longer than B, shouldn’t it get the monitor first?
- Allowing new readers while writer is waiting
  - may lead to faster programs by increasing concurrency
  - if readers must wait for old readers to finish, less work is done
  - What should we do
    - normally either provide a fair implementation
    - or allow programmer to choose (that’s what Java does)

### Implementing Monitors

- Initial value of semaphore is 1
- lock is P(s)
- unlock is V(s)

### Implementing Condition Variables

- this is the warm beer problem
  - it took until 2003 before we actually got this right
  - for further reading
    - "Semaphore condition variables."
Problems with Concurrency

- Race Condition
  - competing, unsynchronized access to shared variable
  - from multiple threads
  - at least one of the threads is attempting to update the variable

- Solved with synchronization
  - guaranteeing mutual exclusion for competing accesses

  - but the language does not help you see what data might be shared — can be very hard

Deadlock

- multiple competing actions wait for each other preventing any to complete

- what can cause deadlock?
  - MONITORS
  - CONDITION VARIABLES
  - SEMAPHORES

Recursive Monitor Entry

- What should we do for a program like this

  - void foo () {
    stack_monitor_enter (mon);
    count = 0;
    stack_monitor_exit (mon);
  }

- Here is implementation of lock, is this okay?

  - void lock (extract locking_lock * l) {
    locking_lock (l); // acquire
    while (0 == hold) {
      \[...
      locking_release (l);
    }
  }

Systems with multiple monitors

- We have already seen this with semaphores
- Consider a system with two monitors, a and b

Waiter Graph Can Show Deadlocks

- Waiter graph
  - edge from lock to thread if thread HOLDs lock
  - edge from thread to lock if thread WANTs lock
  - a cycle indicates deadlock

The Dining Philosophers Problem

- Formulated by Edsger Dijkstra to explain deadlock (circa 1965)

  - 5 philosophers sit at a round table with forks placed in between each

  - if the philosopher holds both forks, it must hold all higher precedence locks

  - deadlock: nothing completes because multiple competing actions wait for each other

  - starvation: some actions never complete

  - no abstraction to simply solve problem, major concern intrinsic to synchronization

  - some ways to handle/avoid:
    - precedence hierarchy of locks
    - detect and destroy: notice deadlock and terminate threads

Avoiding Deadlock

- Don’t use multiple threads
  - you’ll have many idle CPU cores and write asynchronous code

- Don’t use shared variables
  - if threads don’t access shared data, no need for synchronization

  - use only one lock at a time
    - deadlock is not possible, unless thread forgets to unlock

- Organize locks into precedence hierarchy
  - each lock is assigned a unique precedence number
  - before thread X acquires a lock, it must hold all higher precedence locks
  - ensures that any thread holding i can’t be waiting for X

  - Detect and destroy
    - if you can’t avoid deadlock, detect when it has occurred
    - break deadlock by terminating threads (e.g., sending them an exception)

Deadlock and Starvation

- Solved problem: race conditions
  - solved by synchronization abstraction: locks, monitors, semaphores

- Unsolved problems when using multiple locks
  - deadlock: nothing completes because multiple competing actions wait for each other
  - starvation: some actions never complete
  - no abstraction to simply solve problem, major concern intrinsic to synchronization
  - some ways to handle/avoid:
    - precedence hierarchy of locks
    - detect and destroy: notice deadlock and terminate threads

Synchronization Summary

- Spinlock
  - one acquire at a time, busy-wait until acquired

  - read atomic read-write memory operation, implemented in hardware

  - use for locks held for short periods (or when minimal lock contention)

- Monitors and Condition Variables
  - blocking locks, stop thread while it is waiting

  - monitor guarantees mutual exclusion

  - condition variables wait notify provides control transfer among threads

- Semaphores
  - blocking atomic counter, stop thread if counter would go negative

  - introduced to coordinate asynchronous resource use

  - use to implement barriers or monitors

  - use to implement something like condition variables, but not quite

- Problems, problems, problems
  - race conditions to be avoided in synchronization

  - deadlock/livelock to be avoided using synchronization carefully