Implementing Atomic Exchange

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
- Atomicty
  - 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[v] ← m[a]</td>
<td>xchg (ra), rv</td>
</tr>
</tbody>
</table>

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 behavior — usually a bug (serious bug)
- Mutual Exclusion
  - 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 ...

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, lock operater puts itself on a waiter queue and blocks
- Waiting thread itself
- Notifying thread releases one thread on waiter queue

Implementing blocking locks presents a problem

- lock data structure includes a waiter 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)

Spinlock

- A spinlock is
  - a lock where waiter spin scanning memory reads until lock is acquired
  - also called “busy waiting” locking

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

- Spin first on normal read
  - normal reads are very fast and efficient compared to exchange
  - use normal read in loop until lock appears free
  - when lock appears free use exchange to try to grab it
  - if exchange fails then go back to normal read

- Busy waiting pros and cons
  - Spinlocks are necessary and okay if spinner only waits a short time
  - But, using a spinlock to wait for a long time, wastes CPU cycles

Synchronization

- We inverted the lock
  - to explicit parallelism: do things at the same time on different processors
  - manage execution separately

- But, we now have two problems
  - coordinating access to memory (variable shared by multiple threads)
  - cannot flow through shared memory (read and update by another thread)

- Synchronization is the mechanism threads use to
  - ensure mutual exclusion of critical sections
  - wait for and notify of the occurrence of events

Readings for These Next Four Lectures

- Text
  - Shared Variables in Threaded Programs - Synchronizing Threads with Semaphore, Using Threads for Parallelism, Other Concurrency Issues
  - 2nd: 12.4-12.5, 12.6, parts of 12.7
  - 1st: 13.4-13.5, (no equivalent to 12.6), parts of 13.7

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

- We now have a race in the lock code

SPINLOCK
Some Questions About Example

- Why does dequeue have a while loop to check for non-empty?
- Why must condition variable be associated with a specific monitor?
- Why can’t we use condition variable outside of a monitor?
  - This is a natural use of the condition variable
  - This is actually required sometimes... can you think of an example?
  - Experience with Processes and Monitors with Mesa, Lampson & Redell, 1980

Implementing Condition Variables

- Some key observations
  - Wait, notify, and notify_all are called while a monitor is held
  - The monitor must be held when they return
  - Wait must release the monitor before blocking and re-acquire before returning

Implementation

- In the implementation of the 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

Avoiding Deadlock

- Don’t use multiple threads
  - You 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 loop 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 0 can not 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)

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)
  - Cooperation invariant: V(outstanding_request)

Problems with Concurrency

- Race Condition
  - Competing, unsynchronized access to shared variable
  - At least one of the threads is attempting to update the variable
  - Solved with synchronization
  - Guaranteed 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 complete
  - What can cause deadlock?
  - MONITORS
  - CONDITION VARIABLES
  - SEMAPHORES

- Condition variables
  - Is await is WaitObject.wait
  - Signal or signalAll is Hansen “notify” (implement Object notify, notifyAll)

- Policy question
  - Monitor state is head-for-reading
  - Thread A calls monitor_enter() and blocks waiting for monitor to be free
  - Thread B calls monitor_read_and_free() what do we do?

- Disallowing new readers while writer is waiting
  - Is the false thing to do
  - Thread A has been waiting longer than B, shouldn’t it get the monitor first?
- Allowing new writers while writer is waiting
  - May lead to faster programs by increasing concurrency
  - If readers must WAIT for old readers and writer 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)

Semaphore

- Introduced by Edsger Dijkstra for the THE System circa 1968
- Recall that he also introduced the "process" (aka "thread") for this system
- Was concerned about concurrency. Semaphores synchronize interrupts
- Synchornization primitive provided by UNIX to applications

A Semaphore is

- An atomic counter that can never be less than 0
- Attempting to make counter negative blocks calling thread
- P(s) ~ try to decrement s (prolong be varlagent in Dutch)
  - Atomically blocks until s > 0 then decrement s
- V(s) ~ increment s (verlagent in Dutch)
  - Atomically increases s unlocking threads waiting in P as appropriate

- 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
    - Google "semaphores condition variables bimel"

- Using Semaphores to Drink Beer
  - Use semaphore to store glasses head by pitcher
  - Set initial value of empty when creating it
  - Pouring and refilling don’t require a monitor

- Data structure
  - Spinlock unlocked whenSpinlock(l)Spinlock(l)Spinlock(l)
  - Spinlock unlocked whenSpinlock(l)Spinlock(l)Spinlock(l)

- Other ways to use Semaphores
  - Asynchronous Operations
    - Create outstanding_request semaphore
    - Async_read: P(outstanding_request)
    - Cooperation invariant: V(outstanding_request)
  - Rendezvous
    - Two threads wait for each other before continuing
    - Create a semaphore for each thread initialized to 0

Experience with Processes and Monitors with Mesa, Lampson and Redell, 1980

- Deadlock avoided, but all philosophers still starved due to timing problem, special case of starvation

- Monitor state is head-for-reading
- Thread A calls monitor_enter() and blocks waiting for monitor to be free
- Thread B calls monitor_read_and_free() what do we do?

- Disallowing new readers while writer is waiting
  - Is the false thing to do
  - Thread A has been waiting longer than B, shouldn’t it get the monitor first?
- Allowing new writers while writer is waiting
  - May lead to faster programs by increasing concurrency
  - If readers must WAIT for old readers and writer 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)
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

Spinlock

- one acquirer at a time, busy-wait until acquired
- need 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 using synchronization
- deadlock/livelock to be avoided using synchronization carefully

Spinlock

- one acquirer at a time, busy-wait until acquired
- need 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 using synchronization
- deadlock/livelock to be avoided using synchronization carefully