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

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
Name Semantics Assembly
atomic exchange r[v] ← m[r[a]]
m[r[a]] ← r[v]
xchg (ra), rv

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 waiter queue and blocks
• when lock is unlocked, unlocker restarts one thread on waiter queue

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

Implementing blocking locks presents a problem
• 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)
Adding Mutual Exclusion
void enqueue (uthread_queue_t* queue, uthread_t* thread) {
  uthread_monitor_enter ...
    queue->tail=0;
  uthread_monitor_exit (&queue->monitor);
  return thread;
}

Synchronization in Java
(5)
Monitors using the Lock interface
• a few variants allow interruptibility, just trying to
  get lock can fail
  • must release lock before returning
  • new API requires release to be called explicitly
    • a pop could intervene between two steps of pushing, updating linked list

Some Questions About Example
Why can't we use condition variable outside of monitor?
Why is does dequeue have a while loop to check for non-empty?
Pouring and refilling don't require a monitor
Allowing new readers while writer is waiting
Reader-Writer Monitors
If we classify critical sections as
• reader if only reads the shared data
• writer if updates the shared data
Then we can weaken the mutual exclusion constraint
• writers require exclusive access to the monitor
• readers do not
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 enter_read only, what do we do?
Disallowing new readers while writer is waiting
• is the fair thing to do
• has a group of readers 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 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)

Other ways to use Semaphores
Asynchronous Operations
• create outstanding_request semaphore
• async_read: P(outstanding_request)
• completion interrupt: V(outstanding_request)
Rendezvous
• two waits thread for each other before continuing
• create a semaphore for each thread initialized to 0

Synchronization in Java (5)
Monitors using the Lock interface
• a few variants allow interruptibility, just trying lock ...
  • multiple-reader single-writer locks

Condition variables
• await is Wait (replaces Object wait)
• signal or signalAll is notify (replaces Object notify, notifyAll)

Semaphore
Introduction by Edsger Dijkstra for the THE System circa 1966
• recall that he also introduced the “process” (aka “thread”) for this system
• was fearful of asynchrony, Semaphores synchronize interrupts
• synchronization primitive provide 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 (prolog for probe to vanlig in Dutch)
    • atomically checks if s > 0 then decrement s
  • V(s)
    • increment s (verhoog in Dutch)
    • atomically increases s unlocking threads waiting in P as appropriate

Lock-Free Atomic Stack in Java
Recall the problem with concurrent stack
• we solved this problem using locks to ensure mutual exclusion
• now solve without locks, using atomic compare-and-set of top
### 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
    - 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 () {
    uthread_monitor_enter (mon);
      count--;
      if (count>0)
        foo();
    uthread_monitor_exit (mon);
  }
  ```

- Here is implementation of lock, is this okay?

  ```
  void lock (extract blocking lock*) {
    spinlock_lock (&spinlock);
    while (i>0) {
      spinlock_unlock (&spinlock);
      uthread_switch (a);
    }
    spinlock_unlock (&spinlock);
  }
  ```

### Systems with multiple monitors

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

  ```
  bar();
  uthread_monitor_exit (b);
  uthread_monitor_enter (a);
  bar();
  uthread_monitor_exit (a);
  ```

### 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 fork placed in between each
  - fork to left and right of each philosopher and each can use only these 2 forks
  - they are either eating or thinking
    - while eating they are thinking and while thinking they are not eating
    - they never speak to each other
  - large bowl of spaghetti at centre of table requires 2 forks to serve
    - deadlock
    - every philosopher holds fork to left waiting for fork to right (or vice versa)
  - starvation (aka livelock)
    - philosophers still starve (ever get both forks) due to timing problem, but avoid deadlock
    - for example

### 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 i, it must hold all higher precedence locks
  - ensures that any thread holding i 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)