Parallel & Concurrent Programming:
Advanced
Synchronization

Emery Berger
CMPSCI 691W - Spring 2006
Why Synchronization?

- Synchronization serves two purposes:
  - Ensure safety for shared updates
    - Avoid race conditions
  - Coordinate actions of threads
    - Parallel computation
    - Event notification
Synch. Operations

- Safety:
  - Locks

- Coordination:
  - Semaphores
  - Condition variables
Safety

- Multiple threads/processes – access shared resource simultaneously
- Safe only if:
  - All accesses have no effect on resource, e.g., reading a variable, or
  - All accesses idempotent
    - E.g., \( a = \text{abs}(x) \), \( a = \text{highbit}(a) \)
  - Only one access at a time: mutual exclusion
### Safety: Example

#### “The too much milk problem”

<table>
<thead>
<tr>
<th>time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Arrive home</td>
<td>Arrive home</td>
</tr>
<tr>
<td>3:05</td>
<td>Look in fridge, no milk</td>
<td>Look in fridge, no milk</td>
</tr>
<tr>
<td>3:10</td>
<td>Leave for grocery</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>3:15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive at grocery</td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:25</td>
<td>Buy milk</td>
<td></td>
</tr>
<tr>
<td>3:35</td>
<td>Arrive home, put milk in fridge</td>
<td></td>
</tr>
<tr>
<td>3:45</td>
<td></td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td>Arrive home, put up milk</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td>Oh no!</td>
</tr>
</tbody>
</table>

#### Model of need to synchronize activities
Why You Need Locks

thread A
if (no milk & & no note)
  leave note
  buy milk
remove note

thread B
if (no milk & & no note)
  leave note
  buy milk
remove note

Does this too much milk
Mutual Exclusion

- Prevent more than one thread from accessing critical section
- Serializes access to section

- Lock, update, unlock:

```plaintext
lock (&l);
update data; /* critical section */
unlock (&l);
```
Too Much Milk: Locks

**thread A**

lock(&l)
if (no milk)
  buy milk
unlock(&l)

**thread B**

lock(&l)
if (no milk)
  buy milk
unlock(&l)
Atomic Operations

- But: locks are also variables, updated concurrently by multiple threads
  - Lock the lock?
  - Answer: use hardware-level atomic operations
  - Test-and-set
  - Compare-and-swap
Test&Set Semantics

int testAndset (int& v) {
    int old = v;
    v = 1;
    return old;
}

pseudo-code: red = atomic

- What’s the effect of testAndset(value) when:
  - value = 0?
    (“unlocked”)
  - value = 1?
    (“locked”)
Lock Variants

- Blocking Locks
- Spin locks
- Hybrids
**Blocking Locks**

- Suspend thread *immediately*
  - Lets scheduler execute another thread
- Minimizes time spent waiting
- But: always causes context switch

```c
void blockinglock (Lock& l) {
    while (testAndSet(l.v) == 1) {
        sched_yield();
    }
}
```
Spin Locks

Instead of blocking, loop until lock released

```c
void spinlock (Lock& l) {
    while (testAndSet(l.v) == 1) {
        ;
    }
}

void spinlock2 (Lock& l) {
    while (testAndSet(l.v) == 1) {
        while (l.v == 1)
            ;
    }
}```
Other Variants

- **Spin-then-yield:**
  - Spin for some time, then yield
    - Fixed spin time
    - Exponential backoff

- **Queuing locks, etc.:**
  - Ensure fairness and scalability
    - Major research issue in 90’s
  - Not used (yet) in real systems
“Safety”

- Locks can enforce mutual exclusion, but notorious source of errors
  - Failure to unlock
  - Double locking
  - Deadlock
  - Priority inversion
    - not an “error” *per se*
What happens when we call `square()` twice when `x == 0`?
Scoped Locks with RAI

- **Scoped Locks:**
  acquired on entry, released on exit
- **C++:** Resource Acquisition is Initialization

```cpp
class Guard {
public:
    Guard (pthread_mutex_t& l)
    : _lock (l)
    { pthread_mutex_lock (&_lock); }

~Guard (void) {
    pthread_mutex_unlock (&_lock);
}
private:
    pthread_mutex_t _lock;
};
```
Scoped Locks: Usage

- Prevents failure to unlock

```c
pthread_mutex_t l;
void square (void) {
    Guard lockIt (&l);
    // acquires lock
    // do stuff
    if (x == 0) {
        return; // releases lock
    } else {
        x = x * x;
    }
    // releases lock
}
```
Double-Locking

- Another common mistake

```c
pthread_mutex_lock (&l);
// do stuff
// now unlock (or not...)
pthread_mutex_lock (&l);
```

- Now what?
  - Can find with static checkers – numerous instances in Linux kernel
  - Better: avoid problem
Recursive Locks

- Solution: recursive locks
  - If unlocked:
    - threadID = pthread_self()
    - count = 1
  - Same thread locks ⇒ increment count
    - Otherwise, block
  - Unlock ⇒ decrement count
    - Really unlock when count == 0
- Default in Java, optional in POSIX
Avoiding Deadlock

- Cycle in locking graph = **deadlock**
- Standard solution: **canonical order** for locks
  - Acquire in increasing order
  - Release in decreasing order
- Ensures deadlock-freedom, but not always easy to do
Increasing Concurrency

- One object, shared among threads

- Each thread is either a *reader* or a *writer*
  - *Readers* – only read data, never modify
  - *Writers* – read & modify data
**Single Lock Solution**

<table>
<thead>
<tr>
<th>thread A</th>
<th>thread B</th>
<th>thread C</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(&amp;l)</td>
<td>lock(&amp;l)</td>
<td>lock(&amp;l)</td>
</tr>
<tr>
<td>Read data</td>
<td>Modify data</td>
<td>Read data</td>
</tr>
<tr>
<td>unlock(&amp;l)</td>
<td>unlock(&amp;l)</td>
<td>unlock(&amp;l)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread D</th>
<th>thread E</th>
<th>thread F</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(&amp;l)</td>
<td>lock(&amp;l)</td>
<td>lock(&amp;l)</td>
</tr>
<tr>
<td>Read data</td>
<td>Read data</td>
<td>Modify data</td>
</tr>
<tr>
<td>unlock(&amp;l)</td>
<td>unlock(&amp;l)</td>
<td>unlock(&amp;l)</td>
</tr>
</tbody>
</table>

- **Drawbacks of this solution?**

Optimization

- Single lock: safe, but limits concurrency
  - Only one thread at a time, but...

- Insight: Safe to have simultaneous readers
  - Must guarantee mutual exclusion for writers
### Readers/Writers

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread C</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlock(&amp;rw)</td>
<td>wlock(&amp;rw)</td>
<td>rlock(&amp;rw)</td>
</tr>
<tr>
<td>Read data</td>
<td>Modify data</td>
<td>Read data</td>
</tr>
<tr>
<td>unlock(&amp;rw)</td>
<td>unlock(&amp;rw)</td>
<td>unlock(&amp;rw)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread D</th>
<th>Thread E</th>
<th>Thread F</th>
</tr>
</thead>
<tbody>
<tr>
<td>rlock(&amp;rw)</td>
<td>rlock(&amp;rw)</td>
<td>wlock(&amp;rw)</td>
</tr>
<tr>
<td>Read data</td>
<td>Read data</td>
<td>Modify data</td>
</tr>
<tr>
<td>unlock(&amp;rw)</td>
<td>unlock(&amp;rw)</td>
<td>unlock(&amp;rw)</td>
</tr>
</tbody>
</table>

- Maximizes concurrency
**R/W Locks – Issues**

- When readers and writers both queued up, who gets lock?
  - Favor readers
    - Improves concurrency
    - Can starve writers
  - Favor writers
  - Alternate
    - Avoids starvation
Synch. Operations

- Safety:
  - Locks

- Coordination:
  - Semaphores
  - Condition variables
Semaphores

- What’s a “semaphore” anyway?

  A visual signaling apparatus with flags, lights, or mechanically moving arms, as one used on a railroad.

- Regulates traffic at critical section
Semaphores in CS

- Computer science: Dijkstra (1965)

A non-negative integer counter with atomic increment & decrement. Blocks rather than going negative.
Semaphore Operations

- **P(sem), a.k.a. wait** = decrement counter
  - If sem = 0, block until greater than zero
  - P = “prolagen” (proberen te verlagen, “try to decrease”)

- **V(sem), a.k.a. signal** = increment counter
  - Wake 1 waiting process
  - V = “verhogen” (“increase”)

UNIVERSITY OF MASSACHUSETTS, AMHERST • Department of Computer Science
Semaphore Example

- More flexible than locks
  - By initializing semaphore to 0, threads can wait for an event to occur

```c
thread A

// wait for thread B
sem.wait();
// do stuff ...
```

```c
thread B

// do stuff, then
// wake up A
sem.signal();
```
Controlling resources:
- E.g., allow threads to use at most 5 files simultaneously
- Initialize to 5

**thread A**
```c
sem.wait();
// use a file
sem.signal();
```

**thread B**
```c
sem.wait();
// use a file
sem.signal();
```
Synch Problem: Queue

- Suppose we have a thread-safe queue
  - `insert(item)`, `remove()`
- Options for remove when queue empty:
  - Return special error value (e.g., NULL)
  - Throw an exception
  - Wait for something to appear in the queue

- `Wait = sleep()`
  - But sleep when holding lock...
    - Goes to sleep
    - Never wakes up!
Condition Variables

- Wait for 1 event, atomically grab lock
  - `wait(Lock& 1)`
    - If queue is empty, wait
      - Atomically releases lock, goes to sleep
      - Reacquires lock when awakened
  - `notify()`
    - Insert item in queue
      - Wakes up one waiting thread, if any
  - `notifyAll()`
    - Wakes up all waiting threads
Next time

- Advanced Thread Programming