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

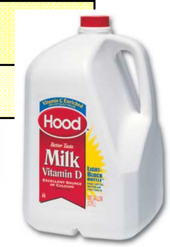
Safety

- 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 = abs(x)`, `a = highbit(a)`
  - Only one access at a time: *mutual exclusion*

Safety: Example

- "The *too much milk* problem"

time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!



- 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 thi

too much milk

Mutual Exclusion

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

- Lock, update, unlock:

`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

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

Spin Locks

- Instead of blocking, loop until lock released

```
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*

Failure to Unlock

```
pthread_mutex_t l;
void square (void) {
  pthread_mutex_lock (&l);
  // acquires lock
  // do stuff
  if (x == 0) {
    return;
  } else {
    x = x * x;
  }
  pthread_mutex_unlock (&l);
}
```

- 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**

```
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

```
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

```
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

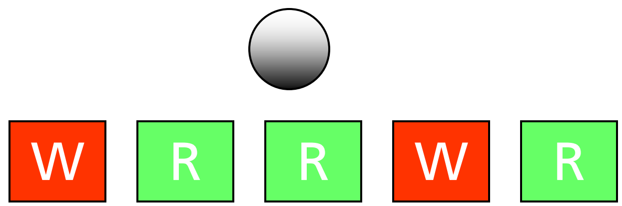
- Solution: **recursive locks**
  - If unlocked:
    - **`threadID = pthread_self()`**
    - **`count = 1`**
  - Same thread locks ⇒ increment count
    - Otherwise, block
  - Unlock ⇒ decrement count
    - **Really unlock** when **`count == 0`**
- Default in Java, optional in POSIX

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

thread A	thread B	thread C

`lock(&l)`
`Read data`
`unlock(&l)`
	`lock(&l)`
	`Modify data`
	`unlock(&l)`
		`lock(&l)`
		`Read data`
		`unlock(&l)`

thread D	thread E	thread F

`lock(&l)`
`Read data`
`unlock(&l)`
	`lock(&l)`
	`Read data`
	`unlock(&l)`
		`lock(&l)`
		`Modify data`
		`unlock(&l)`

- 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

thread A
`rlock(&rw)`
`Read data`
`unlock(&rw)`
	thread B
	`wlock(&rw)`
	`Modify data`
	`unlock(&rw)`
		thread C
		`rlock(&rw)`
		`Read data`
		`unlock(&rw)`

thread D
`rlock(&rw)`
`Read data`
`unlock(&rw)`
	thread E
	`rlock(&rw)`
	`Read data`
	`unlock(&rw)`
		thread F
		`wlock(&rw)`
		`Modify data`
		`unlock(&rw)`

- 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

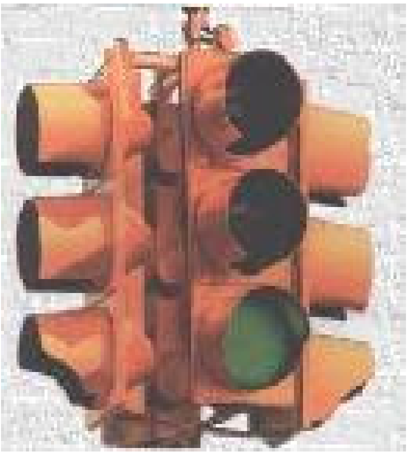
Synch. Operations

- **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" (**pro**beren te ver**lagen**, "try to decrease")
- **V**(sem), a.k.a. **signal** = increment counter
  - Wake 1 waiting process
  - **V =** "verhogen" ("increase")

Semaphore Example

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

thread A

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

thread B

```
// do stuff, then
// wake up A
sem.signal();
```

Counting Semaphores

- Controlling resources:
  - E.g., allow threads to use at most 5 files simultaneously
    - Initialize to 5

thread A

```
sem.wait();
// use a file
sem.signal();
```

thread B

```
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& l)`**
    - 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