CS 5600
Computer Systems

Lecture 5: Synchronization, Deadlock
• Motivating Parallelism
• Synchronization Basics
• Types of Locks and Deadlock
Concurrency vs. Parallelism

• Concurrent execution on a single-core system:

  Core 1: P1 P2 P3 P4 P1 P2 P3 P4 P1 ...

  Time

• Parallel execution on a dual-core system:

  Core 1: P1 P3 P1 P3 P1 P3 P1 P3 P1 ...
  Core 2: P2 P4 P2 P4 P2 P4 P2 P4 P2 ...

  Time
Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

- Transistors
- Clock Speed
- Power Draw
- Perf/Clock
Implications of CPU Evolution

• Increasing transistor count/clock speed
  – Greater number of tasks can be executed concurrently

• However, clock speed increases have essentially stopped in the past few years
  – Instead, more transistors = more CPU cores
  – More cores = increased opportunity for parallelism
Two Types of Parallelism

• Data parallelism
  – **Same task** executes on many cores
  – **Different data** given to each task
  – Example: MapReduce

• Task parallelism
  – **Different tasks** execute on each core
  – Example: any high-end videogame
    • 1 thread handles game AI
    • 1 thread handles physics
    • 1 thread handles sound effects
    • 1+ threads handle rendering
Amdahl’s Law

• Upper bound on performance gains from parallelism
  – If I take a single-threaded task and parallelize it over $N$ CPUs, how much more quickly will my task complete?

• Definition:
  – $S$ is the fraction of processing time that is serial (sequential)
  – $N$ is the number of CPU cores

$$\text{Speedup} \leq \frac{1}{S + \frac{(1-S)}{N}}$$
Example of Amdahl’s Law

• Suppose we have an application that is 75% parallel and 25% serial
  – 1 core: $1/(.25+(1-.25)/1) = ?$
  – 2 core: $1/(.25+(1-.25)/2) = ?$
  – 4 core: $1/(.25+(1-.25)/4) = ?$

• What happens as $N \to \infty$?
  – Speedup $\leq \frac{1}{S+(1-S)/N}$
  – Speedup approaches $1/S$
  – The serial portion of the process has a disproportionate effect on performance improvement
Limits of Parallelism

• Amdahl’s Law is a simplification of reality
  – Assumes code can be cleanly divided into serial and parallel portions
  – In other words, trivial parallelism

• Real-world code is typically more complex
  – Multiple threads depend on the same data
  – In these cases, parallelism may introduce errors

• Real-world speedups are typically < what is predicted by Amdahl’s Law
• Motivating Parallelism
• Synchronization Basics
• Types of Locks and Deadlock
The Bank of Lost Funds

• Consider a simple banking application
  – Multi-threaded, centralized architecture
  – All deposits and withdrawals sent to the central server

```cpp
class account {
    private money_t balance;
    public deposit(money_t sum) {
        balance = balance + sum;
    }
}
```

• What happens if two people try to deposit money into the same account at the same time?
balance = balance + sum;

mov eax, balance
mov ebx, sum
add eax, ebx
mov balance, eax

Thread 1

deposit($50)
mov eax, balance
mov ebx, sum
add eax, ebx
mov balance, eax

Thread 2

deposit($100)
mov eax, balance
mov ebx, sum
add eax, ebx
mov balance, eax
Race Conditions

• The previous example shows a race condition
  – Two threads “race” to execute code and update shared (dependent) data
  – Errors emerge based on the ordering of operations, and the scheduling of threads
  – Thus, errors are nondeterministic
Example: Linked List

- What happens if one thread calls `pop()`, and another calls `push()` at the same time?

```c
elem = pop(&list):
  tmp = list
  list = list->next
  tmp->next = NULL
  return tmp
```

```c
push(&list, elem):
  elem->next = list
  list = elem
```

**Thread 1**
1. `tmp = list`
3. `list = list->next`
5. `tmp->next = NULL`

**Thread 2**
2. `elem->next = list`
4. `list = elem`
Critical Sections

• These examples highlight the critical section problem

• Classical definition of a critical section:
  “A piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread of execution.”

• Two problems
  – Code was not designed for concurrency
  – Shared resource (data) does not support concurrent access
Atomicity

- Race conditions lead to errors when sections of code are interleaved.

- These errors can be prevented by ensuring code executes atomically.

Interleaved Execution

Non-Interleaved (Atomic) Execution
Mutexes for Atomicity

- Mutual exclusion lock (**mutex**) is a construct that can enforce atomicity in code

```c
m = mutex_create();
...
mutex_lock(m);
// do some stuff
mutex_unlock(m);
```
class account {
    mutex m;
    money_t balance

    public deposit(money_t sum) {
        m.lock();
        balance = balance + sum;
        m.unlock();
    }
}
Implementing Mutual Exclusion

• Typically, developers don’t write their own locking-primitives
  – You use an API from the OS or a library

• Why don’t people write their own locks?
  – Much more complicated than they at-first appear
  – Very, very difficult to get correct
  – May require access to privileged instructions
  – May require specific assembly instructions
    • Instruction architecture dependent
Mutex on a Single-CPU System

- On a single-CPU system, the only preemption mechanism is interrupts
  - If interrupts are disabled, the currently executing code is guaranteed to be atomic
- This system is concurrent, but not parallel

```c
void lock_acquire(struct lock * lock) {
    sema_down(&lock->semaphore);
    lock->holder = thread_current();
}

void sema_down(struct semaphore * sema) {
    enum intr_level old_level;
    old_level = intr_disable();
    while (sema->value == 0) { /* wait */ }
    sema->value--;
    intr_level(old_level);
}
```
The Problem With Multiple CPUs

• In a multi-CPU (SMP) system, two or more threads may execute in parallel
  – Data can be read or written by parallel threads, even if interrupts are disabled

```c
CPU 1 - Thread 1
sema_down() {
    while (sema->value == 0) { … }
    sema->value--;
}
```

```c
CPU 2 - Thread 2
sema_down() {
    while (sema->value == 0) { … }
    sema->value--;
}
```
Instruction-level Atomicity

• Modern CPUs have atomic instruction(s)
  – Enable you to build high-level synchronized objects

• On x86:
  – The `lock` prefix makes an instruction atomic
    
    lock inc eax ; atomic increment
    lock dec eax ; atomic decrement
  
  • Only legal with some instructions
  – The `xchg` instruction is guaranteed to be atomic
    
    xchg eax, [addr] ; swap eax and the value in memory
Behavior of xchg

Non-Atomic xchg

CPU 1
eax: 1
1
0
0

memory
0
0
1
1

xchg

CPU 2
eax: 2
0
2
1

xchg

Illegal execution

Atomic xchg

CPU 1
eax: 1
1
0
0

memory
0
0
1
2

xchg

CPU 2
eax: 2
0
2
1

xchg

Legal execution

• Atomicity ensures that each xchgs occurs before or after xchgs from other CPUs
Building a Spin Lock with xchg

spin_lock:

mov eax, 1
xchg eax, [lock_addr]
test eax, eax
jnz spin_lock

spin_unlock:

mov [lock_addr], 0

CPU 1 locks.
CPUs 0 and 2 both try to lock, but cannot.
CPU 1 unlocks.
CPU 0 locks, simply because it requested it slightly before CPU 2.
Well-Behaved Mutexes

• Textbooks refer to the Mutual Exclusion Problem
  – Design a lock mechanism that guarantees the following properties:
    1. **Mutual exclusion**: only one process may hold the lock at a time
    2. **Progress**: the decision about which process gets the lock next cannot be postponed indefinitely
    3. **Bounded waiting**: if all lockers unlock, no process can wait forever to get the lock
  – A mutex having these properties is well-behaved
Building a Multi-CPU Mutex

typedef struct mutex_struct {
    int spinlock = 0; // spinlock variable
    int locked = 0; // is the mutex locked? guarded by spinlock
    queue waitlist; // waiting threads, guarded by spinlock
} mutex;

void mutex_lock(mutex * m) {
    spin_lock(&m->spinlock);
    if (!m->locked){
        m->locked = 1;
        spin_unlock(&m->spinlock);
    } else {
        m->waitlist.add(current_process);
        spin_unlock(&m->spinlock);
        yield();
        // wake up here when the mutex is acquired
    }
}
Building a Multi-CPU Mutex

typedef struct mutex_struct {
    int spinlock = 0;  // spinlock variable
    int locked = 0;    // is the mutex locked? guarded by spinlock
    queue waitlist;    // waiting threads, guarded by spinlock
} mutex;

void mutex_unlock(mutex * m) {
    spin_lock(&m->spinlock);
    if (m->waitlist.empty()) {
        m->locked = 0;
        spin_unlock(&m->spinlock);
    } else {
        next_thread = m->waitlist.pop_from_head();
        spin_unlock(&m->spinlock);
        wake(next_thread);
    }
}
Compare and Swap

• Sometimes, literature on locks refers to compare and swap (CAS) instructions
  – CAS instructions combine an xchg and a test
• On x86, known as compare and exchange

  spin_lock:
  
  mov ecx, 1
  mov eax, 0
  lock cmpxchg ecx, [lock_addr]
  jnz spinlock

  – cmpxchg compares eax and the value of lock_addr
  – If eax == [lock_addr], swap ecx and [lock_addr]
The Price of Atomicity

• Atomic operations are very expensive on a multi-core system
  – Caches must be flushed
    • CPU cores may see different values for the same variable if they have out-of-date caches
    • Cache flush can be forced using a memory fence (sometimes called a memory barrier)
  – Memory bus must be locked
    • No concurrent reading or writing
  – Other CPUs may stall
    • May block on the memory bus or atomic instructions
• Motivating Parallelism
• Synchronization Basics
• Types of Locks and Deadlock
• Lock-Free Data Structures
Other Types of Locks

• Mutex is perhaps the most common type of lock
• But there are several other common types
  – Semaphore
  – Read/write lock
  – Condition variable
    • Used to build monitors
Semaphores

• Generalization of a mutex
  – Invented by Edsger Dijkstra
  – Associated with a positive integer $N$
  – May be locked by up to $N$ concurrent threads

• Semaphore methods
  – wait() – if $N > 0$, $N$--; else sleep
  – signal() – if waiting threads $> 0$, wake one up; else $N$++
The Bounded Buffer Problem

- Canonical example of semaphore usage
  - Some threads **produce** items, add items to a list
  - Some threads **consume** items, remove items from the list
  - Size of the list is bounded

```python
class semaphore_bounded_buffer:
    mutex m
    list buffer
    semaphore S_space = semaphore(N)
    semaphore S_items = semaphore(0)

    def put(item):
        S_space.wait()
        m.lock()
        buffer.add_tail(item)
        m.unlock()
        S_items.signal()

    def get():
        S_items.wait()
        m.lock()
        result = buffer.remove_head()
        m.unlock()
        S_space.signal()
        return result
```
Example Bounded Buffer

<table>
<thead>
<tr>
<th>buffer</th>
<th>S_items</th>
<th>S_space</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Thread 1 | Thread 2 | Thread 3 | Thread 4
---------|---------|---------|---------

put(a)  put(b)  put(c)  get()
Read/Write Lock

- Sometimes known as a **shared mutex**
  - **Many threads** may hold the **read lock** in parallel
  - Only **one thread** may hold the **write lock** at a time
    - Write lock cannot be acquired until all read locks are released
    - New read locks cannot be acquired if a writer is waiting
- Ideal for cases where updates to shared data are rare
  - Permits maximum read parallelization
Example Read/Write Lock

<table>
<thead>
<tr>
<th>list</th>
<th>readers</th>
<th>writers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a, b]</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
When is a Semaphore Not Enough?

In this case, semaphores are not sufficient
– weight is an unknown parameter
– After each put(), totalweight must be checked

```python
class weighted_bounded_buffer:
    mutex m
    list buffer
    int totalweight

get(weight):
    while (1):
        m.lock()
        if totalweight >= weight:
            result = buffer.remove_head()
            totalweight -= result.weight
            m.unlock()
            return result
        else:
            m.unlock()
            yield()

put(item):
    m.lock()
    buffer.add_tail(item)
    totalweight += item.weight
    m.unlock()

• No guarantee the condition will be satisfied when this thread wakes up
• Lots of useless looping :
```
Condition Variables

• Construct for managing control flow amongst competing threads
  – Each condition variable is associated with a mutex
  – Threads that cannot run yet `wait()` for some condition to become satisfied
  – When the condition is satisfied, some other thread can `signal()` to the waiting thread(s)

• Condition variables are not locks
  – They are control-flow managers
  – Some APIs combine the mutex and the condition variable, which makes things slightly easier
Condition Variable Example

```python
class weighted_bounded_buffer:
    mutex m
    condition c
    list buffer
    int totalweight = 0
    int neededweight = 0

get(weight):
    m.lock()
    if totalweight < weight:
        neededweight += weight
        c.wait(m)
    neededweight -= weight
    result = buffer.remove_head()
    totalweight -= result.weight
    m.unlock()
    return result

put(item):
    m.lock()
    buffer.add_tail(item)
    totalweight += item.weight
    if totalweight >= neededweight
        and neededweight > 0:
        c.signal(m)
    else:
        m.unlock()
```

- In essence, we have built a construct of the form: `wait_until(totalweight >= weight)`

- `signal()` hands the locked mutex to a waiting thread
- `wait()` unlocks the mutex and blocks the thread
- When `wait()` returns, the mutex is locked
Monitors

• Many textbooks refer to monitors when they discuss synchronization
  – A monitor is just a combination of a mutex and a condition variable

• There is no API that gives you a monitor
  – You use mutexes and condition variables
  – You have to write your own monitors
    • In OO design, you typically make some user-defined object a monitor if it is shared between threads

• Monitors enforce mutual exclusion
  – Only one thread may access an instance of a monitor at any given time
  – synchronized keyword in Java is a simple monitor
Be Careful When Writing Monitors

Original Code

```
get(weight):
    m.lock()
    if totalweight < weight:
        neededweight += weight
        c.wait(m)
    neededweight -= weight
    result = buffer.remove_head()
    totalweight -= result.weight
    m.unlock()
    return result

put(item):
    m.lock()
    buffer.add_tail(item)
    totalweight += item.weight
    if totalweight >= neededweight and neededweight > 0:
        c.signal(m)
    else:
        m.unlock()
```

Modified Code

```
get(weight):
    m.lock()
    if totalweight < weight:
        neededweight += weight
        c.wait(m)
    result = buffer.remove_head()
    totalweight -= result.weight
    m.unlock()
    return result

put(item):
    m.lock()
    buffer.add_tail(item)
    totalweight += item.weight
    if totalweight >= neededweight and neededweight > 0:
        c.signal(m)
        neededweight -= item.weight
    else:
        m.unlock()
```

Incorrect! The mutex is not locked at this point in the code
# Pthread Synchronization API

## Mutex

```c
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
pthread_mutex_lock(&m);
pthread_mutex_trylock(&m);
pthread_mutex_unlock(&m);
pthread_mutex_destroy(&m);
```

## Condition Variable

```c
pthread_cond_t c;
pthread_cond_init(&c, NULL);
pthread_cond_wait(&c &m);
pthread_cond_signal(&c);
pthread_cond广播(&c);
pthread_cond_destroy(&c);
```

## Read/Write Lock

```c
pthread_rwlock_t rwl;
pthread_rwlock_init(&rwl, NULL);
pthread_rwlock_rdlock(&rwl);
pthread_rwlock_wrlock(&rwl);
pthread_rwlock_tryrdlock(&rwl);
pthread_rwlock_trywrlock(&rwl);
pthread_rwlock_unlock(&rwl);
pthread_rwlock_destroy(&rwl);
```

## POSIX Semaphore

```c
sem_t s;
sem_init(&s, NULL, <value>);
sem_wait(&s);
sem_post(&s);
sem_getvalue(&s, &value);
sem_destroy(&s);
```
Layers of Locks

Thread 1
- mutex A
- mutex B
- lock A
- lock B
- // do something
- unlock B
- unlock A

Thread 2
- lock B
- lock A
- // do something
- unlock A
- unlock B

Deadlock :(
When Can Deadlocks Occur?

• Four classic conditions for deadlock
  1. Mutual exclusion: resources can be exclusively held by one process
  2. Hold and wait: A process holding a resource can block, waiting for another resource
  3. No preemption: one process cannot force another to give up a resource
  4. Circular wait: given conditions 1-3, if there is a circular wait then there is potential for deadlock

• One more issue:
  5. Buggy programming: programmer forgets to release one or more resources
Circular Waiting

- Simple example of circular waiting
  - Thread 1 holds lock $a$, waits on lock $b$
  - Thread 2 holds lock $b$, waits on lock $a$
Avoiding Deadlock

• If circular waiting can be prevented, no deadlocks can occur

• Technique to prevent circles: lock ranking
  1. Locate all locks in the program
  2. Number the locks in the order (rank) they should be acquired
  3. Add assertions that trigger if a lock is acquired out-of-order

• No automated way of doing this analysis
  – Requires careful programming by the developer(s)
Lock Ranking Example

<table>
<thead>
<tr>
<th>#1: mutex A</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock A</td>
<td>assert(islocked(A))</td>
<td>lock B</td>
</tr>
<tr>
<td>assert(islocked(A))</td>
<td>lock B</td>
<td>lock A</td>
</tr>
<tr>
<td>lock B</td>
<td>// do something</td>
<td>// do something</td>
</tr>
<tr>
<td>unlock B</td>
<td>unlock A</td>
<td>unlock A</td>
</tr>
<tr>
<td>unlock A</td>
<td></td>
<td>unlock B</td>
</tr>
</tbody>
</table>

- Rank the locks
- Add assertions to enforce rank ordering
- In this case, Thread 2 assertion will fail at runtime
When Ranking Doesn’t Work

• In some cases, it may be impossible to rank order locks, or prevent circular waiting

• In these cases, eliminate the hold and wait condition using trylock()

Example: Thread Safe List

```python
class SafeList {
    method append(SafeList more_items) {
        lock(self)
        lock(more_items)
    }
}
```

Problem:
Safelist A, B
Thread 1: A.append(B)
Thread 2: B.append(A)

Solution: Replace lock() with trylock()

```python
method append(SafeList more_items) {
    while (true) {
        lock(self)
        if (trylock(more_items) == locked_OK) {
            break
        }
    }
    unlock(self)
}
// now both lists are safely locked
```
• Motivating Parallelism
• Synchronization Basics
• Types of Locks and Deadlock
Beyond Locks

• Mutual exclusion (locking) solves many issues in concurrent/parallel applications
  – Simple, widely available in APIs
  – (Relatively) straightforward to reason about

• However, locks have drawbacks
  – Priority inversion and deadlock only exist because of locks
  – Locks reduce parallelism, thus hinder performance
Lock-Free Data Structures

• Is it possible to build data structures that are thread-safe without locks?
  – YES

• Lock-free data structures
  – Include no locks, but are thread safe
  – However, may introduce starvation
    • Due to retry loops (example in a few slides)
Wait-Free Data Structures

• Wait-free data structures
  – Include no locks, are thread safe, and avoid starvation
  – Wait-free implies lock-free
    • Wait-free is much stronger than lock-free

• Wait-free structures are very hard to implement
  – Impossible to implement for many data structures
  – Often restricted to a fixed number of threads
Advantages of Going Lock-Free

• Potentially much more performant than locking
  – Locks necessitate waits, context switching, CPU stalls, etc…
• Immune to thread killing
  – If a thread dies while holding a lock, you are screwed
• Immune to deadlock and priority inversion
  – You can’t deadlock/invert when you have no locks :)}
Caveats to Going Lock-Free

• Very few standard libraries/APIs implement these data structures
  – Implementations are often platform-dependent
  – Rely on low-level assembly instructions
  – Many structures are very new, not widely known

• Not all data structures can be made lock-free
  – For many years, nobody could figure out how to make a lock-free doubly linked list

• Buyer beware if implementing yourself
  – Very difficult to get right
Lock-free Queue Example: Enqueue

• Usage: one reader, one writer

```cpp
void enqueue(int& t) {
    last->next = new Node(t);
    last = last->next;

    // garbage collect dequeued nodes
    while (first != divider) {
        Node * tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

```cpp
class Node {
    Node * next;
    int data;
};

// Queue pointers
volatile Node * first;
volatile Node * last;
volatile Node * divider;

lock_free_queue() {
    // add the dummy node
    first = last = divider
    = new Node(0);
}
```
Lock-free Queue Example: Dequeue

• Usage: one reader, one writer

```cpp
bool dequeue(int& t) {
    if (divider != last) {
        t = divider->next->value;
        divider = divider->next;
        return true;
    }
    return false;
}
```

class Node {
    Node * next;
    int data;
};

// Queue pointers
volatile Node * first;
volatile Node * last;
volatile Node * divider;

lock_free_queue() {
    // add the dummy node
    first = last = divider = new Node(0);
}
```
Lock-free Queue Example: Enqueue

- Usage: one reader, one writer

```cpp
void enqueue(int& t) {
    last->next = new Node(t);
    last = last->next;

    // garbage collect dequeued nodes
    while (first != divider) {
        Node * tmp = first;
        first = first->next;
        delete tmp;
    }
}
```

```cpp
class Node {
    Node * next;
    int data;
};

// Queue pointers
volatile Node * first;
volatile Node * last;
volatile Node * divider;

lock_free_queue() {
    // add the dummy node
    first = last = divider = new Node(0);
}
```
Why Does This Work?

• The enqueue thread and dequeue thread write different pointers
  – Enqueue: last, last->next, first, first->next
  – Dequeue: divider, divider->next
  – Enqueue operations are independent of dequeue operations
  – If these pointers overlap, then no work needs to be done

• The queue always has >1 nodes (starting with the dummy node)
More Advanced Lock-Free Tricks

• Many lock-free data structures can be built using compare and swap (CAS)

```c
bool cas(int * addr, int oldval, int newval) {
    if (*addr == oldval) { *addr = newval; return true; }
    return false;
}
```

• This can be done atomically on x86 using the `cmpxchg` instruction

• Many compilers have built in atomic swap functions
  – GCC: `__sync_bool_compare_and_swap(ptr, oldval, newval)`
  – MSVC: `InterlockedCompareExchange(ptr,oldval,newval)`
Lock-free Stack Example: Push

- Usage: any number of readers and writers

```java
class Node {
    Node * next;
    int data;
};

// Root of the stack
volatile Node * head;

void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node->next, node));
}
```

```
head -> Node 1 -> Node 2 -> Node 3
New Node 1
```
Lock-free Stack Example: Push

• Usage: any number of readers and writers

```c
class Node {
    Node * next;
    int data;
};

// Root of the stack
volatile Node * head;
```

```c
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node->next, node));
}
```

[Diagram of lock-free stack example with threads 1 and 2]
bool pop(int& t) {
    Node* current = head;
    while (current) {
        if (cas(&head, current, current->next)) {
            t = current->data;
            delete current;
            return true;
        }
        current = head;
    }
    return false;
}

class Node {
    Node* next;
    int data;
};

// Root of the stack
volatile Node* head;
Retry Looping is the Key

• Lock free data structures often make use of the retry loop pattern
  1. Read some state
  2. Do a useful operation
  3. Attempt to modify global state if it hasn’t changed (using CAS)

• This is similar to a spinlock
  – But, the assumption is that wait times will be small
  – However, retry loops may introduce starvation

• Wait-free data structures remove retry loops
  – But are much more complicated to implement
Many Reads, Few Writes

• Suppose we have a map (hashtable) that is:
  – Constantly read by many threads
  – Rarely, but occasionally written
• How can we make this structure lock free?

```cpp
class readmap {
    mutex mtx;
    map<string, string> map;

    string lookup(const string& k) {
        lock l(mtx);
        return map[k];
    }

    void update(const string& k, const string& v) {
        lock lock(mtx);
        map[k] = v;
    }
};
```
Dupeicate and Swap

```cpp
class readmap {
    map<string, string> * map;

    readmap() { map = new map<string, string>(); }

    string lookup(const string& k) {
        return (*map)[k];
    }

    void update(const string& k, const string& v) {
        map<string, string> * new_map = 0;
        do {
            map<string, string> * old_map = map;
            if (new_map) delete new_map;
            // clone the existing map data
            new_map = new map<string, string>(*old_map);
            (*new_map)[k] = v;
            // swap the old map for the new, updated map!
        } while (cas(&map, old_map, new_map));
    }
};
```
Memory Problems

• What is the problem with the previous code?

```c
} while (cas(&map, old_map, new_map));
```

• The old map is not deleted (memory leak)

• Does this fix things?

```c
} while (cas(&map, old_map, new_map));
delete old_map;
```

• Readers may still be accessing the old map!
  – Deleting it will cause nondeterministic behavior

• Possible solution: store the old_map pointer, delete it after some time has gone by
Hazard Pointers

• Construct for managing memory in lock-free data structures

• Straightforward concept:
  – Read threads publish hazard pointers that point to any data they are currently reading
  – When a write thread wants to delete data:
    • If it is not associated with any hazard pointers, delete it
    • If it is associated with a hazard pointer, add it to a list
    • Periodically go through the list and reevaluate the data

• Of course, this is tricky in practice
  – You need lock-free structures to:
    • Enable publishing/updating hazard pointers
    • Store the list of data blocked by hazards
The ABA Problem

• Subtle problem that impacts many lock-free algorithms
• Compare and swap relies on the uniqueness of pointers
  – Example: cas(&head, current, current->next)
• However, sometimes the memory manager will reuse pointers

```c
item * a = stack.pop();
free a;
item * b = new item();
stack.push(b);
assert(a != b); // this assertion may fail!
```
bool pop(int& t) {
    Node* current = head;
    while (current) {
        if (cas(&head, current, current->next)) {
            t = current->data;
            delete current;
            return true;
        }
        current = head;
    }
    return false;
}
Applications of Lock-Free Structures

- Stack
- Queue
- Deque
- Linked list
- Doubly linked list
- Hash table
- Many variations on each
  - Lock free vs. wait free

- Memory managers
  - Lock free malloc() and free()

- The Linux kernel
  - Many key structures are lock-free
References

• Geoff Langdale, Lock-free Programming
  – http://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf

• Herb Sutter, Writing Lock-Free Code: A Corrected Queue