Multi-Object Synchronization
Multi-Object Programs

• What happens when we try to synchronize across multiple objects in a large program?
  – Each object with its own lock, condition variables
  – Is locking modular?

• Performance
• Semantics/correctness
• Deadlock
• Eliminating locks
Synchronization Performance

• Speedup = Time on one CPU/Time on N CPUs
• A program with lots of concurrent threads can have poor speedup, because:
  – Lock contention: only one thread at a time holding a given lock
  – Shared data protected by a lock may ping back and forth between cores
  – False sharing: communication between cores even for data that is not shared
Question

• Suppose a critical section is 5% of the total work per request (on a single CPU)
• What is maximum possible speedup from running each request in its own thread?

• Suppose CPU is 5x slower when executing in critical section due to cache effects
• What is maximum possible speedup?
A Simple Test of Cache Behavior

• An array of locks, each protects a counter
  – Critical section: increment counter
  – Multiple cores
• Test 1: one thread loops over array
• Test 2: two threads loop over different arrays
• Test 3: two threads loop over single array
• Test 4: two threads loop over alternate elements in single array
<table>
<thead>
<tr>
<th>Configuration</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>One thread, one array</td>
<td>51</td>
</tr>
<tr>
<td>Two threads, two arrays</td>
<td>52</td>
</tr>
<tr>
<td>Two threads, one array</td>
<td>197</td>
</tr>
<tr>
<td>Two threads, odd/even</td>
<td>127</td>
</tr>
</tbody>
</table>
Reducing Lock Contention

• Fine-grained locking
  – Partition object into subsets, each protected by its own lock
  – Example: hash table buckets

• Per-processor data structures
  – Partition object so that most/all accesses are made by one processor
  – Example: per-processor heap

• Ownership
  – Check out item, modify, check back in

• Staged architecture
  – Divide and conquer
What If Locks are Still Busy?

• MCS Locks
  – Optimize lock implementation for when lock is contended

• RCU (read-copy-update)
  – Efficient readers/writers lock used in Linux kernel
  – Readers proceed without first acquiring lock
  – Writer ensures that readers are done
Read-Copy-Update

• Restricted update
  – Writer computes new version of data structure
  – Publishes new version with a single atomic instruction

• Multiple concurrent versions
  – Readers may see old or new version

• Integration with thread scheduler
  – Guarantee all readers complete within grace period, and then garbage collect old version
  – OK if write is slow
Read-Copy-Update

Update is Published

Read (Old)

Read (Old)

Read (Old or New)

Write (New)

Grace Period

Grace Period Ends

Read (New)

Read (New)

Read (Old or New)

Delete (Old)

Time
Read-Copy-Update Implementation

• Readers disable interrupts on entry
  – Guarantees they complete critical section in a timely fashion
  – No read or write lock

• Writer
  – Acquire write lock
  – Compute new data structure
  – Publish new version with atomic instruction
  – Release write lock
  – Wait for time slice on each CPU
  – OK to garbage collect
Deadlock Definition

• Resource: any (passive) thing needed by a thread to do its job (CPU, disk space, memory, lock)
  – Preemptable: can be taken away by OS
  – Non-preemptable: must leave with thread

• Starvation: thread waits indefinitely

• Deadlock: circular waiting for resources
  – Deadlock => starvation, but not vice versa
Example: two locks

Thread A

lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();

Thread B

lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
## Bidirectional Bounded Buffer

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buffer1.put(data);</code></td>
<td><code>buffer2.put(data);</code></td>
</tr>
<tr>
<td><code>buffer1.put(data);</code></td>
<td><code>buffer2.put(data);</code></td>
</tr>
<tr>
<td><code>buffer2.get();</code></td>
<td><code>buffer1.get();</code></td>
</tr>
<tr>
<td><code>buffer2.get();</code></td>
<td><code>buffer1.get();</code></td>
</tr>
</tbody>
</table>
## Two locks and a condition variable

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock1.acquire();</td>
<td>lock1.acquire();</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>lock2.acquire();</td>
<td>lock2.acquire();</td>
</tr>
<tr>
<td>while (need to wait)</td>
<td>....</td>
</tr>
<tr>
<td>condition.wait(lock2);</td>
<td>condition.signal(lock2);</td>
</tr>
<tr>
<td>lock2.release();</td>
<td>lock2.release();</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>lock1.release();</td>
<td>lock1.release();</td>
</tr>
</tbody>
</table>
Yet another Example
Dining Lawyers

Each lawyer needs two chopsticks to eat. Each grabs chopstick on the right first.
Necessary Conditions for Deadlock

• Limited access to resources
  – If infinite resources, no deadlock!

• No preemption
  – If resources are virtual, can break deadlock

• Multiple independent requests
  – “wait while holding”

• Circular chain of requests
Question

• How does Dining Lawyers meet the necessary conditions for deadlock?
  – Limited access to resources
  – No preemption
  – Multiple independent requests (wait while holding)
  – Circular chain of requests

• How can we modify system to prevent deadlock?
Example

Thread 1

1. Acquire A
2.
3. Acquire C
4.
5. Wait for B

Thread 2

1.
2. Acquire B
3.
4. Wait for A

How could we have avoided deadlock?
Preventing Deadlock

• Exploit or limit program behavior
  – Limit program from doing anything that might lead to deadlock

• Predict the future
  – If we know what program will do, we can tell if granting a resource might lead to deadlock

• Detect and recover
  – If we can rollback a thread, we can fix a deadlock once it occurs
Exploit or Limit Behavior

• Provide enough resources
  – How many chopsticks are enough?

• Eliminate wait while holding
  – Release lock when calling out of module
  – Telephone circuit setup

• Eliminate circular waiting
  – Lock ordering: always acquire locks in a fixed order
  – Example: move file from one directory to another
Predict the Future

• Banker’s algorithm
  – State maximum resource needs in advance
  – Allocate resources dynamically when resource is needed -- wait if granting request would lead to deadlock
  – Request can be granted if some sequential ordering of threads is deadlock free
Possible System States

Unsafe

Safe

Deadlock
Definitions

• Safe state:
  – For any possible sequence of future resource requests, it is possible to eventually grant all requests
  – May require waiting even when resources are available!

• Unsafe state:
  – Some sequence of resource requests can result in deadlock

• Doomed state:
  – All possible computations lead to deadlock
Banker’s Algorithm

• Grant request iff result is a safe state
• Sum of maximum resource needs of current threads can be greater than the total resources
  – Provided there is some way for all the threads to finish without getting into deadlock
• Example: proceed iff
  – total available resources - # allocated >= max remaining that might be needed by this thread in order to finish
  – Guarantees this thread can finish
Example: Banker’s Algorithm

• n chopsticks in middle of table
• n lawyers, each can take one chopstick at a time
• When is it ok for lawyer to take a chopstick?
• What if each lawyer needs k chopsticks?
Detect and Repair

• Algorithm
  – Scan wait for graph
  – Detect cycles
  – Fix cycles

• Proceed without the resource
  – Requires robust exception handling code

• Roll back and retry
  – Transaction: all operations are provisional until have all required resources to complete operation
Detecting Deadlock

- Thread 0 is waiting for Resource X owned by Thread 1.
- Thread 1 is waiting for Resource Y owned by Thread 0.
- Thread 0 is waiting for Resource Y owned by Thread 1.
- Thread 1 is waiting for Resource X owned by Thread 0.

Resource X and Resource Y are involved in a deadlock scenario where each resource is held by a different thread, leading to an endless waiting cycle.
Non-Blocking Synchronization

• Compare and swap atomic instruction
  – Create copy of data structure
  – Modify copy
  – Swap in new version iff no one else has
  – Restart if pointer has changed
Lock-Free Bounded Buffer

tryget() {
    do {
        copy = ConsistentCopy(p);
        if (copy->front == copy->tail)
            return NULL;
        else {
            item = copy->buf[copy->front % MAX];
            copy->front++;
        }
    } while (((compare&swap(copy, p) != p));
    return item
}