Optimistic Synchronization
in parallel systems

Anders Gidenstam
(andersg@cs.chalmers.se)
synchronization  
1: the relation that exists when things occur at the same time;  
2: an adjustment that causes something to occur or recur in unison  
3: coordinating by causing to indicate the same time; "the synchronization of their watches was an important preliminary"  
Synchronization

- Shared data structures need synchronization

- Synchronization using Locks
  - Mutually exclusive access to whole or parts of the data structure
Shared memory
Multiprocessor Systems

- Uniform Memory Access (UMA)

- Non-Uniform Memory Access (NUMA)
Blocking synchronization

- Mutual exclusion locks
  - Traditional solution
    - Semaphores, spin-locks, disabling interrupts
    - Protects a critical section
  - Drawbacks
    - Blocking
    - Lock convoys
    - Priority inversion
    - Risk of dead-lock
    - Limits parallelism
Hardware support for synchronization

- Synchronization primitives
  - Built into CPU and memory system
  - Atomic (i.e. a critical section of one instruction)

- Examples
  - Test-and-set
  - Compare-and-Swap

```c
bool compare_and_swap(int *target, int old, int new) atomic {
  if (*target = old) {
    *target = new;
    return TRUE;
  }
  return FALSE;
}
```
Non-blocking synchronization

- Lock-Free or Optimistic synchronization
  - Try to do the operation as if there were no interference
    1. Prepare update of shared data
    2. Commit using atomic synchronization primitives
    3. Retry if interfered with
  - At least one concurrent operation always makes progress
- Benefits
  - Fast on average
- Drawbacks
  - Operations can starve
Non-blocking synchronization

- Wait-Free synchronization
  - All operations finish in a finite number of their own steps
  - Benefits
    - Bounded execution times
    - Attractive for real-time systems (WCET known, no blocking)
  - Drawbacks
    - Algorithms and implementations usually complex
    - Average performance may be worse than lock-free
Concurrent applications

<table>
<thead>
<tr>
<th>#CPUs</th>
<th>#Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Traditional desktop applications</td>
</tr>
<tr>
<td>5</td>
<td>Traditional multi-threaded desktop applications</td>
</tr>
<tr>
<td></td>
<td>Multi-threaded applications on new multicore CPU(s)</td>
</tr>
<tr>
<td></td>
<td>High performance multi-threaded applications on multiprocessors</td>
</tr>
</tbody>
</table>
Example: Counting (I)

```c
volatile int shared_counter = 0;
void count_thread() {
    for (int j = 0; j < MAX; j++) {
        shared_counter = shared_counter + 1;
    }
}
```

Thread A

Read shared_counter -> regX

regX = regX + 1

Write regX to shared_counter

Thread B

Read shared_counter -> regX

regX = regX + 1

Write regX to shared_counter

shared_counter = 4

shared_counter = ?
Example: Counting (II)

```c
volatile int shared_counter = 0;     mutex_t mutex;
void count_thread() {
    for (int j = 0; j < MAX; j++) {
        lock(mutex);
        shared_counter = shared_counter + 1;
        unlock(mutex)
    }
}
```

Thread A

Lock mutex
Read shared_counter -> regX
regX = regX + 1
Write regX to shared_counter
Unlock mutex

Thread B

shared_counter = 4
Lock mutex
Read shared_counter -> regX
regX = regX + 1
Write regX to shared_counter
Unlock mutex

shared_counter = 6
Example: Counting (III)

```c
volatile int shared_counter = 0;
void count_thread() {
    for (int j = 0; j < MAX; j++) {
        repeat {
            int old = shared_counter;
            int new = old + 1;
        } until CAS(&shared_counter, old, new)
    }
}

Thread A
shared_counter = 4
Read shared_counter -> regX
regY = regX + 1
CAS(shared_counter, regX, regY) -> True
shared_counter = 5

Thread B
Read shared_counter -> regX
regY = regX + 1
CAS(shared_counter, regX, regY) -> false
Thread B has to retry...
```
Work in progress

- Combining lock-free operations and structures

- Case study: Lock-free memory allocator

  “Remove + Insert” is not atomic. An item may get stuck in “limbo”.
Moving a shared pointer

- **Goal:**
  - Move a pointer value between two shared pointer locations

- **Requirements**
  - The pointer target must stay accessible
  - The same # of shared pointers to the target after the move as before
  - Lock-free behaviour

- **Issues**
  - One atomic CAS is not enough! We’ll need several steps.
  - Interfering threads need to *help* unfinished operations
Moving a shared pointer

From New_pos
Old_pos
To

Note that some tricky details are needed to prevent ABA problems..
Summary

- Non-blocking synchronization
  - Can offer increased performance
  - Avoids
    - Blocking
    - Deadlock
    - Priority inversion
Questions?

- Contact Information:
  - Address:
    Anders Gidenstam
    Computing Science
    Chalmers University of Technology
  - Email:
    andersg @ cs.chalmers.se
  - Web:
    http://www.cs.chalmers.se/~andersg
    http://www.cs.chalmers.se/~dcs