

Non-Blocking Synchronization

Håkan Sundell



Multicore: Important Issues

- Software Scalability
 - More logical CPU's increase overall performance?!
- Software Reliability
 - Correctness?
 - Fault-tolerance?
 - Predictable?

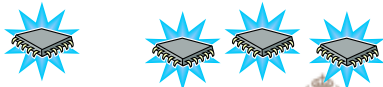


(Parallel) Software

- Programs consist of many tasks (threads)

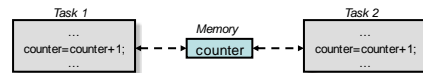


- That execute on one or more (logical) processors



Communication

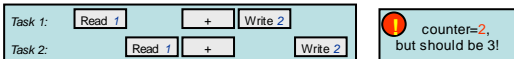
- Tasks need to communicate and work with shared resources.
 - Message Passing
 - High overhead and Abstract system design.
 - Shared Memory
 - Fast and Intuitive



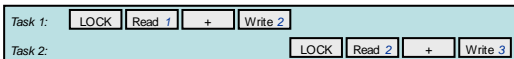
Critical Sections

- Problem: operations on shared variables in programming languages are not atomic.

counter=counter+1; = Read + Write



- Straightforward solution: Apply mutual exclusion



Scheduling

- Problem: We have many tasks and few processors.
 - Each task might have deadlines
 - Each task might have different importance (priorities)
- Solution: Tasks must be scheduled to alternately use the processors.
 - Scheduling algorithm (RM, EDF etc.)
 - Worst case execution time analysis
 - Schedulability analysis (i.e. deadlines must be met).



Multiprocessors

- Needs for performance are increasing
 - Single processor speed has reached its limit (due to power/heat)
 - Single processor in embedded systems might have even lower speed due to environmental constraints (temperature range, humidity etc.)
- We need multiple processors to increase performance!
- Multi-Core!



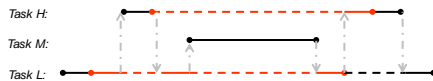
Solutions are not compositional!

- Critical Sections + Scheduling
 - Blocking.** More advanced and pessimistic schedulability analysis.
 - Deadlocks.** Reduced fault-tolerance, if one task fails, other (even all) might also fail.
 - Priority Inversion.** Tasks might not execute with the proper priority even though it was set. Deadlines might be missed.



Priority Inversion

- A high priority task is delayed due to a low priority task holding a shared resource. The low priority task is delayed due to a medium priority task executing.

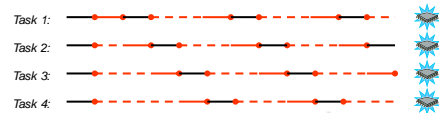


- Solutions: Priority inheritance protocols
 - Works ok for single processors, but definitely not for multiple processors!



Critical Sections + Multiprocessors

- Reduced Parallelism.** Several tasks with overlapping critical sections will cause waiting processors to go idle.



Problems: Summary

- Software Systems need:
 - Shared resources and Communication.
 - Dependency/Stability and Fault-tolerance.
 - Increased Performance and Utilization.
- Current solutions simply do not fit!
 - Communication can be a bottleneck as well as a source for difficult problems.



Communication+Scheduling+Uni/Multiprocessors: New Solution

- Avoid Critical Sections!**
 - Avoid Blocking.** Easier and more optimistic analysis, i.e. less hardware needed.
 - Avoid Deadlocks.** Increased fault-tolerance as failed tasks can not affect others to fail.
 - Avoid Priority Inversion.** Easier and more reliable analysis, and avoids complex and high-overhead solutions.
 - Increased Parallelism.** Increased overall performance, more optimistic analysis, i.e. less hardware needed.



Non-Blocking Synchronization

- The key lies in how mutual exclusion (i.e. mutex, semaphore) is implemented in actual hardware (i.e. processors).
 - Atomic primitives in hardware can atomically update one memory word.
- Sophisticated solutions can exploit the same atomic primitives to support access to shared resources without locks, i.e. non-blocking.



Non-Blocking Algorithms

- **Obstruction-Free.**
 - Guarantees progress in absence of contention.
 - Need extra module for contention management.
- **Lock-Free.**
 - Guarantees that always one operation is making progress.
 - Combined with scheduling information, schedulability analysis can be done.
- **Wait-Free.**
 - Guarantees that any operation will finish in a finite time.
 - Schedulability analysis can be done directly.



Non-Blocking Algorithms

- The counter-problem can obviously be solved with atomic primitives.
 - Supported for example by Win32 API (Interlocked operations)
 - Supported atomic primitives are read-modify-write instructions, i.e. one-word transactions.
- However, programmers need to share more complex structures of data!
 - Can also more advanced data structures be constructed non-blocking?



Applications and Performance

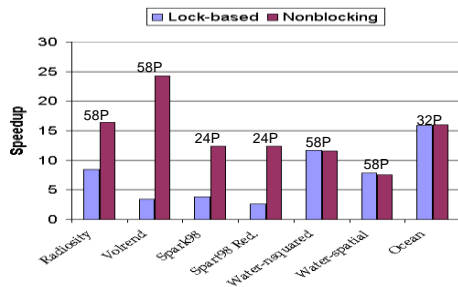
o Tsigas and Zhang 2001

Ocean	simulates eddy currents in an ocean basin.
Radiosity	computes the equilibrium distribution of light in a scene using the radiosity method.
Volrend	renders 3D volume data into an image using a ray-casting method.
Water	Evaluates forces and potentials that occur over time between water molecules.
Spark98	a collection of sparse matrix kernels. Each kernel performs a sequence of sparse matrix vector product operations using matrices that are derived from a family of three-dimensional finite element earthquake applications.

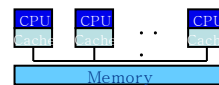
16



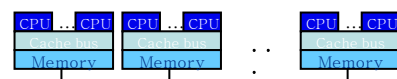
Tsigas and Zhang 2001



Shared Memory



– Uniform Memory Access (UMA)



– Non-Uniform Memory Access (NUMA)



Hardware Synchronization Primitives

- Weak
 - Atomic Read/Write
- Stronger
 - Atomic Test-And-Set (TAS), Fetch-And-Add (FAA), Swap
- Universal
 - Atomic Compare-And-Swap (CAS)
 - Atomic Load-Linked/Store-Conditionally



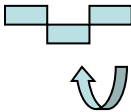
Universal and Conditional Synchronization primitive

- Compare-And-Swap (CAS)

```
bool CAS(int *p, int old, int new) {
    atomic {
        if(*p == old) {
            *p=new;
            return true;
        }
        else return false;
    }
}
```

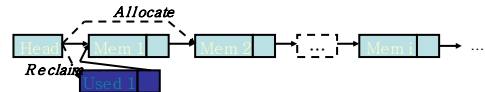
Non-blocking Synchronization

- Lock-Free Synchronization
 - Optimistic approach (i.e. assumes no interference)
 - 1. The operation is prepared to later take effect (unless interfered) using hardware atomic primitives
 - 2. Possible interference is detected via the atomic primitives, and causes a retry
 - Can cause starvation
- Wait-Free Synchronization
 - Always finishes in a finite number of its own steps.



Non-Blocking Synchronization: Example

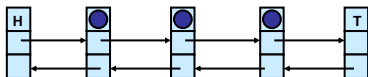
- Lock-Free Stack (i.e. Free-list):
 - Create a linked-list of the free nodes, allocate/reclaim using CAS



- How to make sure that the next-pointer of the first item is not changed before CAS?
 - The ABA problem

Non-Blocking Synchronization: Example

- Lock-Free Doubly Linked Lists



- How to make sure concurrent inserts / deletes and right / left traversals work ok?
 - Update next (and prev) pointer using CAS?

Non-Blocking Synchronization: Problems

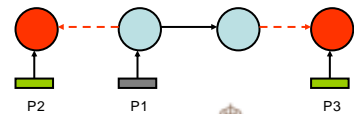
- Algorithmic design
 - Operations on shared data structures usually involve updates of several shared variables
 - Modern shared memory systems only support atomic primitives on *single* memory words!
 - Want parallelism – Avoid bottlenecks
 - All sub-operations also have to be lock-free/wait-free

Memory Reclamation / Allocation

- Dynamic data structures
 - Need dynamic memory allocation (malloc/free).
- Shared data structures and dynamic memory.
 - Needs to be able to safely access data, while other threads might want to "free" data!
- Needs Memory reclamation scheme / Garbage collection
 - Should be Lock-Free / Wait-Free (as function calls will be a sub-operation of lock-free/wait-free algorithm)!
 - System malloc/free is not lock-free/wait-free.
- Needs lock-free/wait-free memory reclamation/allocation schemes!

Concurrent Memory Management

- Concurrent Memory Allocation
 - i.e. malloc/free functionality
- Concurrent Memory Reclamation
 - Questions (among many):
 - When to re-use memory?
 - How to de-reference pointers safely?

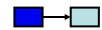


Lock-Free Memory Management

- Memory Allocation
 - Valois 1995: fixed block-size, fixed purpose
 - Michael 2004: Gidenstam et al. 2004, any size, any purpose
- Garbage Collection
 - Valois 1995, Detlefs et al. 2001: reference counting
 - Michael 2002, Herlihy et al. 2002: hazard pointers
 - Gidenstam, Papatriantafyllou, Sundell and Tsigas 2005: hazard pointer + reference counting

Lock-Free Reference Counting

- De-referencing links
 - 1. Read the link contents, i.e. a pointer.
 - 2. Increment (FAA) the reference count on the corresponding object.
- What if the link is changed between step 1 and 2?
 - Solution by Detlefs et al:
 - Use DCAS on step 2 that operates on two arbitrary memory words. Retries if link is changed after step 2.
 - Solution by Valois et al:
 - The reference count field is present indefinitely. Decrement reference count and retries if link is changed after step 2.



Lock-Free Hazard Pointers (Michael 2002)

- De-referencing links
 - 1. Read the link contents, i.e. a pointer.
 - 2. Set a hazard pointer to the read pointer value.
 - 3. Read the link contents again; if not same as in step 1 then restart from step 1.
- Deletion
 - After deleted from data structure, put node on a local list.
 - When the local list reaches a certain size; scan all hazard pointers globally, reclaim memory of all nodes which address does not match the scan.

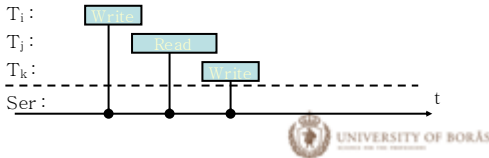


Correctness

- Linearizability (Herlihy 1991)
 - In order for an implementation to be *linearizable*, for every concurrent execution, there should exist an *equal sequential execution* that respects the *partial order* of the operations in the concurrent execution

Linearizability

- All concurrent executions can be transformed into an equivalent serial sequence of atomic operations preserving the partial order



Linearizability

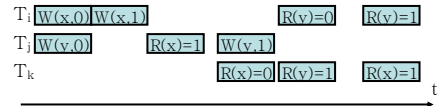
- Proofs
 - Series of intuitive lemmas
 - Formal proofs using semi-automatic proof-engines requiring hundreds of invariants and several man-years of work.

Correctness Proofs

- Define precise sequential semantics
- Define abstract state and its interpretation
 - Show that state is atomically updated
- Define linearizability points
 - Show that operations take effect atomically at these points with respect to sequential semantics
- Creates a total order using the linearizability points that respects the partial order
 - The algorithm is linearizable

Memory Consistency

- Example scenario

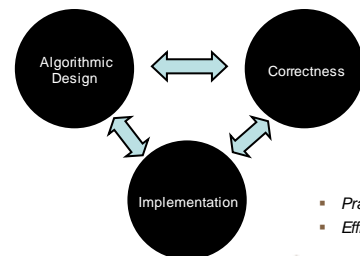


- Models: Relaxed Memory Order, Sequential Consistency, etc.

Non-Blocking Synchronization: Problems

- Implementation
 - Modern shared memory systems do not offer sequential or equivalent level of consistency by default
 - Out-of-order execution
 - Need to specify required read/write \leftrightarrow read/write relative order for each memory access as needed
 - Extra instructions inserted
 - Degrades out-of-order execution, i.e. significantly degrades speed!

Non-Blocking Synchronization: Problems



- Practical?
- Efficient?

Non-Blocking Synchronization: Details

- Memory Management
 - Memory allocation
 - Memory reclamation (garbage collection)
- Atomic primitives
- Common shared data structures
 - Stack
 - Queue
 - Deque
 - Priority Queue
 - Dictionary
 - Hash Table
 - Linked Lists



Lock-Free Memory Management

- Memory Allocation
 - Valois 1995, fixed block-size, fixed purpose
 - Michael 2004, Gidenstam et al. 2004, any size, any purpose
- Garbage Collection
 - Valois 1995, (Detlefs et al. 2001); reference counting
 - Michael 2002, (Herlihy et al. 2002); hazard pointers
 - Gidenstam, Papatriantafidou, Sundell and Tsigas 2005, "Efficient and Reliable Memory Reclamation Based on Reference Counting"






Wait-Free Memory Management

- Hesselink and Groote 2001. Limited to shared tokens.
- Sundell 2005. "Wait-Free Reference Counting and Memory Management"
 - Memory Allocation – fixed block-size, fixed purpose
 - Garbage Collection – reference counting



Software Synchronization Primitives

- Atomic Read/Write.
 - Several results published WF/LF. 
- Multi-variable Read/Write, i.e. Snapshot.
 - Several results published WF/LF.
- LL/SC.
 - Several results published WF/LF. 
- Multi-word Compare-And-Swap (CASN) i.e. transactions
 - Several results published WF/LF. 



Shared Data Structures

- Lock-Free
 - Stack
 - Valois 1995, Michael 2002
 - Queue
 - Valois 1995, Tsigas and Zhang 2001, Michael 2002 and much more
 - Deque
 - Michael 2003, Sundell and Tsigas 2004



Shared Data Structures

- Lock-Free
 - Priority Queue
 - Sundell and Tsigas 2003, 2005
 - Dictionary (and Set)
 - Linked List. (Harris et al. 2001) ,
 - Skip List. Sundell and Tsigas 2003
 - Hash Tables. Michael 2002 and much more
 - Linked Lists
 - Singly-Linked: Valois 1995, (Harris et al. 2001)
 - Doubly-Linked: Sundell and Tsigas 2004, 2008



Non-Blocking Synchronization: Open

- Wait-Free Memory Management.
 - Improvement
- Atomic primitives.
 - Wait-Free multi-word compare-and-swap
- Lock-Free Data Structures.
 - Tree
 - Improvement, disjoint-access-parallelism
- Wait-Free Data Structures.
 - Stack, queue, priority queue, dictionary, hash table, linked lists, trees

