Dekker's Algorithm and Weakly-Ordered

Problem: Dekker's algorithm requires sequentially consistency.
Idea: insert memory barriers between all variables common to both threads.

P0:
flag[0] = true;
sfence();
while (lfence(), flag[1] == true)
  if (lfence(), turn != 0) {
    flag[0] = false;
sfence();
    while (lfence(), turn != 0) {
      // busy wait
    }
    flag[0] = true;
sfence();
  }
  // critical section
  turn = 1;
sfence();
flag[0] = false;

Discussion
Memory barriers lie at the lowest level of synchronization primitives.

Discussion
Memory barriers lie at the lowest level of synchronization primitives.
Where are they useful?
- when several processes implement an automaton and ...
Discussion
Memory barriers lie at the lowest level of synchronization primitives. Where are they useful?
- when several processes implement an automaton and . . .
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems
Discussion

Memory barriers lie at the lowest level of synchronization primitives. Where are they useful?

- when several processes implement an automaton and . . .
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems

Why might they not be appropriate?

- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck
Discussion

Memory barriers lie at the lowest level of synchronization primitives. Where are they useful?
- when several processes implement an automaton and...
- synchronization means coordinating transitions of these automata
- when blocking should not de-schedule threads
- often used in operating systems

Why might they not be appropriate?
- difficult to get right, possibly inappropriate except for specific, proven algorithms
- often synchronization with locks is as fast and easier
- too many fences are costly if store/invalidate buffers are bottleneck

What do compilers do about barriers?
- C/C++: it's up to the programmer, use volatile for all thread-common variables to avoid optimization that are only correct for sequential programs
- C++11: use of atomic variables will insert memory barriers
- Java, Go, ...: the runtime system must guarantee this
Future Many-Core Systems: NUMA

Symmetric multi-processing (SMP) has its limits:

- a memory-intensive computation may cause contention on the bus
- the speed of the bus is limited since the electrical signal has to travel to all participants
- point-to-point connections are faster than a bus, but do not provide possibility of forming consensus
Overhead of NUMA Systems

Communication overhead in a NUMA system.

- Processors in a NUMA system may be fully or partially connected.
- The directory of who stores an address is partitioned amongst processors.

A cache miss that cannot be satisfied by the local memory at A:

- A sends a retrieve request to processor B owning the directory
- B tells the processor C who holds the content
- C sends data (or status) to A and sends acknowledgment to B
- B completes transmission by an acknowledge to A

source: [Intel, 2008]

References

Intel.
An introduction to the intel quickpath interconnect.

L. Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.

P. E. McKenny.
Memory Barriers: a Hardware View for Software Hackers.
Why Memory Barriers are not Enough

Communication via memory barriers has only specific applications:
- coordinating state transitions between threads
- for systems that require minimal overhead (and no de-scheduling)
Often certain pieces of memory may only be modified by one thread at once.
- can use barriers to implement automata that ensure mutual exclusion
- generalize the re-occurring concept of enforcing mutual exclusion

Why Memory Barriers are not Enough

Communication via memory barriers has only specific applications:
- coordinating state transitions between threads
- for systems that require minimal overhead (and no de-scheduling)
Often certain pieces of memory may only be modified by one thread at once.
- can use barriers to implement automata that ensure mutual exclusion
- generalize the re-occurring concept of enforcing mutual exclusion

Atomic Executions

A concurrent program consists of several threads that share common resources:
- resources are often pieces of memory, but may be an I/O entity

---

a=1, b=1

several values of the objects are used to compute new value

certain information form the thread flows into this computation

certain information flows from the computation to the thread
Atomic Executions

A concurrent program consists of several threads that share common resources:
- resources are often pieces of memory, but may be an I/O entity
  - a file can be modified through a shared handle
- for each resource an invariant must be retained
  - a head and tail pointer must define a linked list
- during an update, an invariant may be broken
- an invariant may span several resources

Atomic Executions

A concurrent program consists of several threads that share common resources:
- resources are often pieces of memory, but may be an I/O entity
  - a file can be modified through a shared handle
- for each resource an invariant must be retained
  - a head and tail pointer must define a linked list
- during an update, an invariant may be broken
- an invariant may span several resources
  - several resources must be updated together to ensure the invariant
Overview

We will address the *established* ways of managing synchronization.

- present techniques are available on most platforms
- likely to be found in most existing (concurrent) software
- techniques provide solutions to solve common concurrency tasks
- techniques are the source of common concurrency problems

Presented techniques applicable to C, C++ (pthread), Java, C# and other imperative languages.
Overview

We will address the established ways of managing synchronization. Present techniques are available on most platforms, likely to be found in most existing (concurrent) software, techniques provide solutions to solve common concurrency tasks, and techniques are the source of common concurrency problems.

Presented techniques applicable to C, C++ (pthread), Java, C# and other imperative languages.

Learning Outcomes

- Principle of Atomic Executions
- Wait-Free Algorithms based on Atomic Operations
- Locks: Mutex, Semaphore, and Monitor
- Deadlocks: Concept and Prevention

Atomic Execution: Varieties

Definition (Atomic Execution)

A computation forms an atomic execution if its effect can only be observed as a single transformation on the memory.

Several classes of atomic executions exist:

- **Wait-Free**: an atomic execution always succeeds and never blocks
- **Lock-Free**: an atomic execution may fail but never blocks
- **Locked**: an atomic execution always succeeds but may block the thread
- **Transaction**: an atomic execution may fail (and may implement recovery)

These classes differ in:

- **amount of data** they can access during an atomic execution
- **expressivity** of operations they allow
- **granularity** of objects in memory they require
### Wait-Free Updates

Which operations on a CPU are atomic executions?

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
<th>Program 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i++;</code></td>
<td><code>j = i;</code></td>
<td><code>int tmp = i;</code></td>
</tr>
<tr>
<td></td>
<td><code>i = i + k;</code></td>
<td><code>i = j;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>j = tmp;</code></td>
</tr>
</tbody>
</table>

Answer:
- none by default (even without store and invalidate buffers, *why*)
- but all of them *can* be atomic executions

The programs can be atomic executions:

<table>
<thead>
<tr>
<th>Program 1</th>
<th>Program 2</th>
<th>Program 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i++;</code></td>
<td><code>j = i;</code></td>
<td><code>int tmp = i;</code></td>
</tr>
<tr>
<td></td>
<td><code>i = i + k;</code></td>
<td><code>i = j;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>j = tmp;</code></td>
</tr>
</tbody>
</table>

Answer:
- none by default (even without store and invalidate buffers, *why*)
- but all of them *can* be atomic executions

The programs can be atomic executions:
- *i* must be in memory (e.g. declare as volatile)
- most CPUs can lock the cache for the duration of an instruction; on x86:
Wait-Free Updates
Which operations on a CPU are atomic executions?

Program 1
i++;

Program 2
j = i;
i = i+k;

Program 3
int tmp = i;
i = j;
j = tmp;

Answer:
- none by default (even without store and invalidate buffers, *why?*)
- but all of them *can* be atomic executions

The programs can be atomic executions:
- *i* must be in memory (e.g. declare as volatile)
- most CPUs can *lock* the cache for the duration of an instruction; on x86:
  - Program 1 can be implemented using a *lock inc [addr.i]* instruction
  - Program 2 can be implemented using *mov eax,k*
    *lock xadd [addr.i],eax; mov [addr.j],eax*
  - Program 3 can be implemented using *lock xchg [addr.i],[addr.j]*

⚠️ Without *lock*, the load and store generated by *i++* may be interleaved with a store from another processor.
Wait-Free Bumper-Pointer Allocation

Garbage collectors often use a bumper pointer to allocated memory:

```c
char heap[2^20];
char* firstFree = &heap[0];

char* alloc(int size) {
    char* start = firstFree;
    firstFree = firstFree + size;
    if (start+size>sizeof(heap)) garbage_collect();
    return start;
}
```

- `firstFree` points to the first unused byte
- each allocation reserves the next `size` bytes in `heap`

Thread-safe implementation:
- the `alloc` function can be used from multiple threads when implemented using a `lock xadd [firstFree],eax` instruction
- `~` requires inline assembler