

Atomic Sections for Modern Languages: Improving the Reliability and Security of Concurrent Software

Michael Ringenburg
Dan Grossman

Concurrency is Dangerous

- Concurrency is a common source of software errors.
- These errors can result in buggy software and security vulnerabilities.
- As of 10/20/04:
 - Searching for “race condition” on securityfocus.com yielded 951 hits.
 - Searching the US CERT website generated 451 hits for “race condition”.

... but Important!

- A widely used programming idiom for decades
- Allows the programmer to hide latency and unresponsiveness
 - I/O heavy applications
 - GUIs

Why is Concurrency Hard?

- Threads may share resources, e.g. memory.
- Multiple threads may access the same resource in a non-deterministic order.
 - Thread A writes a memory location, and then thread B reads the same location.
 - Or thread B reads before thread A writes.
- This may lead to unpredictable behavior.

Our Solution

- Most systems use locks to synchronize accesses to shared memory.
 - Only one thread may hold a given lock.
 - Other threads that try to acquire the lock must wait until the holding thread releases it.
- We propose replacing locks with atomic.
 - Programmer marks code blocks or functions with `atomic`.
 - Language implementation provides a strong guarantee that the marked block appears to execute “all-at-once”, from the perspective of other threads.

We Claim ...

- Atomic is a better shared-memory synchronization primitive
 - Strong guarantees
 - Easy to use
- We can efficiently implement atomic for a wide class of systems
 - We don't need to use locks or halt other threads
 - We don't need to disable interrupts or prohibit function calls

Why Atomic is Better, Part I

- When the programmer marks a block with `atomic`, the implementation *guarantees* that it will appear to execute atomically.
- In particular, unlike locks, `atomic`:
 - Does not have to rely on other code acquiring locks to prevent race conditions.
 - Does not have to rely on other code releasing locks to prevent deadlock.

Why Atomic is Better, Part II

- It's often exactly what programmers are trying to accomplish with locks.
- Locks and `atomic` can coexist. We can even implement locks with `atomic`.
- Atomic is easy to use:

```
void deposit(int x){  
    synchronized(this){  
        balance += x;  
    }  
}
```

```
void deposit(int x){  
    atomic {  
        balance += x;  
    }  
}
```

Atomic Implementation

- Our first and most efficient approach works on all systems where threads sharing memory do not exhibit true parallelism.
 - Uniprocessor systems
 - Systems where all threads of a process run on the same processor, and shared memory is not used for interprocess communication.
 - e.g., Microsoft Research's Singularity OS project
 - O'Caml and DrScheme thread libraries
- Developing approaches for other systems.
- Targets those applications that benefit from the latency-hiding provided by concurrency.

The idea

- An optimistic approach: we assume most atomic blocks can be completed quickly
- When the currently executing thread reaches an atomic block:
 - Start executing atomic block
 - If we finish the block before the thread is interrupted, we don't need to do anything special
 - If we're interrupted, roll back and try again next time
- We allow scheduler to pre-empt threads, so we maintain the benefits of concurrency
- Atomic blocks are usually short, so rollbacks will be rare.

Some details

- To make rollback possible, in an atomic block we must:
 - Keep a log of all writes in an atomic block.
 - Buffer all output and message sends, and avoiding flushing until the block completes.
- Blocking operations can trigger immediate rollbacks when the requested resources are not available.
- Duplicate functions - one copy logs, other does not. Call logging version inside atomic blocks.

Some Questions

- What if an atomic block is too long to complete in a time slice?
- What about function pointers that could be used in both atomic and non-atomic contexts?
- What is the most efficient way to handle logging?
- What if an atomic block throws an exception?
- We have answers for these (and more!). Come talk to me later if you're curious.

Current and Future Work

- We're implementing AtomCAML–O'CamI with atomicity.
- Plan to implement for C# and integrate with MSR's Singularity OS.
- We have also designed a system that works when threads exhibit true parallelism.
- We plan to develop a lock inference system.

Summary

- Concurrency is dangerous, but important.
- We propose using atomicity as a shared-memory synchronization primitive.
- We provide a strong guarantee of atomicity, and an efficient implementation for an important class of systems.
- We're working on implementations for an even larger class of systems.