Atomic Variables and Operations

Godmar Back

Virginia Tech

March 18, 2024

VIRGINIA TECH

Preface

- You can ignore this entire lecture if you write only programs that use proper locking for all accesses (read or write) to shared variables and that use only semaphores and condition variables for inter-thread signaling
- When proper locking is used, multiple threads can access a shared address space and they see the same values of these shared variables, corresponding to the basic intuition of a shared-memory multi-threading programming model
  - this is unlike `fork()`
- This lecture touches on what lies beyond

VIRGINIA TECH

Story So Far

- Only data-race free programs provide sequential consistency (a sequential order of all memory operations in terms of "steps" is observed by all threads and it is consistent with the order in the program).
- The traditional way to ensure this is by using locks, semaphores, and condition variables
  - Locks: The second thread to acquire a lock and enter a critical section will see all updates made by the first thread to have acquired the lock
  - Semaphores/Condition Variables: A thread returning from a call to wait will see all updates performed prior to the signal operation that caused the thread to return from wait
- Ensuring data-race freedom validates programmer intuition
  - Opposite: 2 threads see updates in different order: thread 1 updates A then B, thread 2 sees new value of B and old value of A
  - Data-race freedom stipulates only that a sequentially consistent ordering exists, it doesn't say which one it is
- See Adve & Boehm [2] for precise definition and discussion

VIRGINIA TECH

What if Locks Are Too Slow?

- Or: are there other ways to constrain compiler & processor?
- C11/C++11 atomic variables are "synchronization variables" [1]
- Their use disallows certain observed interleavings for the memory operations preceding and following accesses to these variables
- They do not place threads into the BLOCKED state
- Concurrent accesses to synchronization variables are not considered races
- These variables also can be atomically updated in read-modify-write operations
- By default (`memory_order_seq_cst`), they ensure the existence of a sequentially consistent ordering for accesses to them and accesses to non-atomic variables in between atomic accesses

VIRGINIA TECH

Recap: failure of busy-waiting "done" flag check

waitingonaflag.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool done;
int x;

void thread1() {
    x = rand() % 2;
    done = true;
}

int thread2() {
    while (!done) { }
    return x;
}
```

- compiler reorders statements
- compiler replaces loops with ifs

compiled with gcc 7.5

```
thread1:
    subq    $8, %rsp
    call    rand@PLT
    movl    %eax, %edx
    movb    $1, done(%rip)   # done = true
    shrl    $31, %edx
    addl    %edx, %eax
    andl    $1, %eax
    subl    %edx, %eax
    movl    %eax, x(%rip)    # x = ...
    addq    $8, %rsp
    ret

thread2:
    cmpb    $0, done(%rip)
    jne .L8      # if !done goto L8
.L7:             # else: loop forever
    jmp .L7
.L8:
    movl    x(%rip), %eax
    ret
```

VIRGINIA TECH

C11 Atomics

waitingonaflag-atomic.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdatomic.h>

atomic_bool done;
int x;

void thread1() {
    x = rand() % 2;
    done = true;
}

int thread2() {
    while (!done) { }
    return x;
}
```

- See C11 atomic types

compiled with gcc 7.5

```
thread1:
    subq    $8, %rsp
    call    rand@PLT
    movl    %eax, %edx
    shrl    $31, %edx
    addl    %edx, %eax
    andl    $1, %eax
    subl    %edx, %eax
    movl    %eax, x(%rip)
    movb    $1, done(%rip)
    mfence
    addq    $8, %rsp
    ret

thread2:
.L5:
    movzbl  done(%rip), %eax
    testb   %al, %al
    je  .L5
    movl    x(%rip), %eax
    ret
```

VIRGINIA TECH

C11 Atomics (ARM64)

without atomics

```
thread1:
    stp x29, x30, [sp, -16]!
    mov x29, sp
    bl  rand
    cmp w0, 0
    adrp    x2, :got:x
    adrp    x1, :got:done
    and w0, w0, 1
    mov w3, 1
    ldr x2, [x2, #:got_lo12:x]
    csneg   w0, w0, w0, ge
    ldr x1, [x1, #:got_lo12:done]
    str w0, [x2]
    strb    w3, [x1]
    ldp x29, x30, [sp], 16
    ret

thread2:
    adrp    x0, :got:done
    ldr x0, [x0, #:got_lo12:done]
    ldrb    w0, [x0]
    cbnz    w0, .L8
.L7:
    b   .L7
.L8:
    adrp    x0, :got:x
    ldr x0, [x0, #:got_lo12:x]
    ldr w0, [x0]
    ret
```

with atomics

```
thread1:
    stp x29, x30, [sp, -16]!
    mov x29, sp
    bl  rand
    cmp w0, 0
    adrp    x2, :got:x
    adrp    x1, :got:done
    and w0, w0, 1
    ldr x2, [x2, #:got_lo12:x]
    csneg   w0, w0, w0, ge
    ldr x1, [x1, #:got_lo12:done]
    str w0, [x2]
    mov w0, 1
    stlrb   w0, [x1]
    ldp x29, x30, [sp], 16
    ret

thread2:
    adrp    x1, :got:done
    ldr x1, [x1, #:got_lo12:done]
.L5:
    ldarb   w0, [x1]
    tst w0, 255
    beq .L5
    adrp    x0, :got:x
    ldr x0, [x0, #:got_lo12:x]
    ldr w0, [x0]
    ret
```

VIRGINIA TECH

Simple Use Cases

- Certain accesses that were previously unsafe can be done now

Checking Write-Once Variables

```
atomic_bool gameover;
// ....
if (gameover) {
    // game is over and it is safe to access the results
}
```

Double Checked Locking Idiom

```
pthread_mutex_lock lock;
_Atomic struct sometype * s;

// Goal
struct sometype *
makeSingleton() {
    if (s == NULL) {   // access without lock
       pthread_mutex_lock(&lock);
       if (s == NULL) {  // double-check
            tmp = malloc(...);
            initialize(tmp);
            s = tmp;  // s being atomic, all writes
                      // inside initialize are seen
                      // when another thread sees
                      // s != NULL
       }
       pthread_mutex_unlock(&lock);
    }
    return s;
}
```

VIRGINIA TECH

Read-modify-write Operations

atomicupdates.c

```c
#include <stdatomic.h>

atomic_int i;
atomic_ulong d;

void atomic_updates()
{
    i = i + 5;  // non-atomic
    i += 5;
    d /= 2;
}
```

- Certain operations are turned by the compiler into atomic updates
  - e.g., `a++`, `a *= 2`
  - but not `a = a + 1`

atomicupdates.s

```
atomic_updates:
    movl    i(%rip), %eax
    addl    $5, %eax
    movl    %eax, i(%rip)
    mfence
    lock addl   $5, i(%rip)
    movq    d(%rip), %rax
.L2:
    movq    %rax, %rdx
    shrq    %rdx
    lock cmpxchgq   %rdx, d(%rip)
    jne .L2
    rep ret
```

- Either using atomic instructions provided by the architecture, or using a loop based on atomic compare-and-exchange or equivalent

VIRGINIA TECH

Combining Atomics with Lock-based Synchronization

- Tricky, consider

Atomics + Condition Variables

```
atomic_bool gameover;
// ...
// BUG: Checking the condition `gameover`
// is not atomic with respect to calling pthread_cond_wait
while (!gameover) {
    pthread_mutex_lock(&lock);
    pthread_cond_wait(&cond, &lock);
    pthread_mutex_unlock(&lock);
}
```

- Binary instrumentation-based race detection tools (Helgrind, DRD) are generally unaware of atomics

Should you prefer this:

```
atomic_int inqueuecount;   // count of items in queue
pthread_mutex_lock queuelock;  // queue lock

void enqueue(struct item *item)
{
    inqueuecount++;
    pthread_mutex_lock(&queuelock);
    add_to_queue(item);
    pthread_mutex_unlock(&queuelock);
}
```

to this?

```
int inqueuecount;
pthread_mutex_lock queuelock;

void enqueue(struct item *item)
{
    pthread_mutex_lock(&queuelock);
    inqueuecount++;
    add_to_queue(item);
    pthread_mutex_unlock(&queuelock);
}
```

VIRGINIA TECH.

Lock-free Synchronization

- Although locks work sufficiently well for many scenarios, and provide a general facility for implementing any kind of atomic modifications, they have a number of drawbacks; to list some:
  - Potential for reduced CPU utilization when synchronization is too coarse-grained
  - Increased potential for deadlock when too fine-grained
  - Potential for performance decrease when highly contended
  - Potential for priority inversion (low-priority threads hold up high-priority threads by holding locks those threads want)
  - Convoying: threads holding locks for long periods of time create "convoys" behind them
  - No good support for asynchronous termination (kill) of threads holding locks
  - Don't play well with Unix signals
- These shortcomings gave rise to certain "lock-free" synchronization algorithms that are implemented using atomic operations
  - data structures: lock-free stacks, lists, etc.
  - e.g. `java.util.concurrent.ConcurrentHashMap`
- But their study is a topic for a separate lecture or class

VIRGINIA TECH

Addendum: the `volatile` keyword

- In C/C++, `volatile` says that any access (read or write) should be considered to have a side-effect, thus the compiler cannot optimize it out or reorder it.
- Great for memory mapped I/O, for instance
- Unlike atomics, it has no effect on what other threads see (does not introduce fences or acquire/release load/stores), and thus cannot be used for interthread communication
  - Historic note: prior to the arrival of C11 support, programmers used `volatile` in hackish and unreliable attempts at getting the compiler to produce the desired code
- In Java, `volatile` is similar to atomic variables in C11/C++11 in the default setting `memory_order_seq_cst`, except without the ability to do atomic read-modify-writes (see `java.util.concurrent.atomic` for the latter).

VIRGINIA TECH

References

[1] C11 atomic operations library.
https://en.cppreference.com/w/c/atomic.

[2] Hans-J. Boehm and Sarita V. Adve.
You don't know jack about shared variables or memory models.
Commun. ACM, 55(2):48–54, February 2012.

VIRGINIA TECH