Lock-Free Programming

Peter Chapin

CIS-4230, Parallel Programming

Vermont State University

Suppose…

- You wanted to track the number of times each word occurs in a large document (or large set of documents)
  - Create some binary search tree (e.g., RedBlack Tree)
  - For each word, if the word is not already mentioned as a key in the tree, add it with a count of one. Otherwise, increment the count associated with the word.
- Later in the program, this information is accessed. Using the word as a key, look up its associated count.
  - `insert( word_tree, “Hello” );`
  - `int count = lookup( word_tree, “Hello”);  // e.g., 42`

Why?

- The use of a tree data structure offers an advantage in terms of efficiency (log time) for insert and lookup operations.
- Imagine the program does extensive processing on the word count information, presumably with other data.
  - Thus, the program uses parallel threads.
- Is it okay for multiple threads to build the tree?
- Is it okay for multiple threads to read the tree?

Multiple Reads

- It is generally not an issue for multiple threads to read a data structure in parallel
  - No data is (usually) modified, so there is no interference between threads.
- Except…
  - Some data structures are modified by reads!
  - Caching
  - Restructuring during reads
- This isn’t common, however.
  - For example, RedBlack Trees don’t usually cache or restructure themselves during reads.

Our (Hypothetical) Application

- If our application spends more time reading the tree than building it (which is typical)…
  - … We don’t have to worry about it!
  - Read in parallel from multiple threads.
  - Build the tree in one thread (e.g., while reading the input files)

Building the Tree

- But what if we want to build the tree in parallel, too?
  - For example, we have a **huge** number of files to process
  - We have multiple threads reading different subsets of those files…
  - … and inserting/incrementing word counts in parallel.
- If we do nothing, we'll have *race conditions*.
  - Two threads read a count (e.g., 41)
  - Both threads increment the count to 42.
  - Both threads write back 42.
  - *But wait! The count should have been 43!*
- <u>Also, structural problems when adding and adjusting nodes</u>

One Solution: Mutual Exclusion

- We could use a mutex object associated with the tree.
  - Each thread locks the mutex before trying to insert.
  - If the mutex is already locked (by an earlier thread), the new thread is suspended until the mutex is unlocked (by the earlier thread).
  - Thus, each thread has *mutually exclusive* access to the shared structure.
- This works and is relatively easy to implement and understand.
- <u>Unfortunately, it also reduces concurrency and parallelism!</u>
  - But… do we care??

Issues with Locks

- Locks have some problems.
  - **Overhead**. If a thread suspends, the OS is involved (for kernel threads). There is significant overhead going into the OS and letting the OS schedule some other thread. *If the time the lock is held is short, this overhead can be significant*.
  - **Deadlock**. If two (or more) locks are involved, the threads can try to lock both and end up suspended, waiting for each other.
    - Thread 1 locks A
    - Thread 2 locks B
    - Thread 1 locks B ***suspends***
    - Thread 2 locks A ***suspends***

Lock-Free?

- Is it possible to create a mutex-like lock without using an actual lock?

Global variable →

```
int flag = 0;  // A value of 1 means locked.
```

Each thread executes →

```
while( flag == 1 ) /* Wait */ ;
flag = 1;

// Mutually exclusive access?

flag = 0;
```

Sometimes called a *spinlock*

- When a thread waits, it spins in a loop ("busy waiting").
  - This is normally bad but can be okay if the waiting time is short.

Problem #1

- An optimizing compiler will likely enregister the value of `flag` in the while condition. It will then test the register repeatedly without looking at changes to the in-memory variable.

- Tell the compiler this value can change for reasons outside its control (e.g., in a different thread). So, reload it from memory whenever it is needed.

```
volatile int flag = 0;  // A value of 1 means locked.

while( flag == 1 ) /* Wait */ ;
flag = 1;

// Mutually exclusive access?

flag = 0;
```

Problem #2

- The solution with `volatile` isn't good enough.
    1. Suppose two threads are spinning with `flag == 1`.
    2. A third thread sets `flag` to 0.
    3. Both threads see the change simultaneously and exit the while loop.
    4. Both threads set `flag` to 1.
    5. Both threads are now in the "mutually exclusive" region, aka the *critical section*.
- There is still a *race condition*.
- The solution is surprisingly hard to get right (see *Peterson's Algorithm*)

Compare and Swap

- This really needs assistance from the hardware.
  - We need an operation that can't be interrupted by another processor.
  - A special machine cycle is usually required, which only hardware can do.
    - Read. Get a value from memory.
    - Write. Put a value into memory.
    - Read-Modify-Write. Get a value, change it (e.g., increment), and put the result back.
    - Compare-and-Swap. If a value hasn't changed, exchange it with a new value.
  - Read/Write alone allows another processor to get in the middle.
  - The last two options above need special instructions or processor modes.
- Compare-and-Swap is a powerful primitive and popular.

C-Like Pseudo-Code

```
int compare_and_swap(int *reg, int oldval, int newval)
{
    ATOMIC();
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    END_ATOMIC();
    return old_reg_val;
}
```

On the CPU, this entire function is a single, atomic instruction.

Return old value, which might be unexpected (if another thread changed it before we did this)

Source: Wikipedia

CMPXCHG (x86_64)

- The “compare and exchange” instruction does this on x86_64 architecture. Other architectures may call it something different.

```
mov eax, expected_value   ; Load the expected value into EAX
mov ecx, new_value        ; Load the new value to be exchanged into ECX
lock cmpxchg [dest], ecx  ; Atomically compare [dest] with EAX
```

- Compares `eax` with `[dest]`. If they are equal, `[dest]` is loaded with `ecx`. Otherwise, `eax` is loaded with `[dest]`.
- The `lock` prefix is needed for multiprocessor systems.