Lecture 6b:

Locks and Avoiding Locks

Better spinlocks

Non-blocking Synchronization

Read-Copy Update

Transactional Memory (nope)

CSC 469 / CSC 2208

Fall 2019

(with thanks to Tom Hart, Paul McKenney)
Idea: Don’t lock if we don’t need to!

Next time:

- Non-Blocking Synchronization (NBS)
  - Use term “lockless” to describe strategies that avoid locking
NBS Basics

- Make change optimistically, roll back and retry if conflict detected

```c
atomic_inc(int *counter) {
    int value;
    do {
        value = *counter;
    } while (!CAS(counter, value, value+1);
}
```

- Complex updates (e.g. modifying multiple values in a structure) are hidden behind a single commit point using atomic instructions
Example: Stack Data Structure

- Lock-based synchronization:

```c
/* definitions */

typedef struct node_s {
    int val;
    struct node_s *next;
} node_t;

typedef struct stack_s {
    node_t *top;
    lock_t *stack_lock;
} stack_t;

void push(stack_t *S, node_t *n) {
    lock(S->stack_lock);
    n->next = S->top; S->top = n;
    unlock(S->stack_lock);
}

node_t* pop(stack_t *S) {
    node_t *n = NULL;
    lock(S->stack_lock);
    if (S->top != NULL) {
        n = S->top;
        S->top = S->top->next;
    }
    unlock(S->stack_lock);
    return n;
}
```
Non-blocking stack (take 1)

```c
/* definitions */

typedef struct node_s {
    int val;
    struct node_s *next;
} node_t;

/* Stack type is just a */
* pointer to a node.
*/
typedef node_t *stack_t;

void push(stack_t *S, node_t *n) {
    node_t *first;
    do {
        first = *S;
        n->next = first;
    } while (!CAS(S,first,n));
}

node_t* pop(stack_t *S) {
    node_t *first, *second;
    do {
        first = *S;
        if (first != NULL) {
            second = first->next;
        } else return NULL;
    } while (!CAS(S,first,second));
    return first;
}
```

What's wrong?
ABA Problem

- Ti, Tj both doing pops and pushes, interleaved as follows:

  Ti: pop()
  first
  second (interrupt)

  Tj:
  a = pop();
  b = pop();
ABA Problem

- \( \text{CAS}(x, y, z) \) succeeds if value stored at \( x \) matches \( y \)

### Diagram

- Ti: \( \text{pop()} \)
  - first
  - second
  - (interrupt)

- Tj:
  - \( a = \text{pop}(); \)
  - \( b = \text{pop}(); \)
  - \( \text{push}(n); \)
  - \( \text{push}(a); \)

\( \text{CAS}(S, \text{first}, \text{second}) \)
ABA Problem

- CAS(x, y, z) succeeds if value stored at x matches y

Ti: pop()
    
    first
    
    second
    
    (interrupt)

Tj:
    
    a = pop();
    
    b = pop();
    
    push(n);
    
    push(a);

CAS(S, first, second)
One Solution

• Include a version number with every pointer
  • pointer_t = <pointer, version>
  • Increment version number (atomically) every time you modify pointer
  • Change to version number guarantees CAS will fail if pointer has changed
  • Requires double-word CAS operation (not every architecture provides this)
• Use garbage collection to reclaim memory later
  • May restrict reuse of memory
Using NBS

- Good for simple data structures, update heavy
- When you need NBS constraints/guarantees
  - Progress in face of failure
  - Linearizability
    - Everyone agrees on all intermediate states
- Both constraints are often irrelevant!
Constraints Irrelevant?

- Real systems don’t fail the way theoretical ones do
  - Software bugs are not always fail-stop
  - Preemption/interrupt is not a failure
    - And can be controlled by system programmer or scheduler-conscious synchronization
  - Page fault is not a failure
    - Over-provision memory... if shared data really is paged out, it will have to be brought into memory before progress is made anyway
- Don’t always need intermediate states, just final
  - Linearizability implies dependency → limits parallelism
  - If events are unrelated, asynchronous, does it matter which happened first?
Read-Copy Update (RCU)

• What is RCU?
  • Paul McKenney’s PhD thesis
  • a key part of the Linux scalability effort

• Ok, what is it really?
  • Reader-writer synchronization mechanism
    • Readers use no locks; best for read-mostly data structures
    • Writers create new versions atomically
      • typically by locking out other writers
  • Readers can continue to access old versions
    • Old versions must be deleted at some point
    • “poor man’s garbage collection”
Why RCU?

• Consider concurrent hash table example
  • Hash function selects bucket (entry in an array)
  • Collisions handled by chaining (linked list per bucket)
  • Use per-bucket locks to increase concurrency

• But recall costs of synchronization operations...
What about NBS?

- Non-blocking synchronization is possible for hash table operations
- But still expensive, *even for read-only operations*
- Why? Consider concurrent lookup and delete operations:

T1: read N

T1 obtains pointer to Node N. Needs to ensure N continues to exist until T1 is done using it.

T2: remove N

T2 must detect that Node N is in use and defer deletion.
Reference counting solution

- T1 can increment reference count on N
  - And on every node along path to N!
  - Requires an atomic update per node from T1
- T2 must defer deletion of a node with elevated reference count

T1: read N

T1: atomic_inc(N->refcount)

T2: remove N

T2: while(N->refcount > 1) {};}
Reader/Writer locks?

- Concurrent reads, exclusive writes

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>Reader</th>
<th>Reader</th>
<th>Blocked</th>
<th>Reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 1</td>
<td>Reader</td>
<td>Reader</td>
<td>Blocked</td>
<td>Reader</td>
</tr>
<tr>
<td>CPU 2</td>
<td>Reader</td>
<td>Reader</td>
<td>Blocked</td>
<td>Reader</td>
</tr>
<tr>
<td>CPU 3</td>
<td>Reader</td>
<td>Reader</td>
<td>Spin</td>
<td>Writer</td>
</tr>
</tbody>
</table>

- Lots of “dead time” as all readers wait for single writer to finish.
RCU Design Principle

• Avoid mutual exclusion!

CPU 0  Reader  Reader  Reader
CPU 1  Reader  Reader  Reader
CPU 2  Reader  Reader  Reader
CPU 3  Reader  Reader  Writer  Reader

• No more “dead time”
• But how can this be implemented?
RCU Basics

1. Publish/Subscribe mechanism
   • Updater publishes new data
   • Reader subscribes to current version
2. Mechanism to wait for previous readers to complete (for deletion)
3. Mechanism to maintain multiple versions of recently updated objects (for readers)

From [http://lwn.net/Articles/262464](http://lwn.net/Articles/262464)
When is it safe to read a pointer?
- RCU Readers use no locks
- Compiler, CPU may reorder memory assignments and reads

```c
/* definitions */
struct foo {
    int a;
    int b;
    int c;
};

/* gp == global ptr */
struct foo *gp = NULL;

T1 (Updater):
    p = kmalloc(sizeof(*p));
    p->a = 1;
    p->b = 2;
    p->c = 3;
    gp = p; // gp can be read by others

T2 (Reader):
    p = gp; // get ptr to shared data
    if (p != NULL)
        use(p->a, p->b, p->c);
```
Memory Order “Mischief”

Original order as written:

T1 (Updater):
  p = kmalloc(sizeof(*p));
  p->a = 1;
  p->b = 2;
  p->c = 3;
  gp = p;

T2 (Reader):
  retry:
  p = gp;
  if (p == NULL)
    goto retry;
  use(p->a, p->b, p->c);

Mischievous re-ordering:

T1 (Updater):
  p = kmalloc(sizeof(*p));
  gp = p;
  p->a = 1;
  p->b = 2;
  p->c = 3;

T2 (Reader):
  retry:
  p = guess(gp);
  use(p->a, p->b, p->c);
  if (p != gp)
    goto retry;

• Both compiler and CPU may reorder memory assignments and reads
  • Value speculation (reader example) also occurs in some CPUs
When is it safe to read a pointer?

- RCU Readers use no locks
- Compiler, CPU may reorder memory assignments and reads
Readers use `rcu_dereference()` to "subscribe" to the current version of some data object.

- Also need a way to detect when reader is done using that version.
• Wait... we said readers use no locks!
  • These API calls mark the boundaries of an RCU reader’s critical section (c.s.).
  • Lightest-weight implementation (assume non-preemptible environment)
    • `#define rcu_read_lock() // Nothing at all`
    • `#define rcu_read_unlock() // Nothing at all`
• Why even bother if they do nothing?
  • In preemptible environment, must (at minimum) disable pre-emption
  • Useful clue for programmers
    • Must not access rcu-protected pointer outside c.s.
RCU Deletion Example

• T1 traversing linked list, T2 removes an element:

T1: read N

T2: remove N
• After removal – T1 continues to use N and later nodes in the list

When is it ok to delete N (and reuse the memory for something else)?
Handling read-reclaim races

• RCU uses *quiescent state based reclamation* (QSBR)

• **Defn:** A *quiescent state* for a thread $T$ is a state in which $T$ holds no references to shared data

• **Defn:** A *grace period* is an interval in which every thread has passed through at least one quiescent state

• **Basic Idea:** elements removed from a data structure can be reclaimed after a grace period, since no thread can still be holding a reference to the old element at that point
Any element N removed before this point…

Grace Period

… can be reclaimed after this point.

Thread 1

Thread 2

Thread 3

T2 cannot hold ref to N after this point.  

T3 cannot hold ref to N after this point.  

T1 cannot hold ref to N after this point.
How to define Quiescent States?

- Application dependent!
- For OS kernels, some natural ones exist
  - E.g. a context switch in a non-preemptive kernel
- RCU primitives
  - `rcu_read_lock()` and `rcu_read_unlock()`
    - Surround read-side critical sections
    - No overhead (#define’d as nothing) in non-preemptive kernels
    - Low overhead in preemptive kernels (disable preemption)
  - `synchronize_rcu()`
    - Wait until all pre-existing RCU read-side critical sections complete
    - Force execution on all CPUs
      - Works because RCU readers can’t be preempted.
PPC Hash Table with RCU

![Graph showing searches per unit time normalize to ideal versus number of CPUs for different algorithms: "ideal", "RCU", "HPBR", "spinbkt", "brlock", "globalrw".](image)
Growth of RCU use in Linux

(Oct. 15, 2019, generated daily)
...but still small in comparison

Graph from http://www.rdrop.com/users/paulmck/RCU/linuxusage.html
(Oct. 15, 2019, generated daily)
When to use which tool

• Read-mostly situations
  • RCU (if algorithm can tolerate concurrent reads and updates)

• Update-heavy situations
  • Simple data structures and algorithms: NBS
  • Complex data structures and algorithms: Locking

“When the only tool you have is a hammer, everything looks like a nail.”

• It’s good to have lots of tools in your toolbox
Announcements

• Midterm next week Wednesday!
  • Check web page for location

• Tutorial on Monday will review practice questions