ECE 454
Computer Systems Programming

Avoiding Locks

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin

With thanks to Angela Demke Brown, Tom Hart, Paul McKenney
Overview

• Challenges with Locking
• Non-Blocking Synchronization
• Read-Copy Update
Challenges with Locking
Locking: A Necessary Evil?

- Locks - easy solution to critical section problem
  - Protect shared data from corruption due to simultaneous updates
  - Protect against inconsistent views of intermediate states

- But locks have lots of problems
  - 1. Deadlock
  - 2. Priority inversion
  - 3. Not fault tolerant
  - 4. Convoying
  - 5. Expensive, even when uncontended

- Not easy to use correctly!
1. Deadlock
1. Deadlock

- Textbook definition: Set of threads blocked waiting for event that can only be caused by another thread in the same set

  /* a threaded program with a potential for deadlock */

  Thread1()
  {
    lock(a);
    lock(b);
    do_work();
    unlock(b);
    unlock(a);
  }

  Thread2()
  {
    lock(b);
    lock(a);
    do_work();
    unlock(a);
    unlock(b);
  }

- Solutions exists but add complexity
  - E.g., specify lock order
2. Priority Inversion

- Lower priority thread gets spinlock
- Higher priority thread becomes runnable and preempts it
  - Needs lock, starts spinning
  - Lock holder can’t run and release lock

Solutions exist but add complexity
- E.g. disable preemption while holding spinlock, implement priority inheritance, etc.
3. Not Fault Tolerant

- If lock holder crashes, or gets delayed, no one makes progress

  ☑️ lock

  ✗ CRASH!

- Delays can happen due to preemption, page faults
  - Disable such delays, e.g., pin pages in memory
  - Avoid critical sections when delays will happen

- Crashes require abort / restart
4. Convoying

- Threads started at different times occasionally access shared data
- Expect shared data accesses to be spread out over time
  - Lock contention should be low
- Delay of lock holder allows other threads to catch up
  - Lock becomes contended and tends to stay that way
  - => Convoying
5. Expensive, Even When Uncontended!

<table>
<thead>
<tr>
<th>Operation</th>
<th>Nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>0.24</td>
</tr>
<tr>
<td>Clock Cycle</td>
<td>0.69</td>
</tr>
<tr>
<td>Atomic Increment</td>
<td>42.09</td>
</tr>
<tr>
<td>Cmpxchg Blind Cache Transfer</td>
<td>56.80</td>
</tr>
<tr>
<td>Cmpxchg Cache Transfer and Invalidate</td>
<td>59.10</td>
</tr>
<tr>
<td>SMP Memory Barrier (eieio)</td>
<td>75.53</td>
</tr>
<tr>
<td>Full Memory Barrier (sync)</td>
<td>92.16</td>
</tr>
<tr>
<td>CPU-Local Lock</td>
<td>243.10</td>
</tr>
</tbody>
</table>

McKenney, 2005 – 8-CPU 1.45 GHz PPC
Critical Section Efficiency

- Assuming little to no contention, and no caching effects in CS

\[
\text{Efficiency} = \frac{T_c}{T_c + T_a + T_r}
\]

- Ta and Tr can take 100+ cycles, even with no contention
- Critical section efficiency must be addressed!
Causes: Deeper Memory Hierarchy

- Memory speeds have not kept up with CPU speeds
  - 1984: no caches needed, since instructions slower than memory accesses
  - after ~2005: 3-4 level cache hierarchies, since instruction speeds are orders of magnitude faster than memory accesses
- Synchronization ops typically execute at memory speed
Causes: Deeper Pipelines

Then:

| Fetch | Execute | Retire |

Now:

1984: Many cycles per instruction

2005: Many instructions per cycle

- 20 stage pipelines
- CPU logic executes instructions out-of-order to keep pipeline full
- Synchronization instructions cannot be reordered
- => Synchronization stalls the pipeline
Performance

• Main issue with lock performance used to be contention
  • Techniques were developed to reduce overheads in contended case
    • E.g., MCS locks

• Today, issue is degraded performance even when locks are always available
  • Together with other concerns about locks
Locks: A Necessary Evil?

Idea: Don’t lock if we don’t need to!

• Use “lockless” synchronization
  • Design data structures so that locks are not required
Non-Blocking Synchronization
Non-Blocking Synchronization (NBS) Basics

- Think of NBS as a “lockless” synchronization scheme
  - With locking, threads access shared object under mutual exclusion
  - With NBS, threads can access shared object concurrently

- Idea: make change optimistically, if conflict detected, roll back

```c
// atomically increment *counter using CAS
atomic_inc(int *counter) {
    int value;
    do {
        value = *counter;  // save value of 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: Lock-Based Stack

class Node {
    Node *next;
    int data;
};
Node *head; Lock *l;

void push(Node *node) {
    lock(l);
    node->next = head;
    head = node;
    unlock(l);
}

Node *pop() {
    int current = NULL;
    lock(l);
    if (head) {
        current = head;
        head = head->next;
    }
    unlock(l);
    return current;
}
Example: Lock-Free Stack

```c
void push(Node *node) {
    do {
        node->next = head;
    } while (!CAS(&head, node->next, node));
}

Node *pop() {
    Node *current = head;
    while (current) {
        if (CAS(&head, current, current->next)) {
            return current;
        }
        current = head; // head may have changed
    }
    return NULL;
}
```

class Node {
    Node *next;
    int data;
};

Node *head;
ABA Problem

- Notice that `pop` reads `head` twice.
- If the value of `head` hasn’t changed, then `head` is updated.
- What if another thread updates `head` in between, does other work, and then changes `head` back to the old value?

```c
Node *pop() {
    Node *current = head;
    while (current) {
        if (CAS(&head, current, current->next)) {
            return current;
        }
    }
    ...
}
```
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:
  - Ti: starts \texttt{pop()}
    - head is A
    - current is A
    - current->next is B (loaded in reg)
    - Ti \textbf{interrupted before} it performs:
      \texttt{CAS(&head, current, current->next)},
      i.e., before head is assigned to B

```c
Node *current = head;
...
if (CAS(&head, current, current->next))
```

Diagram:

```
  head
  ↓
  A
  ↓
  B
  ↓
  C
```
ABA Problem

• Say Ti, Tj are both doing pops and pushes on this stack:
ABA Problem

• Say Ti, Tj are both doing pops and pushes on this stack:
• Tj:
  • a=pop()
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:
  - Tj:
    1. \( a = \text{pop}(\) \)
    2. \( b = \text{pop}(\) \)

![Diagram of stack with labels B, A, C and arrows indicating a and b]
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:
- Tj:
  - a=pop()
  - b=pop()
  - push(N)
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:
  - Tj:
    - a = pop()
    - b = pop()
    - push(N)
    - push(a)
      - ‘a’ is the same node that was returned by first pop()
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:
  - Tj:
    - a=pop()
    - b=pop()
    - push(N)
    - push(a)
  - Ti resumes: head is A
    - current is A, current->next is B
    - CAS succeeds, sets head to B!
    - Returns A, A->next set to NULL
    - Stack should have been N, C
One Solution

• Include a version number with every pointer
  • `pointer_t = <pointer, version>`
  • Increment version number every time pointed-to data is modified
  • Atomically update pointer and version using double-word CAS
    • Consider pop code: `CAS(&head, current, current->next)`
    • Say `current = <A, 1>; After head is updated, head = <A, 2>`
    • Version number ensures CAS will fail if data has changed

• Issues
  • Not every architecture provides double-word CAS operation
  • Old versions of pointers need to be freed
    • Use garbage collection to reclaim memory later
    • May restrict reuse of memory
Using NBS

- Generally used for simple, update-heavy data structures
  - E.g., linked list
  - See https://en.wikipedia.org/wiki/Non-blocking_linked_list
  - Hard to design data structures that use NBS
When do we need NBS Guarantees?

• When we need linearizability
  • Everyone agrees on all intermediate states
    • All updates appear instantaneous, occur in total order
    • Reads return value of last completed write
  • Imposes dependency between operations
    • Limits parallelism

• Do we always need linearizability?
  • Consider “top” program that lists all existing processes
Read-Copy Update (RCU)
Read-Copy Update (RCU)

- **What is RCU?**
  - Paul McKenney’s PhD thesis
  - A key part of the Linux scalability effort

- **Reader-writer synchronization mechanism**
  - Supports concurrency between multiple readers + single updater
  - **Readers use no locks**
    - Hence best for read-mostly data structures
  - **Writers create new versions atomically**
    - Either using atomic instructions or by locking out other writers
  - **Readers may continue to access old versions**
    - Old versions must be deleted at some point
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

- Consider concurrent lookup and remove operations:

  T1: read N

  T1 obtains pointer to Node N. Need 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
  - Requires atomic update for each node along path to N on a read!
- T2 must defer deletion of a node with elevated reference count

```c
T1: atomic_inc(N->refcount)
T2: while(N->refcount > 1) {};
```
Reader/Writer locks?

- Concurrent reads, exclusive writes

- 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

- Three key ideas
  - Use publish/subscribe ordering mechanism
    - Orders operations so readers see consistent, atomic updates
  - Maintain multiple versions of recently updated objects
    - Ensures readers that are concurrent with writers will read consistent (but perhaps stale) data versions
  - Wait for previous readers to complete
    - For deleting old versions
- All three together ensure that reads can be performed correctly without using locks
- See LWN article: http://lwn.net/Articles/262464
Understanding the Need for Publish/Subscribe

- No locks are being used by reader
- When is it safe to dereference the gp pointer, i.e., is use(p->a) guaranteed to return 1?

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

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

T1 (Writer):
    p = malloc(sizeof(*p));
    p->a = 1;
    gp = p;  // gp can be read by others

T2 (Reader):
    retry:
        p = gp;  // get ptr to shared data
        if (p == NULL)
            goto retry;
        use(p->a);
```
Problem 1

Compiler, CPU can reorder memory assignments and reads

T1 (Writer):
\[ p = \text{malloc} \left( \text{sizeof}(*p) \right); \]
\[ p->a = 1; \]
\[ \text{gp} = p; \]

Problem 1

T1 (Writer):
\[ p = \text{kmalloc} \left( \text{sizeof}(*p) \right); \]
\[ \text{gp} = p; \]
\[ p->a = 1; \]

T2 (Reader):
retry:
\[ p = \text{gp; } // \text{get ptr to shared data} \]
if (p == NULL)
\[ \text{goto retry; } \]
use(p->a);  // may read uninitialized value!
Memory Order “Reader Mischief”

Compiler, CPU can reorder memory assignments and reads

T1 (Writer):
```c
p = malloc(sizeof(*p));
p->a = 1;
gp = p; // gp can be read by others
```

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

T2 (Reader):
```
retry:
p = guess(gp);
use(p->a); // old value
if (p != gp) // fails!
goto retry;
```
RCU Publish/Subscribe Ordering Mechanism

- Enforce ordering with `rcu_assign_pointer/rcu_dereference`
  - They encapsulate memory barriers, ensuring the correct ordering

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

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

T1 (Writer):
p = malloc(sizeof(*p));
p->a = 1;
gp = p; rcu_assign_pointer(gp,p);

T2 (Reader):
retry
    p = gp; p = rcu_dereference(gp);
    if (p == NULL)
        goto retry;
    use(p->a);
```
Maintaining Multiple Versions

• Two examples using linked list
  • Update
  • Deletion
RCU List Element Update

- T1 traversing linked list, T2 updates an element:
RCU List Element Update

- T1 traversing linked list, T2 updates an element:

  T1: read N

  RC: T2 Reads and makes a Copy of N

  T2: update N
RCU List Element Update

- T1 traversing linked list, T2 updates an element:

  T1: read N  
  T2: update N

  RC: T2 Reads and makes a Copy of N  
  U: T2 Updates prev to N’ atomically

Why make a copy of N?

When is it ok to delete N (and reuse the memory for something else)?
RCU List Element Deletion

- T1 traversing linked list, T2 removes an element:

T1: read N

T2: remove N
RCU List Element Deletion

- After removal – T1 continues to use N and later nodes in the list

![Diagram showing RCU List Element Deletion]

T1: read N
T2: remove N

T2 updates prev to next atomically

When is it ok to delete N (and reuse the memory for something else)?
Waiting for Previous Readers

• RCU needs to wait for previous readers to reclaim old versions
• RCU uses quiescent-state based reclamation (QSBR) to handle these read-reclaim races
• Definition: A quiescent state for a thread T is a state in which T holds no references to any shared data
• Definition: A grace period is an interval in which every thread has passed through at least one quiescent state
• QSBR 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
Illustration

Element removed at this point

Element can be reclaimed after this point

Grace Period

Thread 1

Thread 2

Thread 3

Time
How to define Quiescent States?

- Application dependent!
- For OS kernels, some natural ones exist
  - Assume that references to RCU data structures are only held within read or write critical sections
    - Read critical section: thread reads an RCU-protected data structure
    - Write critical section: thread writes an RCU-protected data structure
  - Assume that read critical sections do not block
    - i.e., No context switch occurs within a read-side critical section
- Then, a context switch is a quiescent state
  - No references are held across a context switch
Reader-Side Quiescence Primitives: Read Lock/Unlock

/* definitions */
struct foo {
    int a;
};

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

T1 (Writer):
    p = malloc(sizeof(*p));
    p->a = 1;
    rcu_assign_pointer(gp, p);
    // when can we free(p)?

T2 (Reader):
    rcu_read_lock(); // notice, no lock var
    p = rcu_dereference(gp);
    if (p != NULL)
        use(p->a);
    rcu_read_unlock();

lock/unlock do not spin or block!

• rcu_read_lock/unlock disable context switch within read-side critical section
• Write can detect that read is in progress (reader is not quiescent) and does not delete data that is being accessed by reader
Writer-Side Quiescence Primitive:
Synchronize RCU

- synchronize_rcu()
  - Wait until all pre-existing RCU read-side critical sections complete
- Implementation:

```c
synchronize_rcu() {
    for_each_online_cpu(cpu)
        run_on(cpu); // runs the current thread on cpu
}
```

- synchronize_rcu() runs the current thread on all CPUs
  - Forces context switches on each of the CPUs
  - Ensures that it waits for the grace period
RCU Synchronization

rcu_dereference()

Reader → rcu_assign_pointer() → Writer

rcu_read_lock(), rcu_read_unlock()

Quiescent State → synchronize_rcu()
// Reader traverses
// a linked list
rcu_read_lock();
// next line uses
// rcu_dereference
hlist_for_each_entry_rcu(p, q, head, list) {
    // p is a linked
    // list node
    do_something(p->value);
}
rcu_read_unlock();

// Writer searches and updates
// a list element
p = search(head, key);
if (p == NULL) {
    /* unlock and return. */
}
q = kmalloc(sizeof(*p), GFP_KERNEL);
*q = *p; // read and copy
q->value = ...;
// atomically replace p with q
// next line uses rcu_assign_pointer
list_replace_rcu(&p->list, &q->list);
// wait for grace period
synchronize_rcu();
// free previous version
kfree(p);
PPC Hash Table with RCU

![Graph showing searches per unit time normalized to ideal vs. number of CPUs for different approaches such as "ideal", "RCU", "HPBR", "spinbkt", "brlock", and "globalrw".]
Growth of RCU Use in Linux

…but Still Small in Comparison

When to Use Which Tool?

• Read-mostly situations
  • If algorithm can handle concurrent reads + single updater: RCU

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

• When you only have a hammer, everything looks like a nail

• It’s good to have lots of tools in your toolbox!
Some Resources

- LWN article on lockless algorithms
  https://lwn.net/Kernel/Index/#Lockless_algorithms

- Load dependent ordering behavior in Alpha:
  http://www.cs.umd.edu/~pugh/java/memoryModel/AlphaReordering.html

- An excellent book on multi-processor synchronization and lockless algorithms: The art of multiprocessor programming by Maurice Herlihy & Nir Shavit