Lecture 19:

Fine-grained synchronization & lock-free data structures

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2014
Tunes

“Fly so Free”
Luella and the Sun

“Well, we wanted to call it fly-so lock free, but then everyone in the music industry would think we were dorks.”

- Melissa Mathes
Example: a sorted linked list

```c
struct Node {
  int value;
  Node* next;
};

struct List {
  Node* head;
};

void insert(List* list, int value) {
  Node* n = new Node;
  n->value = value;

  // assume case of inserting before head of
  // of list is handled here (keep code simple)

  Node* prev = list->head;
  Node* cur = list->head->next;

  while (cur) {
    if (cur->value > value)
      break;

    prev = cur;
    cur = cur->next;
  }

  prev->next = n;
  n->next = cur;
}

void delete(List* list, int value) {
  // assume case of deleting first element is
  // handled here (keep code simple)

  Node* prev = list->head;
  Node* cur = list->head->next;

  while (cur) {
    if (cur->value == value) {
      prev->next = cur->next;
      delete cur;
      return;
    }

    prev = cur;
    cur = cur->next;
  }

  return;
}
```

What can go wrong if multiple threads operate on the linked list simultaneously?
Example: simultaneous insertion

Thread 1 attempts to insert 6
Thread 2 attempts to insert 7

Thread 1:
Example: simultaneous insertion

Thread 1 attempts to insert 6
Thread 2 attempts to insert 7

Thread 1:

Thread 2:

Thread 1 and thread 2 both compute same prev and cur. Result: one of the insertions gets lost!

Result: (assuming thread 1 updates prev->next before thread 2)
Solution 1: protect the list with a single lock

```c
struct Node {
    int value;
    Node* next;
};

struct List {
    Node* head;
    Lock lock;
};

void insert(List* list, int value) {
    Node* n = new Node;
    n->value = value;

    lock(list->lock);

    // assume case of inserting before head of
    // of list is handled here (keep code simple)
    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value > value)
            break;

        prev = cur;
        cur = cur->next;
    }

    prev->next = n;
    n->next = cur;

    unlock(list->lock);
}

void delete(List* list, int value) {
    lock(list->lock);

    // assume case of deleting first element is
    // handled here (keep code simple)
    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            unlock(list->lock);
            return;
        }

        prev = cur;
        cur = cur->next;
    }

    prev = cur;
    cur = cur->next;

    unlock(list->lock);
}
```

Per list lock
Single global lock per data structure

- **Good:**
  - It is relatively simple to implement correct mutual exclusion for data structure operations

- **Bad:**
  - Operations on the data structure are serialized
  - May limit parallel application performance
Challenge: who can do better?

```c
struct Node {
  int value;
  Node* next;
};

struct List {
  Node* head;
};

void insert(List* list, int value) {
    Node* n = new Node;
    n->value = value;
    // assume case of inserting before head of
    // of list is handled here (keep code simple)
    Node* prev = list->head;
    Node* cur = list->head->next;
    while (cur) {
        if (cur->value > value)
            break;
        prev = cur;
        cur = cur->next;
    }
    prev->next = n;
    n->next = cur;
}

void delete(List* list, int value) {
    // assume case of deleting first element is
    // handled here (keep code simple)
    Node* prev = list->head;
    Node* cur = list->head->next;
    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            return;
        }
        prev = cur;
        cur = cur->next;
    }
}
```

3 → 5 → 10 → 11 → 18
**Solution 2: fine-grained locking**

```c
struct Node {
    int value;
    Node* next;
    Lock* lock;
};
struct List {
    Node* head;
    Lock* lock;
};

void insert(List* list, int value) {
    Node* n = new Node;
    n->value = value;

    // assume case of insert before head handled
    // here (keep code simple)
    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;
    cur = list->head->next;

    lock(prev->lock);
    if (cur)
        lock(cur->lock);
    unlock(list->lock);

    while (cur)
    {
        if (cur->value > value)
            break;

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur)
            lock(cur->lock);
    }

    prev->next = n;
    n->next = cur;

    unlock(prev->lock);
    unlock(cur->lock);
}

void delete(List* list, int value) {
    // assume case of delete head handled here
    // (keep code below simple)
    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;
    cur = list->head->next;

    lock(prev->lock);
    if (cur)
        lock(cur->lock);
    unlock(list->lock);

    while (cur)
    {
        if (cur->value == value)
        {
            prev->next = cur->next;
            unlock(prev->lock);
            unlock(cur->lock);
            delete cur;
            return;
        }

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur)
            lock(cur->lock);
    }

    unlock(prev->lock);
}
```
“Hand-over-hand” locking

Thread 0: delete(11)
Hand-over-hand locking

Thread 0: delete(11)
Thread 1: delete(10)
Hand-over-hand locking

Thread 0: delete(11)
Thread 1: delete(10)
Hand-over-hand locking

Thread 0: delete(11)
Thread 1: delete(10)
Fine-grained locking

- Goal: enable parallelism in data structure operations
  - Reduces contention for global data structure lock
  - In previous linked-list example: a single monolithic lock is overly conservative (operations on different parts of the linked list can proceed in parallel)

- Challenge: tricky to ensure correctness
  - Determining when mutual exclusion is required
  - Deadlock? (how do you immediately know the prior linked-list code is deadlock free?)
  - Livelock?

- Costs?
  - Overhead of taking a lock each traversal step (extra instructions + traversal now involves memory writes)
  - Extra storage cost (a lock per node)
  - I bet you can imagine a middle-ground solution that trades off some parallelism for reduced overhead (hint: similar issue to selection of task granularity)
Practice exercise

- Implement a fine-grained locking implementation of a binary search tree supporting insert and delete

```c
struct Tree {
    Node* root;
};

struct Node {
    int value;
    Node* left;
    Node* right;
};

void insert(Tree* tree, int value);
void delete(Tree* tree, int value);
```
**Single reader, single writer bounded queue**

```c
struct Queue {
    int data[N];
    int head, tail;
};

void init(Queue* q) {
    q->head = q->tail = 0;
}

// return false if queue is full
bool push(Queue* q, int value) {
    if (q->tail == MOD_N(q->head - 1))
        return false;
    q.data[q->tail] = value;
    q->tail = MOD_N(q->tail + 1);
    return true;
}

// returns false if queue is empty
bool pop(Queue* q, int* value) {
    if (q->head != q->tail) {
        *value = q->data[q->head];
        q->head = MOD_N(q->head + 1);
        return true;
    }
    return false;
}
```

- Two threads (one producer, one consumer) accessing queue at the same time.
- Threads never synchronize or wait on each other
  - When queue is empty or full, in this implementation operations fail (return false)

* Assume a sequentially consistent memory system for now (or the presence of appropriate memory fences, or C++0x atomic<>*)
Single reader, single writer **unbounded queue** *  
  
```c
struct Node {
    Node* next;
    int value;
};

struct Queue {
    Node* head;
    Node* tail;
    Node* reclaim;
};

void init(Queue* q) {
    q->head = q->tail = q->reclaim = new Node;
}

void push(Queue* q, int value) {
    Node* n = new Node;
    n->next = NULL;
    n->value = value;
    q->tail->next = n;
    q->tail = q->tail->next;

    while (q->reclaim != q->head) {
        Node* tmp = q->reclaim;
        q->reclaim = q->reclaim->next;
        delete tmp;
    }
}

// returns false if queue is empty
bool pop(Queue* q, int* value) {
    if (q->head != q->tail) {
        *value = q->head->next->data;
        q->head = q->head->next;
        return true;
    }
    return false;
}
```

- Tail points to last element added
- Head points to element BEFORE head of queue
- Allocation and deletion performed by the same thread (producer)

* Assume a sequentially consistent memory system for now (or presence of appropriate memory fences, or C++0x atomic<>)

Source: Dr. Dobbs Journal
Single reader, single writer unbounded queue

push 3, push 10

pop (returns 3)

pop (returns 10)

pop (returns false... queue empty)

push 5 (triggers reclaim)
Blocking algorithms

- A blocking algorithm allows one thread to prevent other threads from completing operations on a shared data structure indefinitely.

- Example:
  - Thread 0 takes a lock on a node in our linked list.
  - Thread 0 is swapped out by the OS, or crashes, or is just really slow (takes a page fault), etc.
  - No other threads can complete operations on the data structure (although thread 0 is not actively making progress modifying it).

- An algorithm using locks is blocking regardless of whether the lock implementation uses spinning or preemption.
Non-blocking algorithms are lock-free if some thread is guaranteed to make progress ("systemwide progress")

- In lock-free case, it is not possible to preempt one of the threads at an inopportune time and prevent progress by rest of system
- Note: any one thread is allowed to starve by this definition
Lock-free stack (first try)

```
struct Node {
    Node* next;
    int value;
};

struct Stack {
    Node* top;
};

void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, Node* n) {
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

Node* pop(Stack* s) {
    while (1) {
        Node* old_top = s->top;
        if (old_top == NULL)
            return NULL;
        Node* new_top = old_top->next;
        if (compare_and_swap(&s->top, old_top, new_top) == old_top)
            return old_top;
    }
}
```

Main idea: as long as no other thread has modified the stack, a thread’s modification can proceed.

Note difference from fine-grained locks example earlier: before, implementation locked a part of a data-structure for fine-grained access. Here, threads do not hold lock on data-structure at all.

* Still assuming a sequentially consistent memory system for simplicity (or presence of appropriate memory fences, or use of C++0x’s atomic type modifier)
The ABA problem

Thread 0

begin pop() (local variable: old_top = A, new_top = B)

begin pop() (local variable old_top == A)

CAS succeeds (sets top to B!)
complete pop() (returns A)

Thread 1

begin push(A)
complete push(A)

begin push(D)
complete push(D)

modify A: e.g., set value = 42
begin push(A)
complete push(A)

Stack structure is corrupted! (lost D)
Lock-free stack using counter for ABA soln

```c
struct Node {
    Node* next;
    int value;
};

struct Stack {
    Node* top;
    int pop_count;
};

void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, Node* n) {
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

Node* pop(Stack* s) {
    while (1) {
        int pop_count = s->pop_count;
        Node* top = s->top;
        if (top == NULL)
            return NULL;
        Node* new_top = top->next;
        if (double_compare_and_swap(&s->top, top, new_top, &s->pop_count, pop_count, pop_count+1))
            return top;
    }
}
```

- Maintain counter of pop operations (technically a counter wrap-around issue still exists)
- Requires machine to support “double compare and swap” (DCAS) or doubleword CAS
- Could also solve ABA problem with node allocation and/or element reuse policies

* Still assuming a sequentially consistent memory system for simplicity

Test to see if either have changed (in this example: return true if no changes)
Compare and swap on x86

- x86 supports a “wide” compare and swap instruction
  - Not quite the “double compare-and-swap” used in the code on the previous slide
  - But could simply ensure the stack’s count and top fields are contiguous in memory to use the 64-bit wide single compare-and-swap instruction below.

- cmpxchg8b
  - “compare and exchange eight bytes”
  - compare-and-swap of 32-bit values

- cmpxchg16b
  - “compare and exchange 16 bytes”
  - compare-and-swap of 64-bit values
Lock-free linked list insertion

```c
struct Node {
  int value;
  Node* next;
};

struct List {
  Node* head;
};

// insert new node after specified node
void insert_after(List* list, Node* after, int value) {
  Node* n = new Node;
  n->value = value;

  // assume case of insert into empty list handled
  // here (keep code simple for class discussion)

  Node* prev = list->head;

  while (prev->next) {
    if (prev == after) {
      while (1) {
        Node* old_next = prev->next;
        n->next = old_next;
        if (compare_and_swap(&prev->next, old_next, n) == old_next)
          return;
      }
    }

    prev = prev->next;
  }
}
```

Compared to fine-grained locking implementation:

- No overhead of taking locks
- No per-node storage overhead
Lock-free linked list deletion

Supporting lock-free deletion significantly complicates data-structure
Consider case where B is deleted simultaneously with successful insertion of E after B.
B now points to E, but B is not in the list!

For the curious:
Harris 2001. A Pragmatic Implementation of Non-blocking Linked-Lists
Fomitchev 2004. Lock-free linked lists and skip lists
Lock-free vs. locks performance comparison

Lock-free algorithm run time normalized to run time of using pthread mutex locks

Source: Hunt 2011. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures
In practice: why lock free data-structures?

- When optimizing parallel programs in this class you often assume that only your program is using the machine
  - Because you care about performance
  - Typical assumption in scientific computing, graphics, data analytics, etc.

- In these cases, well written code with locks can be as fast (or faster) than lock-free code

- But there are situations where code with locks can suffer from tricky performance problems
  - Multi-programmed situations where page faults, pre-emption, etc. can occur while thread is in a critical section
  - Creates problems like priority inversion, convoying, crashing in critical section, etc. that are often discussed in OS classes
Summary

- Use fine-grained locking to reduce contention (maximize parallelism) in operations on shared data structures
  - But fine-granularity can increase code complexity (errors) and increase execution overhead

- Lock-free data structures: non-blocking solution to avoid overheads due to locks
  - But can be tricky to implement (ensuring correctness in a lock-free setting has its own overheads)
  - Still requires appropriate memory fences on modern relaxed consistency hardware

- Note: a lock-free design does not eliminate contention
  - Compare-and-swap can fail under heavy contention, requiring spins
More reading

- Michael and Scott 1996. Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
  - Multiple reader/writer lock-free queue


- Good blog posts and articles: