Systems I – CS 6013

Computer Architecture and Operating Systems

Lecture 29: Locked Data Structures

MASTER OF SOFTWARE DEVELOPMENT (MSD) PROGRAM

J. DAVISON DE ST. GERMAIN

SPRING 2024

(Adapted from Slides by Prof. Andrea Arpaci-Dusseau & Nima Honarmand, and previous MSD presentations)

Lecture 29 – Topics

- Locked Data Structures

Announcements

- Lisp – lots of insidious parentheses

Review

- Concurrency leads to non-deterministic results
  - Different results even with same inputs
  - We call this a **race condition**
  - **Data races** are race conditions involving *write contention.*
- We saw mutexes (locks) for protecting critical sections.
- We saw condition variables for more complicated thread operations.
  - One thread signaling another thread(s)
  - Condition variables + locks = the monitor pattern.
    - Monitoring a global condition.

Today

- Some practical applications of locks / mutexes

- We'll design (to work with Threads):
  - A simple counter
  - A concurrent (thread-safe) linked list
  - A concurrent queue
  - A concurrent hash table

Case Study 1: A Threaded Counter

```
int value;
void init(){
    value = 0;
}
void increment(){
    value++;
    // How many assembly
    // instructions?
    //     3
}
void decrement(){
    value--;
}
```

- Potential Race Conditions?
  - Everything except init has a race condition.
  - How would you fix these?
  - A simple approach: lock every operation that could have a race condition.

Case Study 1: A Concurrent Counter

```
int value;
std::mutex m;
void init(){
    value = 0;
}
void increment(){
    m.lock();
    value++;
    m.unlock();
}
void decrement(){
    m.lock();
    value--;
    m.unlock();
}
```

- Evaluation
  - This is indeed a safe counter.
  - However, is it actually faster to use multiple threads in this example?
    - Thread creation and locks have overhead.
    - This actually slows down as we throw more threads at it!
- This does illustrate the pitfalls of naive locking.
- The solution to this is nontrivial, and we won't go over it.
  - See textbook, Chapter 29, Approximate counter.

Case Study 2: A linked list [in C]

```
typedef struct __node_t {
    int data;
    struct __node_t * next;
} node_t;

typedef struct __list_it {
    node_t * head;
} list_t;

void init( list_t * L ){
    L->head = NULL;
}
```

```
int insert( list_t * L, int data ){
    node_t * new_node =
            malloc( sizeof( node_t ) );
    if( new_node == NULL ){
        return -1;
    }
    new_node->data = data;
    new_node->next = L->head;
    L->head = new_node;
    return 0;
}
```

- Difference from C++?
    - Constructor, functional vs OOP, etc. (see next slides)

C Syntax - Explanation

```
struct node_t {
  int             data;
  struct node_t * next;
};
```

- `'_t'` `<-` naming convention to say it is a type
- "`struct`" must be repeated every time you use node_t, so to create the head of the list:

- `struct node_t * head;`

```
typedef struct __node_t {
   int data;
   struct __node_t *next;
} node_t;
```

- `'__'` `<-` naming convention to say "you should never use/see this"
- `typedef` defines a type named `node_t;`
- `node_t * head;`

Case Study 2: A linked list [in C]

```
typedef struct __node_t {
    int data;
    struct __node_t *next;
} node_t;
```

- In C++, you would say:
  - `list.insert( data )`
- In C, you have to pass both the list, and the data into the insert function.

```
int insert( list_t * L, int data ){
    node_t * new_node =
            malloc( sizeof( node_t ) );
    if( new_node == NULL ){
        return -1;
    }
    new_node->data = data;
    new_node->next = L->head;
    L->head = new_node;
    return 0;
}
```

Case Study 2: A linked list [in C]

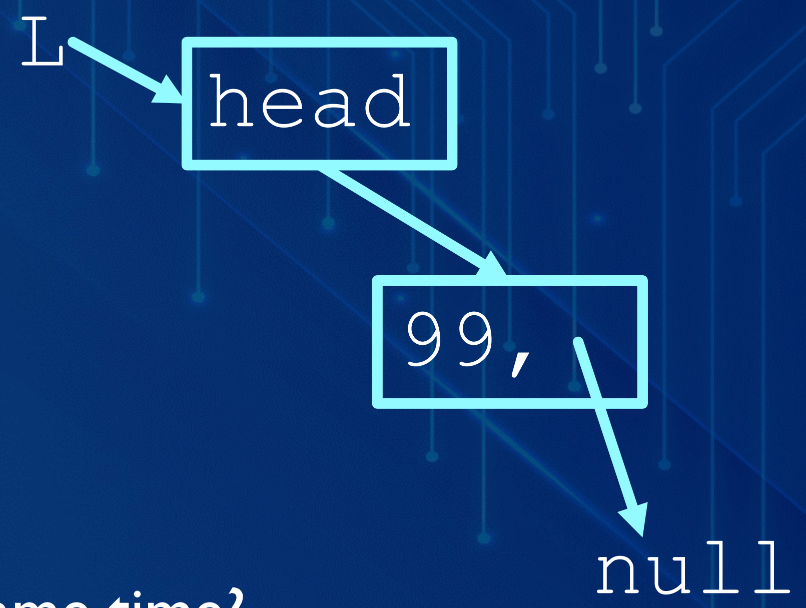
```
struct node_t { int       data;
                node_t * next; }

struct list_t { node_t * head; }

void insert( list_t * L, int data ){
   node_t * new_node =
           malloc( sizeof( node_t ) );
   new_node->data = data;
   new_node->next = L->head;
   L->head = new_node;
}
...
insert( list, 99 );
```

Draw the picture!!!

```
void init( list_t * L ){
   L->head = NULL;
}
```

Case Study 2: A linked list [in C]

```
void insert( list_t * L, int data ){
   node_t * new_node =
           malloc( sizeof( node_t ) );
   new_node->data = data;
   new_node->next = L->head;
   L->head = new_node;
}
```

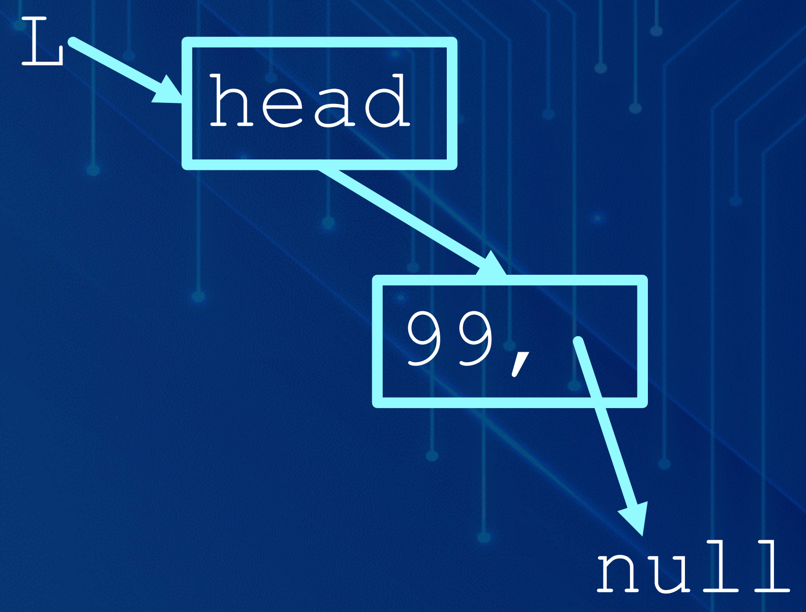


- Ask: What happens if multiple threads run this code at the same time?
  - Identify the race conditions.
  - Where should the mutex object live?
  - Where should I lock and unlock the mutex?
  - 1st attempt: don't want any of it interrupted – lock everything

Case Study 2: Concurrent List v2


```
void insert( list_t * L, int data ){
  lock( m );
   node_t new_node =
          malloc( sizeof( node_t ) );
   if( new_node == NULL ) {
      return;
   }
   new_node->data = data;
   new_node->next = L->head;
   L->head = new_node;
  unlock( m );
}
```

m.unlock();



- What is m? Where is it defined?
  - Mutex. Somewhere in the global scope so all threads can see it.

- What's missing?
  - How do we fix (avoid) needing this 2nd unlock?
  - Use a scoped lock!
  - But is there a better way?

Case Study 2: Concurrent List v3


```
void insert( list_t * L, int data ){
    lock( m ); // scoped lock
    node_t new_node =
            malloc( sizeof( node_t ) );
    if( new_node == NULL ) {
        return; // throw error?
    }
    new_node->data = data;
    new_node->next = L->head;
    L->head = new_node;
}
```

- Evaluation
  - This is thread-safe.
  - However, it is inefficient. There is a giant lock around every bit of code.
  - This is a coarse-grained lock.
  - A little reading reveals that malloc() is thread safe.
  - Then we don't need to lock malloc().
  - This allows a finer-grained lock.
  - What about the update of `new_node`? Does it need to be locked?
    - Is there a difference between the update of `data` and `next`?
    - `new_node` is a local (non-shared) variable.
    - However, `next` is referencing `L`, which is a global variable, so must be locked.
    - (See next slide for updated version of code)

Case Study 2: Concurrent List v3


```
void insert( list_t * L, int data ){

    node_t new_node =
           malloc( sizeof( node_t ) );
    if( new_node == NULL ) {
      return; // throw error?
    }
    new_node->data = data;
    lock( m );
    new_node->next = L->head;
    L->head = new_node;
    unlock( m );
}
```

- Performance ramifications?
  - Better because less code is locked
- Note, could actually use a scoped lock in this code too (in which case we would not need the unlock at the end)

Expanding List Functionality

- Let's see another linked-list example for locking.
- Next, we will write a function (`lookup`) to search our linked list.
  - We'll return return false if the item is not found, true if it is found.
- We can write a concurrent version right away if we use a coarse-grained lock.

Case Study 3: Concurrent List Lookup


```
bool lookup( list_t * L, int data ){

    m.lock();
    node_t * curr = L->head;
    while( curr ){
        if( curr->data == data ) {
            m.unlock();
            return true; // success
        }
        curr = curr -> next;
    }
    m.unlock();
    return false; // failure
}
```

- Why do we need to lock the list during a lookup?
    - List could be changed out from under us.
- Why the 2 unlocks?
    - 2 paths out of the function – must remember to unlock in both paths.
- Why is this a really bad way to lock this function?
    - Here, we're locking the whole list before we go through it - a coarse-grained locking strategy.
- What would a fine-grained strategy look like?
    - Think about what actually needs a lock…

Hand-over-Hand Locking

- Lock each node as we traverse the list.
- When done with a node, unlock it and lock the next one in turn.
- If locks can be implemented with low overhead, this is great.
- In practice, this is too fine-grained and will be expensive.
- Lock once every few times? Possible.

Case Study 4: Michael and Scott Two-Lock Concurrent Queue

- This is the algorithm that you must use for your assignment.
- Represent the queue as a singly-linked list with head *and* tail pointers (two ends to the list).
    - *Always want enqueue and dequeue to be O(1) operations.
- Use a single lock for enqueue, and a separate lock for dequeue.
- Head always points to a dummy node. Dummy node is the first node in the list.
- Nodes are inserted after the last node of the list.
- Nodes are deleted from the beginning of the list.

Case Study 4: Michael and Scott Two-Lock Concurrent Queue

- Initially, both head and tail point to a dummy node.
- If we didn't have the dummy variable, both head and tail could point to the same node.
  - This would cause problems if enqueue and dequeue occurred simultaneously.
  - Two locks + two threads = potential for "deadlock" or "livelock".
    - More later.
- As we enqueue stuff, we update tail pointer.
- As we dequeue stuff, we update the head pointer.

Case 4: Michael/Scott Concurrent Queue

```
struct node_t { int       data;
                node_t * next; }

struct queue_t { node_t * head;
                 node_t * tail;
                 mutex head_m,
                       tail_m; }

void init( queue_t * q ) {
    node_t * tmp = malloc(…);
    tmp->next = NULL;
    q->head = tmp;
    q->tail = tmp;
}
```

```
void enqueue( queue_t * q, int value ) {
    node_t * tmp = malloc( … );
    tmp->data = value;
    tmp->next = NULL;

    tail_m.lock();
    q->tail->next = tmp;
    q->tail = tmp;
    tail_m.unlock();
}
```

- Why have a separate tail and head lock?
  - Allows enqueue and dequeue to happen simultaneously!
  - Almost always, the head and tail are not next to each other.

Case 4: Michael/Scott Concurrent Queue


```
struct node_t { int      data;
                node_t * next; }

struct queue_t { node_t * head;
                 node_t * tail;
                 mutex head_m,
                       tail_m; }

void init( queue_t * q ) {
   node_t * tmp = malloc(…);
   tmp->next = NULL;
   q->head = tmp;
   q->tail = tmp;
}
```

```
bool dequeue( queue_t * q, int * value ) {
   head_m.lock();
   node_t * tmp = q->head; // dummy node
   node_t * new_head = tmp->next;
   if( new_head == NULL ){
       head_m.unlock();
       return false; // Nothing in queue
   }
   *value = new_head->data;
   q->head = new_head;
   free( tmp );
   head_m.unlock();
   return true;
}
```

Evaluation

- Enqueue is a nice example of fine-grained locking.
- The dummy node is a great example of how serial data structures must be modified for concurrency
  - …beyond just adding locks.
- Michael and Scott show that two-lock queue scales very well and avoids deadlock / livelock.

- If you use scoped locks, the lock will automatically unlock when it goes out of scope.
  - Use curly braces { } to define scopes.
  - Allows for unlock even after returning from middle of function!

Case Study 5: Concurrent Hash Table

- Final example: a hash table without resizing.
- We'll use separate chaining instead of probing.
  - Recall that this means an array of linked lists.
  - Each "bucket" in the table is now a linked list.
  - Collisions are resolved by hashing to the same bucket, then growing the corresponding linked list.
- We can use our concurrent list from earlier!
  - `list_t table[ NUM_BUCKETS ].`
- Hash table insert is just a call to the list_t insert function.
- Lookup is a call to the list_t lookup.

Evaluation

- The concurrent hash table scales well.
- Far better than the linked list does.
- Constant time roughly maintained as we increase the number of threads.

Summary

- Coarse-grained locking (locking large chunks of code) is easy, but inefficient.
- Fine-grained locking
  - trickier to get right without deadlock/livelock
  - may need to redesign data structure or code
  - may still scale poorly
- Enabling concurrency doesn't always improve performance!
  - Sometimes all we can do is hope it doesn't degrade it.
- Next week: concurrency bugs, lock-free data structures

~ *Fin* ~