TCSS 422: OPERATING SYSTEMS

Lock-based Data structures

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

FEEDBACK FROM 4/20/2017

- What are synchronization primitives?
- Synchronization primitives are the datatypes provided by the Thread API to support the coordination of shared memory between threads
- Memory is "synchronized" so that no two threads change a shared variable at the same time
- Absence of synchronization results in programs with unpredictable / non-deterministic behavior
- Locks: pthread_mutex_t, Condition variable: pthread_cond_t, Semaphore: sem_t
- How do they work?
  - Primitives are the datatypes used by pthread locks, wait, and signals, which support thread coordination
  - Locks are built using atomic HW (CPU) instructions to ensure mutual exclusion for critical sections of code which modify shared data

FEEDBACK - 2

- What is a dead lock?
  - A deadlock results when a pthread is blocked waiting for a lock that is never released, or for a signal which is missed or never sent.
  - The program typically freezes unexpectedly, and does not finish
  - Example: Test-and-Set spin lock implementation
- Does the C pseudocode implementation for spin locks translate to one line of assembly, or is it multiple lines?
- Is the C pseudocode implementation for spin locks atomic?

FEEDBACK - 3

- What is the purpose of a sloppy counter?
  - The sloppy counter provides a more performant shared counter data structure
  - The sloppy counter implementation trades accuracy for speed
- If you need an atomic instruction, have researchers tried to implement locking in hardware? If you need an atomic instruction to implement test-and-set for example, it seems like a CPU manufacturer could add some instruction to their instruction set architecture (ISA) on multicore CPUs.
  - HW approach to locking is to avoid them with: ***lock-free data structures***
  - Data structures implemented using atomic CPU instructions

LOCK-FREE DATA STRUCTURES

- Lock-free data structures in Java
- Java.util.concurrent.atomic package
- Classes:
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicIntegerFieldUpdater
  - AtomicLong
  - AtomicLongArray
  - AtomicLongFieldUpdater
  - AtomicReference
- See: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html

FEEDBACK: THREAD POOLS VS. QUEUES

- What is the difference between a ***thread pool*** and a ***thread queue***?
- **Thread pool:**
  - Group of pre-instantiated, idle threads which stand ready for work.
  - Pay for initialization time just once, reduce overhead
  - Recycle threads for future use by returning to the pool when work completes
  - Excellent design for programs with a large number of short tasks
  - Less applicable for programs with a small number of long tasks

THREAD QUEUES

- Locks with ***thread queues*** offer an alternative to spin locks
- Used to track which threads are waiting to acquire the lock
- **Goal**: improve fairness for acquiring locks
- **Lock with thread queue:**
  - Spin briefly to acquire outer guard lock
  - **Park()** thread if main lock is unavailable
  - When lock is released a thread is removed from the queue using unpark() in a first-in first-out (FIFO) manner
  - Without Queues/FIFO, there is no guarantee of fairness for which thread will acquire the lock next compared to standard unlock()

THREAD QUEUES - 2

- Improves overall fairness for sharing of locks
- Fairness improved due to FIFO nature of Queues
- Standard HW/OS lock()/unlock() implementation doesn't guarantee fairness
  - Thread acquiring the lock next is left to chance
- Programmer using thread queues is allowed control
  - Ensure fairness, prevents starvation
  - Prioritize which thread should acquire lock next
  - Better approach for synchronizing large #'s of threads
- See chapter 28, section 14…

THREAD QUEUES - 3

- Requires OS support to add/remove threads to/from queue(s)
- Solaris API:
  - park(): adds thread to queue, puts thread to sleep
  - unpark(threadID): removes next thread from queue, lock is passed to awoken thread
- Linux API: implemented with futex()

LOCK BASED DATA STRUCTTURES

OBJECTIVES

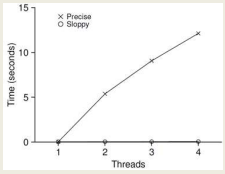
- Chapter 29
  - Concurrent Data Structures
  - Performance
  - Lock Granularity

LOCK-BASED CONCURRENT DATA STRUCTURES

- Adding locks to data structures make them **thread safe**.
- Considerations:
  - Correctness
  - Performance
  - Lock granularity

CONCURRENT COUNTERS - PERFORMANCE

- iMac: four core Intel 2.7 GHz i5 CPU
- Each thread increments counter 1,000,000 times



Traditional vs. sloppy counter
Sloppy Threshold (S) = 1024

Synchronized counter scales poorly.

CONCURRENT LINKED LIST - 1

- Simplification - only basic list operations shown
- Structs and initialization:

```
1       // basic node structure
2       typedef struct __node_t {
3               int key;
4               struct __node_t *next;
5       } node_t;
6
7       // basic list structure (one used per list)
8       typedef struct __list_t {
9               node_t *head;
10              pthread_mutex_t lock;
11      } list_t;
12
13      void List_Init(list_t *L) {
14              L->head = NULL;
15              pthread_mutex_init(&L->lock, NULL);
16      }
17
(Cont.)
```


CONCURRENT LINKED LIST - 2

- Insert – adds item to list
- Everything is critical!
  - There are two unlocks

```
(Cont.)
18      int List_Insert(list_t *L, int key) {
19              pthread_mutex_lock(&L->lock);
20              node_t *new = malloc(sizeof(node_t));
21              if (new == NULL) {
22                      perror("malloc");
23                      pthread_mutex_unlock(&L->lock);
24              return -1; // fail
26              new->key = key;
27              new->next = L->head;
28              L->head = new;
29              pthread_mutex_unlock(&L->lock);
30              return 0; // success
31      }
(Cont.)
```


CONCURRENT LINKED LIST - 3

- Lookup – checks list for existence of item with key
- Once again everything is critical
  - Note - there are also two unlocks

```
(Cont.)
32
32      int List_Lookup(list_t *L, int key) {
33              pthread_mutex_lock(&L->lock);
34              node_t *curr = L->head;
35              while (curr) {
36                      if (curr->key == key) {
37                              pthread_mutex_unlock(&L->lock);
38                              return 0; // success
39                      }
40                      curr = curr->next;
41              }
42              pthread_mutex_unlock(&L->lock);
43              return -1; // failure
44      }
```


CONCURRENT LINKED LIST

- First Implementation:
  - Lock **everything** inside Insert() and Lookup()
  - If malloc() fails lock must be released
    - Research has shown "***exception-based control flow***" to be error prone
    - 40% of Linux OS bugs occur in rarely taken code paths
    - Unlocking in an exception handler is considered a poor coding practice
    - There is nothing specifically wrong with this example however
- Second Implementation …

CCL – SECOND IMPLEMENTATION

- Init and Insert

```
1       void List_Init(list_t *L) {
2               L->head = NULL;
3               pthread_mutex_init(&L->lock, NULL);
4       }
5
6       void List_Insert(list_t *L, int key) {
7               // synchronization not needed
8               node_t *new = malloc(sizeof(node_t));
9               if (new == NULL) {
10                      perror("malloc");
11                      return;
12              }
13              new->key = key;
14
15              // just lock critical section
16              pthread_mutex_lock(&L->lock);
17              new->next = L->head;
18              L->head = new;
19              pthread_mutex_unlock(&L->lock);
20      }
21
```

CCL – SECOND IMPLEMENTATION - 2

- Lookup

```
(Cont.)
22      int List_Lookup(list_t *L, int key) {
23              int rv = -1;
24              pthread_mutex_lock(&L->lock);
25              node_t *curr = L->head;
26              while (curr) {
27                      if (curr->key == key) {
28                              rv = 0;
29                              break;
30                      }
31                      curr = curr->next;
32              }
33              pthread_mutex_unlock(&L->lock);
34              return rv; // now both success and failure
35      }
```


KEYS TO ADDING LOCKS TO DATA STRUCTURE

- How many locks to add?
- Where should we lock? Unlock?
- If lock is broad (e.g. lock many lines), concurrency is restricted. More waiting…
- If lock is narrow (e.g. few lines), execution with multiple threads can be more concurrent. Less waiting…

CONCURRENT LINKED LIST PERFORMANCE

- Using a single lock for entire list is not very performant
- Users must "wait" in line for a single lock to access/modify any item
- Hand-over-hand-locking (lock coupling)
  - Introduce a lock for each node of a list
  - Traversal involves handing over previous node's lock, acquiring the next node's lock…
  - Improves lock granularity
  - Degrades traversal performance
- Consider hybrid approach
  - Fewer locks, but more than 1
  - Best lock-to-node distribution?

MICHAEL AND SCOTT CONCURRENT QUEUES

- Improvement beyond a single master lock for a queue (FIFO)
- Two locks:
  - One for the **head** of the queue
  - One for the **tail**
- Synchronize enqueue and dequeue operations
- Add a dummy node
  - Allocated in the queue initialization routine
  - Supports separation of head and tail operations
- Items can be added and removed by separate threads at the same time

CONCURRENT QUEUE

- Remove from queue

```
1       typedef struct __node_t {
2               int value;
3               struct __node_t *next;
4       } node_t;
5
6       typedef struct __queue_t {
7               node_t *head;
8               node_t *tail;
9               pthread_mutex_t headLock;
10              pthread_mutex_t tailLock;
11      } queue_t;
12
13      void Queue_Init(queue_t *q) {
14              node_t *tmp = malloc(sizeof(node_t));
15              tmp->next = NULL;
16              q->head = q->tail = tmp;
17              pthread_mutex_init(&q->headLock, NULL);
18              pthread_mutex_init(&q->tailLock, NULL);
19      }
20
(Cont.)
```


CONCURRENT QUEUE - 2

- Add to queue

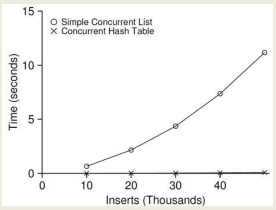
```
(Cont.)
21      void Queue_Enqueue(queue_t *q, int value) {
22              node_t *tmp = malloc(sizeof(node_t));
23              assert(tmp != NULL);
24
25              tmp->value = value;
26              tmp->next = NULL;
27
28              pthread_mutex_lock(&q->tailLock);
29              q->tail->next = tmp;
30              q->tail = tmp;
31              pthread_mutex_unlock(&q->tailLock);
32      }
(Cont.)
```

CONCURRENT HASH TABLE

- Consider a simple hash table
  - Fixed (static) size
  - Hash maps to a bucket
    - Bucket is implemented using a concurrent linked list
    - One lock per hash (bucket)
    - Hash bucket is a linked lists

INSERT PERFORMANCE – CONCURRENT HASH TABLE

- Four threads – 10,000 to 50,000 inserts
  - iMac with four-core Intel 2.7 GHz CPU



The simple concurrent hash table scales magnificently.

CONCURRENT HASH TABLE

```
1       #define BUCKETS (101)
2
3       typedef struct __hash_t {
4               list_t lists[BUCKETS];
5       } hash_t;
6
7       void Hash_Init(hash_t *H) {
8               int i;
9               for (i = 0; i < BUCKETS; i++) {
10                      List_Init(&H->lists[i]);
11              }
12      }
13
14      int Hash_Insert(hash_t *H, int key) {
15              int bucket = key % BUCKETS;
16              return List_Insert(&H->lists[bucket], key);
17      }
18
19      int Hash_Lookup(hash_t *H, int key) {
20              int bucket = key % BUCKETS;
21              return List_Lookup(&H->lists[bucket], key);
22      }
```


QUESTIONS