TCSS 422: OPERATING SYSTEMS

Locks

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

FEEDBACK FROM 1/25/2017

- Makefile question:
- What is the "all" target?
  "make all" - builds all targets defined in the makefile
  target: a keyword that invokes a specific build (e.g. path in the makefile) to compile and link a particular set of files
- What is the "clean" target?
  Clean deletes all executable and link files leaving only source files behind.
- Pthread_create(), pthread_join() example– can you explain it again?

FEEDBACK - 2

- Why wouldn't we want an atomic update instruction for a B-tree (Ch.26, section 5, page 10)

  "would we really want the hardware to support an 'atomic update of B-tree' instruction? Probably not, at least in a sane instruction set

  - Here the author is pointing out that CPU designers can get carried away and have too many specialized instructions to do compound operations.
  - Complex instructions goes against CPU instruction set design principles
  - Regular, simple, compact, easy to program, implement
  - Impacts compiler design and complexity

FEEDBACK - 3

- How does test and set work?
  - What is the advantage provided by testing and setting the old value?
  - Before we assume we have the lock, we test if the lock wasn't already held.
  - In contrast to basic spin lock which just assumes that setting the lock int to 1 always works

FEEDBACK - 4

- Explain what lock initialization does… (manpage)
  - pthread_mutex_init: initializes the mutex object pointed to by mutex according to the mutex attributes specified in mutexattr.
    If mutexattr is NULL, default attributes are used instead.
  - The LinuxThreads implementation supports only one mutex attribute, the mutex kind, which is either ``fast'', ``recursive'', or ``error checking''. The kind of a mutex determines whether it can be locked again by a thread that already owns it. The default kind is ``fast''.
  - See pthread_mutexattr_init(3) for more information on mutex attributes.

OBJECTIVES

- Spin Locks - review
- Yielding
- Queues and User Control

FETCH-AND-ADD

- HW CPU Instruction
- Increment counter atomically-*as a unit* in one instruction

```
1   int FetchAndAdd(int *ptr) {
2       int old = *ptr;
3       *ptr = old + 1;
4       return old;
5   }
```

  - Fetch and return value
  - Increment by 1

TICKET LOCK

- Can build Ticket Lock using Fetch-and-Add
- Ensures progress of all threads (fairness)

```
1   typedef struct __lock_t {
2       int ticket;
3       int turn;
4   } lock_t;
5
6   void lock_init(lock_t *lock) {
7       lock->ticket = 0;
8       lock->turn = 0;
9   }
10
11  void lock(lock_t *lock) {
12      int myturn = FetchAndAdd(&lock->ticket);
13      while (lock->turn != myturn)
14              ; // spin
15  }
16  void unlock(lock_t *lock) {
17      FetchAndAdd(&lock->turn);
18  }
```


TICKET LOCK - 2

```
1   typedef struct __lock_t {
2       int ticket;
3       int turn;
4   } lock_t;
5
6   void lock_init(lock_t *lock) {
7       lock->ticket = 0;
8       lock->turn = 0;
9   }
10
11  void lock(lock_t *lock) {
12      int myturn = FetchAndAdd(&lock->ticket);
13      while (lock->turn != myturn)
14              ; // spin
15  }
16  void unlock(lock_t *lock) {
17      FetchAndAdd(&lock->turn);
18  }
```

TB while (1 != 1) **acquire lock**

TB myturn=1 ticket=2 turn=0

TA myturn=0 ticket=1 turn=0

TA while (0 != 0) **acquire lock**

TB while (0 != 1) **spin**

TA-unlock myturn=0 ticket=2 turn=1

YIELD() – SYSTEM CALL

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

- Give up the CPU – instead of busy waiting…
  - running →ready
- Ready relinquishes the CPU for another thread (ctxt. switch)
- How does the thread get the CPU back?
  - OS must opportunistically reschedule it: ready → running

HARDWARE SPIN LOCKS - SUMMARY

- Simple, correct
- Slow
- With long locks, waiting threads spin for entire timeslice
  - Repeat comparison continuously
  - Busy waiting

How To Avoid *Spinning*?
Need both HW & OS Support !

THREAD QUEUES

- Don't allow the OS to control your program
  - Use internal **Thread Queues**
- Allows programmer to maintain control
  - Ensure fairness, prevent starvation
  - Better for synchronizing large #'s of threads
- Require OS support to add/remove threads to/from queue(s)
- Solaris API:
  - park(): puts thread to sleep
  - unpark(threadID): wakes specified thread
- Linux API: futex()

THREAD QUEUES - 2

```
1   typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3   void lock_init(lock_t *m) {
4       m->flag = 0;
5       m->guard = 0;
6       queue_init(m->q);
7   }
8
9   void lock(lock_t *m) {
10      while (TestAndSet(&m->guard, 1) == 1)
11          ; // acquire guard lock by spinning
12      if (m->flag == 0) {
13          m->flag = 1; // lock is acquired
14          m->guard = 0;
15      } else {
16          queue_add(m->q, gettid());
17          m->guard = 0;
18          park();
19      }
20  }
21  …
```

Guard uses a spin-lock to protect the critical sections in lock() and unlock()

Obtain guard lock

try to obtain actual lock

lock unavailable; add thread to queue

potential wakeup/waiting race

THREAD QUEUES - 3

- Unlock

```
22  void unlock(lock_t *m) {
23      while (TestAndSet(&m->guard, 1) == 1)
24          ; // acquire guard lock by spinning
25      if (queue_empty(m->q))
26          m->flag = 0; // let go of lock; no one wants it
27      else
28          unpark(queue_remove(m->q)); // hold lock (for next thread!)
29      m->guard = 0;
30
```

Obtain guard lock (spin)

wake up thread from queue

release guard lock

- Note: no change to m->flag if unparking a thread
- Lock is passed to the unparked thread "directly"

WAKEUP/WAITING RACE

- Thread B: context switch occurs immediately before call to park()
- Thread A: releases lock, calls unpark, queue is empty
- Thread B: regains context, proceeds to lock itself forever

- Need new system call
  - **setpark()-** informs OS about soon to be parked thread
  - Subsequent calls to unpark() are aware that ThreadB is about to park
  - ThreadB's call to park() immediately returns

FUTEX

- **F**ast **U**serspace Mu**TEX**
- Linux futex system calls similar to park() and unpark()
- Linux uses an in-kernel queue
- Provides a futex() system call

- Provides atomic-*as a unit* compare-and-block operation

- **Futex is a lower-level construct**
- Used as building blocks for:
  mutex, condition variables, semaphores

- Objective: reduce the number of system calls

FUTEX: WRITE YOUR OWN MUTEX LOCK

- futex_wait(addr, expected)
  - Put calling thread to sleep
  - If value @ addr != expected → return immediately
- futex_wake(addr)
  - Wake one thread that is waiting on the queue

- These are not exposed as C library calls directly
  - Call futex() with FUTEX_WAIT or FUTEX_WAKE

- Use a 32-bit integer
  - The leftmost bit (the +/- sign) tracks the lock state
    - 0 – free
    - 1 – locked
  - Remaining 31 bits: identifies thread

FUTEX: MUTEX_LOCK PSUEDO CODE

```
Void mutex_lock(int *mutex) {
    int v;
    /* Bit 31 was clear, we got the mutex (this is a fast lock!)
    if (atomic_bit_test_set (mutex, 31) == 0)
        return;
    // "adds" mutex to queue
    atomic_increment (mutex);
    while (1)  {
        // is lock available?
        if (atomic_bit_test_set (mutex, 31) ==0 {
            // remove mutex from queue – it has the lock now
            atomic_decrement (mutex);
            return;
        }
        // Have to wait. Make sure futex value is locked (negative)
        v = *mutex;
        iv (v >= 0)
          continue;
        // wait to be woken up when lock is available
        // this is not a spin lock… (signal)
        futex_wait (mutex, v);
    }
}
```

FUTEX: MUTEX UNLOCK PSUEDO CODE

```
void mutex_unlock(int *mutex) {

    // Adding 0x80000000 to counter results in 0 if and only if
    // there are no other interested threads

    if (atomic_add_zero (mutex, 0x80000000))
        return;

    // There are other threads waiting for this lock (mutex)
    // wake one of them up..
    // (e.g. dequeue it)
    futex_wake (mutex);

}
```

- Interesting note: Futex bug in Redhat Linux
- https://www.infoq.com/news/2015/05/redhat-futex

HYBRID - TWO PHASE LOCKS

- Hybrid between spin-locks and yielding
- Useful if lock is about to be released
- First phase – spin lock
  - Spin for some time waiting for the lock to be released
  - If lock is not acquired after time expires enter phase two.
- Second phase - yield
  - Thread sleeps (yields)
  - Is awoken when the lock becomes free

LOCK BASED DATA STRUCTTURES

OBJECTIVES

- 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

COUNTER STRUCTURE W/O LOCK

- Synchronization weary --- not thread safe

```
1       typedef struct __counter_t {
2               int value;
3       } counter_t;
4
5       void init(counter_t *c) {
6               c->value = 0;
7       }
8
9       void increment(counter_t *c) {
10              c->value++;
11      }
12
13      void decrement(counter_t *c) {
14              c->value--;
15      }
16
17      int get(counter_t *c) {
18              return c->value;
19      }
```

CONCURRENT COUNTER

```
1       typedef struct __counter_t {
2               int value;
3               pthread_lock_t lock;
4       } counter_t;
5
6       void init(counter_t *c) {
7               c->value = 0;
8               Pthread_mutex_init(&c->lock, NULL);
9       }
10
11      void increment(counter_t *c) {
12              Pthread_mutex_lock(&c->lock);
13              c->value++;
14              Pthread_mutex_unlock(&c->lock);
15      }
16
```

- Add lock to the counter
- Require lock to change data

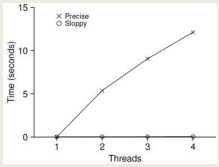
CONCURRENT COUNTER - 2

- Decrease counter
- Get value

```
(Cont.)
17      void decrement(counter_t *c) {
18              Pthread_mutex_lock(&c->lock);
19              c->value--;
20              Pthread_mutex_unlock(&c->lock);
21      }
22
23      int get(counter_t *c) {
24              Pthread_mutex_lock(&c->lock);
25              int rc = c->value;
26              Pthread_mutex_unlock(&c->lock);
27              return rc;
28      }
```


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.

PERFECT SCALING

- Achieve (N) performance gain with (N) additional resources
- Throughput:
- Transactions per second
- 1 core
- N = 100 tps
- 10 core
- N = 1000 tps

SLOPPY COUNTER

- Provides single logical shared counter
  - Implemented using local counters for each ~CPU core
    - 4 CPU cores = 4 local counters & 1 global counter
    - Local counters are synchronized via local locks
  - Global counter is updated periodically
    - Global counter has lock to protect global counter value
    - Sloppiness threshold (S): Update threshold of global counter with local values
    - Small (S): more updates, more overhead
    - Large (S): fewer updates, more performant, less synchronized
- Why this implementation?
  Why do we want counters local to each CPU Core?

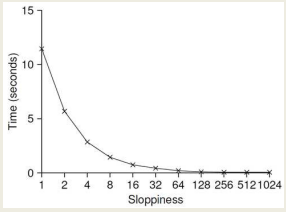
SLOPPY COUNTER - 2

- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

THRESHOLD VALUE S

- Consider 4 threads increment a counter 1000000 times each
- Low S → What is the consequence?
- High S → What is the consequence?



SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

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      }
```


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

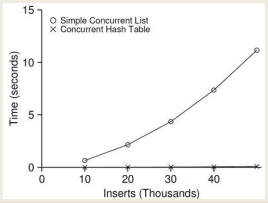
```
(Cont.)
21      void Queue_Enqueue(queue_t *q, int value) {
22              node_t *tmp = malloc(sizeof(node_t));
23              assert(tmp != NULL);
24
25              tmp->value = value;
26              tmp->next = NULL;
27
28              pthread_mutex_lock(&q->tailLock);
29              q->tail->next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