TCSS 422: OPERATING SYSTEMS

Locks

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

OBJECTIVES

- Lock Metrics
- Spin Locks
- Hardware support- atomic instructions for locks
- Yielding
- Queues and User Control

LOCKS

- Ensure code critical section(s) are executed atomically
  - Only one thread is allowed to execute a critical section at any given time
  - Ensures the code snippets are "mutually exclusive"
- Protect a global counter:

```
balance = balance + 1;
```

- A "critical section":

```
1    lock_t mutex; // some globally-allocated lock 'mutex'
2    …
3    lock(&mutex);
4    balance = balance + 1;
5    unlock(&mutex);
```


LOCKS - 2

- Lock variables are called "MUTEX"
  - Short for mutual exclusion (that's what they guarantee)
- Lock variable store the state of the lock
- States
  - **Locked** (acquired or held)
  - **Unlocked** (available or free)
- Only 1 thread can hold a lock

LOCKS - 3

- `pthread_mutex_lock(&lock)`
  - Try to acquire lock
  - If lock is free, calling thread will acquire the lock
  - Thread with lock enters critical section
    - Thread "owns" the lock
- No other thread can acquire the lock before the owner releases it.

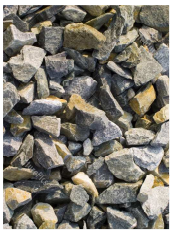
LOCKS - 4

- Program can have many mutex (lock) variables to "serialize" many critical sections
- Locks are also used to protect data structures
  - Prevent multiple threads from changing the same data simultaneously
  - Programmer can make sections of code "granular"
    - Fine grained – means just one grain of sand at a time through an hour glass
  - Similar to relational database transactions
    - DB transactions prevent multiple users from modifiying a table, row, field

FINE GRAINED?

- Is this code a good example of "*fine grained parallelism*"?

```
pthread_mutex_lock(&lock);
a = b++;
b = a * c;
*d = a + b +c;
FILE * fp = fopen ("file.txt", "r");
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
ListNode *node = mylist->head;
Int i=0
while (node) {
  node->title = str1;
  node->subheading = str2;
  node->desc = str3;
  node->end = *e;
  node = node->next;
  i++
}
e = e - i;
pthread_mutex_unlock(&lock);
```



GRANULAR PARALLELISM

```
pthread_mutex_lock(&lock_a);
pthread_mutex_lock(&lock_b);
a = b++;
pthread_mutex_unlock(&lock_b);
pthread_mutex_unlock(&lock_a);

pthread_mutex_lock(&lock_b);
b = a * c;
pthread_mutex_unlock(&lock_b);

pthread_mutex_lock(&lock_d);
*d = a + b +c;
pthread_mutex_unlock(&lock_d);

FILE * fp = fopen ("file.txt", "r");
pthread_mutex_lock(&lock_e);
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
pthread_mutex_unlock(&lock_e);

ListNode *node = mylist->head;
int i=0 . . .
```

EVALUATING LOCK IMPLEMENTATIONS

- Correctness
  - Does the lock work?
  - Are critical sections mutually exclusive? (atomic?)
- Fairness
  - Are threads competing for a lock have a fair chance of acquiring it?
- Overhead

BUILDING LOCKS

- Locks require hardware support
  - To minimize overhead, ensure fairness and correctness
  - Special "atomic" instructions to support lock implementation
  - Atomic exchange instruction
    - `XCHG`
  - Compare and exchange instruction
    - `CMPXCHG`
    - `CMPXCHG8B`
    - `CMPXCHG16B`

HISTORICAL IMPLEMENTATION

- To implement mutual exclusion
  - Disable interrupts upon entering critical sections

```
1  void lock() {
2      DisableInterrupts();
3  }
4  void unlock() {
5      EnableInterrupts();
6  }
```

- Any thread could disable system-wide interrupt
  - What if lock is never released?
- On a multiprocessor processor each CPU has its own interrupts
  - Do we disable interrupts for all cores simultaneously?
- While interrupts are disabled, they could be lost
  - If not queued…

SPIN LOCK IMPLEMENTATION

- Without atomic assembly instructions
- "Do-it-yourself" Locks
- Is this lock implementation: Correct? Fair? Performant?

```
1   typedef struct __lock_t { int flag; } lock_t;
2
3   void init(lock_t *mutex) {
4       // 0 → lock is available, 1 → held
5       mutex->flag = 0;
6   }
7
8   void lock(lock_t *mutex) {
9       while (mutex->flag == 1)  // TEST the flag
10          ;  // spin-wait (do nothing)
11      mutex->flag = 1;  // now SET it !
12  }
13
14  void unlock(lock_t *mutex) {
15      mutex->flag = 0;
16  }
```

DIY: CORRECT?

- Correctness requires luck…

Thread1	Thread2
call lock()	
while (flag == 1)	
interrupt: switch to Thread 2	
	call lock()
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

- Here both threads have "acquired" the lock simultaneously

DIY: PERFORMANT?

```
void lock(lock_t *mutex)
{
  while (mutex->flag == 1);   // while lock is unavailable, wait…
  mutex->flag = 1;
}
```

- What is wrong with while(<cond>); ?
- Spin-waiting wastes time actively waiting for another thread
- while (1); will "peg" a CPU core at 100%
  - Continuously loops, and evaluates mutex->flag value…
  - Generates heat…

DIY: TEST-AND-SET INSTRUCTION

- C implementation
  - Implements atomicity for a spin lock
  - Try this…

```
1   int TestAndSet(int *ptr, int new) {
2       int old = *ptr;  // fetch old value at ptr
3       *ptr = new;      // store 'new' into ptr
4       return old;      // return the old value
5   }
```

- lock() method will checks that TestAndSet doesn't return 1
- Comparison is in the caller

DIY: TEST-AND-SET - 2

- Requires a preemptive scheduler on single CPU core system
- Lock is never released without a context switch

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0 indicates that lock is available,
7       // 1 that it is held
8       lock->flag = 0;
9   }
10
11  void lock(lock_t *lock) {
12      while (TestAndSet(&lock->flag, 1) == 1)
13              ;          // spin-wait
14  }
15
16  void unlock(lock_t *lock) {
17      lock->flag = 0;
18  }
```


SPIN LOCK EVALUATION

- **Correctness:**
  - Spin locks guarantee: critical sections won't be executed simultaneously by (2) threads
- **Fairness:**
  - No fairness guarantee. Once a thread has a lock, nothing forces it to relinquish it…
- **Performance:**
  - Spin locks perform "busy waiting"
  - Spin locks are best for short periods of waiting
  - Performance is slow when multiple threads share a CPU
    - Especially for long periods

COMPARE AND SWAP

- Checks that the lock variable has the expected value FIRST, before changing its value
  - If so, make assignment
  - Return value at location
- Adds a comparison to TestAndSet
- Useful for wait-free synchronization
  - Supports implementation of shared data structures which can be updated atomically using the HW support CompareAndSwap instruction
  - Shared data structure updates become "wait-free"
  - Upcoming in Chapter 32

COMPARE AND SWAP

- Compare and Swap

```
1   int CompareAndSwap(int *ptr, int expected, int new) {
2       int actual = *ptr;
3       if (actual == expected)
4               *ptr = new;
5       return actual;
6   }
```

- Spin lock usage

```
1   void lock(lock_t *lock) {
2       while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3               ; // spin
4   }
```

- X86 provides "`cmpxchgl`" compare-and-exchange instruction
  - `cmpxchg8b`
  - `cmpxchg16b`

TWO MORE "LOCK BUILDING" CPU INSTRUCTIONS

- Instructions used together to support synchronization
- No support on x86 processors
  - Supported by RISC: Alpha, PowerPC, ARM
- Load-linked (LL)
  - Loads value into register
  - Same as typical load
  - Used as a mechanism to track competition
- Store-conditional (SC)
  - Performs "mutually exclusive" store
  - Allows only one thread to store value

LL/SC LOCK

```
1   int LoadLinked(int *ptr) {
2       return *ptr;
3   }
4
5   int StoreConditional(int *ptr, int value) {
6       if (no one has updated *ptr since the LoadLinked to this address) {
7               *ptr = value;
8               return 1; // success!
9       } else {
10              return 0; // failed to update
11      }
12  }
```

- LL instruction loads pointer value (ptr)
- SC only stores if the load link pointer has not changed
- Requires HW support
  - C code is psuedo code

LL/SC LOCK - 2

```
1   void lock(lock_t *lock) {
2       while (1) {
3               while (LoadLinked(&lock->flag) == 1)
4                       ; // spin until it's zero
5               if (StoreConditional(&lock->flag, 1) == 1)
6                       return; // if set-it-to-1 was a success: all done
7                               otherwise: try it all over again
8       }
9   }
10
11  void unlock(lock_t *lock) {
12      lock->flag = 0;
13  }
```

- Two instruction lock

FETCH-AND-ADD

- HW CPU Instruction
- Increment counter atomically 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

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 !

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  }
```

- Change thread state:
  - 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

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 for adding/removing theads to/from queue(s)
- Solaris
  - park(): puts thread to sleep
  - unpark(threadID): wakes specified thread
- Linux: 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 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 - 2

- 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
  - 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 - 3

```
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 - 4

```
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);

}
```

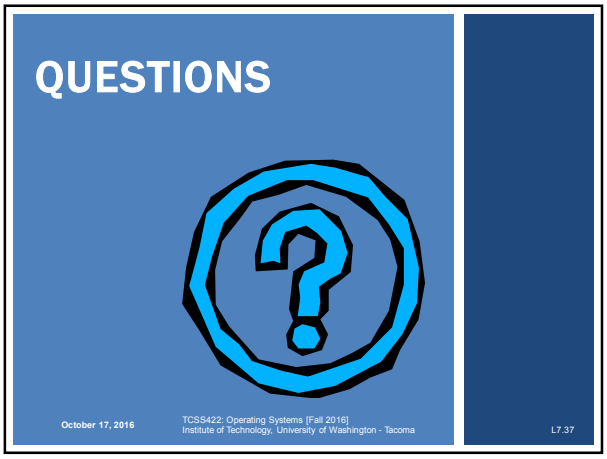
- Interesting note: Futex bug in Redhat Linux
- https://www.infoq.com/news/2015/05/redhat-futex

TWO PHASE LOCKS

- Hybrid between spin-locks and yielding
- Useful if lock is about to be released

- First phase
  - Spin lock for some time waiting for the lock to be released
  - If lock is not acquired after time expires enter phase two.

- Second phase
  - Thread sleeps (yields)
  - Is awoken when the lock becomes free


QUESTIONS
October 17, 2016
TCSS422: Operating Systems [Fall 2016]
Institute of Technology, University of Washington - Tacoma
L7.37