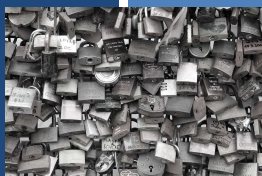


TCCS 422: OPERATING SYSTEMS

Locks

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



FEEDBACK FROM 4/20/2017

- What does it mean for a thread to be joinable?

```
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

- pthread_attr_t state: PTHREAD_CREATE_JOINABLE, PTHREAD_CREATE_DETACHED

- Joinable threads: another thread can synchronize on the thread termination and recover its termination code using pthread_join(3).

When a joinable thread terminates, some resources are kept allocated and released only when another thread performs pthread_join(3) on that thread.

April 20, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.2

FEEDBACK - 2

- pthread_attr_t state: PTHREAD_CREATE_DETACHED

- In the detached state, the thread's resources are released immediately when it terminates. pthread_join(3) cannot be used to synchronize detached threads on termination.

- A thread created in the joinable state can later be put in the detached thread using pthread_detach(3).

April 20, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.3

FEEDBACK - 3

- What is an atomic instruction?

- Processors have instructions that can be used to implement lock-free and wait-free algorithms (and locks)

- Atomic instructions **CAN NOT** be preempted by a **context-switch**

- They represent the lowest level instructions of the machine...

Examples:

- Atomic read-write instruction
- Atomic swap, called (XCHG)
- Test-and-set
- Fetch-and-add
- Compare-and-swap (CAS)
- Compare-and-exchange (CMPXCHG) - x86

Theme: One assembly instruction: loads value from memory into CPU register, makes a change, stores back into memory

April 20, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.4

FEEDBACK - 4

- What are some examples of best practices for multithreading with locks?

- We cover some in Chapter 29 & 32...

- How do you know if a pointer or variable is on the stack, or the heap?

- All variables are on the stack, unless we explicitly call malloc.
- We typically create a pointer on the stack to point to a variable (or struct) on the heap.

- What is Void ** ?

- Is a pointer to a void *
- So a void * is a pointer to any memory block on the heap, and a
- Void ** , is a pointer, to a pointer to any memory block on the heap
- By any memory block, we mean, it can be any type of struct or datatype

April 20, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.5

FEEDBACK - 5

- What are attributes for pthread_mutex_t lock?

- pthread_mutex_t is the lock data structure (e.g. it's struct)

- From :

- /usr/include/bits/pthreadtypes.h
- Ubuntu 16.04

April 20, 2017

TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.6

```
typedef union
{
    struct __pthread_mutex_s
    {
        int __lock;
        unsigned int __count;
        int __owner;
#ifdef __x86_64__
        unsigned int __nusers;
#endif
        /* KIO must stay at this position in the structure to maintain
        binary compatibility. */
        int __kind;
#ifdef __x86_64__
        short __spins;
        short __elision;
        __pthread_list_t __list;
        #define __PTHREAD_MUTEX_HAVE_PREV 1
        /* Mutex spins initializer used by PTHREAD_MUTEX_INITIALIZER. */
        #define __PTHREAD_SPINS 0, 0
    #else
        unsigned int __nusers;
        __extension__ union
        {
            struct
            {
                short __spins;
                short __elision;
            };
            #define __PTHREAD_MUTEX_DATA __spins
            #define __PTHREAD_SPINS { 0, 0 }
        };
        __pthread_list_t __list;
    #endif
    } __data;
    char __size[___SIZEOF_PTHREAD_MUTEX_T];
    long int __align;
} pthread_mutex_t;
```

April 20, 2017 TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma L7.7


FEEDBACK - 6

- How does the "Test and Set" lock work?
- How does it spin?
- "Test and Set" is an improvement on the basic spin lock
- "Compare and Swap" is an improvement on "Test and Set"
- Each improvement adds additional checking to verify that the lock **is not** held by another thread first.
- For multi-core CPUs, context switching with non-atomic locks, interrupts the spin lock, causing a race condition where two threads **think** they've acquired the lock at the same time

April 20, 2017 TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma L7.8

SPIN LOCK IMPLEMENTATION

- Operate without atomic-as a unit 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 }
```

April 18, 2017 TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma L7.9

FEEDBACK - 7

- 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

April 20, 2017 TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma L7.10

FEEDBACK - 8

- Why use fine grained parallelism? Isn't that the same as just blocking a whole block of code? Because only one thread can access that block of code, because they need to acquire the lock?
- The idea is that we might have many threads (let's say 10), and if our code has 10 locks protecting 10 variables individually...

then it's possible that **some** of the threads will be operating in different parts of the shared code and **making progress** because each thread **DOES NOT HAVE TO ACQUIRE THE GLOBAL LOCK...**

April 20, 2017 TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma L7.11

FINE GRAINED 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 %d", str1, str2, str3, &e);
pthread_mutex_unlock(&lock_e);

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



April 18, 2017 TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma L7.12

FEEDBACK - 9

- Could the assignments be on the calendar... YES
- Grading for assignment 0 – should be posted
 - Some students did not submit an answers file which interprets the output. You may submit as a text file to canvas, and send and email to the grader and CC Wes to request regrading
- In the future: please read and follow assignment specs
- What can we expect to be on the midterm?
 - We will have an in-class mock exam.
 - Roughly coverage is from Ch 1-9, 26-32, with a focus on italicized chapters... more discussion to follow

April 20, 2017

TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.13

OBJECTIVES

- Finishing up Chapter 28
 - Spin Locks - demonstration
 - Spin Locks - review
 - Yielding
 - Queues and User Control
- Chapter 29 – Lock Based Data Structure

April 20, 2017

TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.14

SPIN LOCKING DEMO

- Install httpd on CentOS7:
- `wget http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-9.noarch.rpm`
- `sudo rpm -ihv epel-release-7-9.noarch.rpm`
- `sudo yum install httpd`
- Spin locking demo
 - CentOS 7 VM
 - With 1 CPU core
 - With 2 CPU cores

April 20, 2017

TCS5422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L7.15

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

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.16

## 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 }

```

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.17

## 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 }

```

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.18

## 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

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.19

## 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 !

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.20

## 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
  - Thread pools**: track and reuse your own 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()

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.21

## 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) // acquire guard lock by spinning
11 ; // Obtain guard lock
12 if (m->flag == 0) { // lock is acquired
13 m->flag = 1; // try to obtain actual lock
14 m->guard = 0;
15 }
16 else {
17 queue_add(m->q, gettid()); // lock unavailable; add thread to queue
18 park(); // potential wakeup/waiting race
19 }
20 }
21 ...
```

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.22

## THREAD QUEUES - 3

- Unlock

```
22 void unlock(lock_t *m) {
23 while (TestAndSet(&m->guard, 1) == 1) // Obtain guard lock (spin)
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)); // wake up thread from queue
29 m->guard = 0; // release guard lock
30 }
```

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

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.23

## 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

April 20, 2017

TCS5422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.24

## FUTEX



- Fast **U**space **MuTEx**
- 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

|                |                                                                                                       |       |
|----------------|-------------------------------------------------------------------------------------------------------|-------|
| April 20, 2017 | TCS422: Operating Systems [Spring 2017]<br>Institute of Technology, University of Washington - Tacoma | L7.25 |
|----------------|-------------------------------------------------------------------------------------------------------|-------|

## 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

|                |                                                                                                       |       |
|----------------|-------------------------------------------------------------------------------------------------------|-------|
| April 20, 2017 | TCS422: Operating Systems [Spring 2017]<br>Institute of Technology, University of Washington - Tacoma | L7.26 |
|----------------|-------------------------------------------------------------------------------------------------------|-------|

## 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;
 if (v >= 0)
 continue;
 // wait to be woken up when lock is available
 // this is not a spin lock... (signal)
 futex_wait (mutex, v);
 }
}

```

|                |                                                                                                       |       |
|----------------|-------------------------------------------------------------------------------------------------------|-------|
| April 20, 2017 | TCS422: Operating Systems [Spring 2017]<br>Institute of Technology, University of Washington - Tacoma | L7.27 |
|----------------|-------------------------------------------------------------------------------------------------------|-------|

## 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>


|                |                                                                                                       |       |
|----------------|-------------------------------------------------------------------------------------------------------|-------|
| April 20, 2017 | TCS422: Operating Systems [Spring 2017]<br>Institute of Technology, University of Washington - Tacoma | L7.28 |
|----------------|-------------------------------------------------------------------------------------------------------|-------|

## 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

|                |                                                                                                       |       |
|----------------|-------------------------------------------------------------------------------------------------------|-------|
| April 20, 2017 | TCS422: Operating Systems [Spring 2017]<br>Institute of Technology, University of Washington - Tacoma | L7.29 |
|----------------|-------------------------------------------------------------------------------------------------------|-------|

## LOCK BASED DATA STRUCTURES



|                |                                                                                                       |       |
|----------------|-------------------------------------------------------------------------------------------------------|-------|
| April 20, 2017 | TCS422: Operating Systems [Spring 2017]<br>Institute of Technology, University of Washington - Tacoma | L7.30 |
|----------------|-------------------------------------------------------------------------------------------------------|-------|

## OBJECTIVES

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

April 20, 2017

TCCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.31

## LOCK-BASED CONCURRENT DATA STRUCTURES

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

April 20, 2017

TCCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.32

## 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 }
```

April 20, 2017

TCCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.33

## 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

April 20, 2017

TCCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.34

## 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 }
```

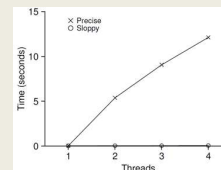
April 20, 2017

TCCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.35

## 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.**

April 20, 2017

TCCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.36

PERFECT SCALING

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

April 20, 2017

TCSS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.37

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?

April 20, 2017

TCSS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.38

QUESTIONS



April 20, 2017

TCSS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma

L7.55