

TCCS 422: OPERATING SYSTEMS

Concurrency: An Introduction



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

FEEDBACK FROM 4/13

- **Chapter 9: Scheduling**
 - How does stride scheduling handle a new job?
- **Chapter 26:**
 - What is the difference between a process and a thread?
- **What is volatile?**
 - **In Java**, all reads and writes to a variable are guaranteed to be atomic unless the variable is a long or double
 - JVMs determine exact implementation
 - When volatile is used in Java, reads and writes to long and double become atomic

April 18, 2017

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

L6.2

FEEDBACK - 2

- **What is volatile?**
- **In C**: Qualifier added to variable when declared. Provides a hint to the compiler that the variable may change at any time--without action by nearby code.
- Variable should be declared volatile when its value could change unexpectedly. Three scenarios:
 - 1. Memory-mapped peripheral registers
 - 2. Global variables modified by an interrupt service routine
 - 3. Global variables accessed by multiple threads
- <https://barrgroup.com/Embedded-Systems/How-To/C-Volatile-KeyWord>

April 18, 2017

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

L6.3

FEEDBACK - 3

- **How does pthread_join() work?**
 - Waits for specified thread to terminate
 - Thread must be joinable
 - Copies exit/return data from child worker to pointed addr
 - See man page. . .
- **How does it wait?**
 - Blocks main thread from proceeding until child finishes
 - Relinquishes the CPU until child exits
 - Not a busy wait
- **What does it wait for?**
 - pthread_join() waits for child thread to finish

April 18, 2017

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

L6.4

OBJECTIVES

- Chapter 26
 - Introduction to threads
 - Race condition
 - Critical section
- Chapter 27 - Thread API
- Chapter 28 - Locks

April 18, 2017

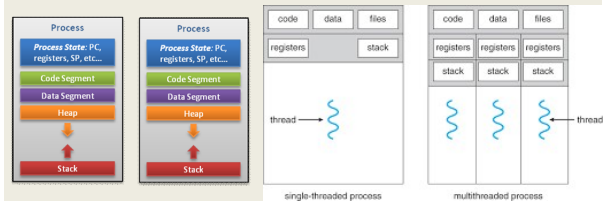
TCCS422: Operating Systems [Spring 2017]
Institute of Technology, University of Washington - Tacoma

L6.5

PROCESSES VS. THREADS



- **What's the difference between forks and threads?**
 - **Forks**: duplicate a process
 - Think of **CLONING** - There will be two identical processes at the end
 - **Threads**: no duplicate of code/heap, lightweight execution threads



April 18, 2017

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

L6.6

THREADS - 2

- Enables a single process (program) to have multiple “workers”
- Supports independent path(s) of execution within a program
- Each thread has its own Thread Control Block (TCB)
 - PC, registers, SP, and stack
- Code segment, memory, and heap are shared

April 18, 2017

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

L6.7

RACE CONDITION

- What is happening with our counter?
 - When counter=50, consider code: counter = counter + 1
 - If synchronized, counter will = 52

	OS	Thread1	Thread2	(after instruction)
				PC %eax counter
[before critical section		100 0 50
		mov 0x8049a1c, %eax		105 50 50
		add \$0x1, %eax		108 51 50
	Interrupt	save T1's state		
[restore T2's state		100 0 50
			mov 0x8049a1c, %eax	105 50 50
			add \$0x1, %eax	108 51 50
			mov %eax, 0x8049a1c	113 51 51
	Interrupt	save T2's state		
[restore T1's state		108 51 50
			mov %eax, 0x8049a1c	113 51 51


April 18, 2017

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

L6.8

CRITICAL SECTION

- Code that accesses a shared variable must not be **concurrently** executed by more than one thread
- Multiple **active** threads inside a critical section produces a **race condition**.
- Atomic execution** (all code executed as a *unit*) must be ensured in **critical** sections
 - These sections must be **mutually exclusive**



April 18, 2017

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

L6.9

LOCKS

- To demonstrate how critical section(s) can be executed “atomically-as a *unit*” Chapter 27 & beyond introduce locks

```
1 lock_t mutex;  
2 . . .  
3 lock(&mutex);  
4 balance = balance + 1;  
5 unlock(&mutex);
```

Critical section


- Counter example revisited

April 18, 2017

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

L6.10

LINUX
THREAD API



April 18, 2017

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

L6.11

THREAD CREATION

- pthread_create

```
#include <pthread.h>  
  
int  
pthread_create(  
    pthread_t* thread,  
    const pthread_attr_t* attr,  
    void* (*start_routine)(void*),  
    void* arg);
```

- thread: thread struct
- attr: stack size, scheduling priority... (optional)
- start_routine: function pointer to thread routine
- arg: argument to pass to thread routine (optional)

April 18, 2017

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

L6.12

PTHREAD_CREATE - PASS ANY DATA

```
#include <pthread.h>

typedef struct _myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    --
}
```

April 18, 2017

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

L6.13

PASSING A SINGLE VALUE

- Here we "cast" the pointer to pass/return a primitive data type

```
1 void *mythread(void *arg) {
2     int m = (int) arg;
3     printf("%d\n", m);
4     return (void *) (arg + 1);
5 }
6
7 int main(int argc, char *argv[]) {
8     pthread_t p;
9     int rc, m;
10    pthread_create(&p, NULL, mythread, (void *) 100);
11    pthread_join(p, (void **) &m);
12    printf("returned %d\n", m);
13    return 0;
14 }
```

April 18, 2017

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

L6.14

PASSING A SINGLE VALUE

Using this approach on your CentOS 7 VM
How large (in bytes) can the primitive data type be?

How large (in bytes) can the primitive data type be on a 32-bit operating system?

```
9 int rc, m;
10 pthread_create(&p, NULL, mythread, (void *) 100);
11 pthread_join(p, (void **) &m);
12 printf("returned %d\n", m);
13 return 0;
14 }
```

April 18, 2017

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

L6.15

WAITING FOR THREADS TO FINISH

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- thread: which thread?
- value_ptr: pointer to return value
type is dynamic / agnostic
- Returned values *must* be on the heap
- Thread stacks destroyed upon thread termination (join)
- Pointers to thread stack memory addresses are invalid
 - May appear as gibberish or lead to crash (seg fault)
- Not all threads join - *What would be Examples ??*

April 18, 2017

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

L6.16

What will this code do?

```
struct myarg {
    int a;
    int b;
};

void *worker(void *arg)
{
    struct myarg *input = (struct myarg *) arg;
    printf("a=%d b=%d\n", input->a, input->b);
    struct myarg output;
    output.a = 1;
    output.b = 2;
    return (void *) &output;
}

int main (int argc, char * argv[])
{
    pthread_t p1;
    struct myarg args;
    struct myarg *ret_args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p1, NULL, worker, &args);
    pthread_join(p1, (void **) &ret_args);
    printf("returned %d %d\n", ret_args->a, ret_args->b);
    return 0;
}
```

Data on thread stack

**\$./pthread_struct
a=10 b=20
Segmentation fault (core dumped)**

How can this code be fixed?

April 18, 2017

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

L6.17

How about this code?

```
struct myarg {
    int a;
    int b;
};

void *worker(void *arg)
{
    struct myarg *input = (struct myarg *) arg;
    printf("a=%d b=%d\n", input->a, input->b);
    input->a = 1;
    input->b = 2;
    return (void *) &input;
}

int main (int argc, char * argv[])
{
    pthread_t p1;
    struct myarg args;
    struct myarg *ret_args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p1, NULL, worker, &args);
    pthread_join(p1, (void **) &ret_args);
    printf("returned %d %d\n", ret_args->a, ret_args->b);
    return 0;
}
```

**\$./pthread_struct
a=10 b=20
returned 1 2**

April 18, 2017

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

L6.18

ADDING CASTS

- Casting
- Suppresses compiler warnings when passing "typed" data where (void) or (void *) is called for
- Example: uncasted capture in pthread_join
pthread_int.c: In function 'main':
pthread_int.c:34:20: warning: passing argument 2 of 'pthread_join' from incompatible pointer type [-Wincompatible-pointer-types]
pthread_join(p1, &p1val);
- Example: uncasted return
In file included from pthread_int.c:3:0:
/usr/include/pthread.h:250:12: note: expected 'void **' but argument is of type 'int **'
extern int pthread_join (pthread_t __th, void **__thread_return);

April 18, 2017

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

L6.19

ADDING CASTS - 2

- pthread_join
int * p1val;
int * p2val;
pthread_join(p1, (void *)&p1val);
pthread_join(p2, (void *)&p2val);
- return from thread function
int * counterval = malloc(sizeof(int));
*counterval = counter;
return (void *) counterval;

April 18, 2017

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

L6.20

LOCKS

- pthread_mutex_t data type
 - /usr/include/bits/pthread_types.h
// Global Address Space
static volatile int counter = 0;
pthread_mutex_t lock;
- ```
void *worker(void *arg)
{
 int i;
 for (i=0; i<100000000; i++) {
 int rc = pthread_mutex_lock(&lock);
 assert(rc==0);
 counter = counter + 1;
 pthread_mutex_unlock(&lock);
 }
 return NULL;
}
```

April 18, 2017

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

L6.21

## LOCKS - 2

- Ensure critical sections are executed atomically-as a unit
  - Provides implementation of "Mutual Exclusion"
- API  
int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);  
int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);
- Example w/o initialization & error checking  
pthread\_mutex\_t lock;  
pthread\_mutex\_lock(&lock);  
x = x + 1; // or whatever your critical section is  
pthread\_mutex\_unlock(&lock);
- Blocks forever until lock can be obtained
- Enters critical section once lock is obtained
- Releases lock

April 18, 2017

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

L6.22

## LOCK INITIALIZATION

- Assigning the constant  
pthread\_mutex\_t lock = PTHREAD\_MUTEX\_INITIALIZER;
- API call:  
int rc = pthread\_mutex\_init(&lock, NULL);  
assert(rc == 0); // always check success!
- Initializes mutex with attributes specified by 2<sup>nd</sup> argument
- If NULL, then default attributes are used
- Upon initialization, the mutex is initialized and unlocked

April 18, 2017

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

L6.23

## LOCKS - 3

- Error checking wrapper  
// Use this to keep your code clean but check for failures  
// Only use if exiting program is OK upon failure  
void Pthread\_mutex\_lock(pthread\_mutex\_t \*mutex) {  
 int rc = pthread\_mutex\_lock(mutex);  
 assert(rc == 0);  
}
- What if lock can't be obtained?  
int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);  
int pthread\_mutex\_timelock(pthread\_mutex\_t \*mutex,  
 struct timespec \*abs\_timeout);
- trylock - returns immediately (fails) if lock is unavailable
- timelock - tries to obtain a lock for a specified duration

April 18, 2017

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

L6.24

## CONDITIONS AND SIGNALS

- Condition variables support "signaling" between threads

```
int pthread_cond_wait(pthread_cond_t *cond,
 pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```



- pthread\_cond\_t datatype**
- pthread\_cond\_wait()**
  - Waits (sleeps)
  - Listens for a "signal"
  - Releases the lock until signaled

April 18, 2017

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

L6.25

## CONDITIONS AND SIGNALS - 2

```
int pthread_cond_signal(pthread_cond_t * cond);
int pthread_cond_broadcast(pthread_cond_t * cond);
```

- pthread\_cond\_signal()**
  - Called to send a "signal" to all listeners → to wake them up
  - The goal is to unblock (at least one) to respond to the signal
- pthread\_cond\_broadcast()**
  - Unblocks all threads currently blocked on the specified condition
  - Used when all threads should respond to the signal
- Which thread is unblocked first?**
  - Determined by OS scheduler (based on priority)
  - Thread(s) gain the lock individually (based on priority) as if they called `pthread_mutex_lock()`

April 18, 2017

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

L6.26

## CONDITIONS AND SIGNALS - 3

- Wait example

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
 pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- wait puts thread to sleep, releases lock
- when awoken, lock reacquired (and released by this code)
- Another thread signals the thread

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Code performs required work before other thread(s) can continue

April 18, 2017

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

L6.27

## CONDITION AND SIGNALS - 4

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
 pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- Why do we wait inside a while loop?
- The while ensures upon awakening the condition is rechecked
  - A signal may have been raised, but the condition to proceed has not been satisfied.
  - Without checking the condition the thread may proceed to execute when it should not.

April 18, 2017

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

L6.28

## PTHREADS LIBRARY

- Compilation**
  - `gcc -pthread pthread.c -o pthread`
  - Requires explicitly linking the library with compiler flag
- List of pthread manpages**
  - `man -k pthread`

April 18, 2017

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

L6.29

## SAMPLE MAKEFILE

```
CC=gcc
CFLAGS=-pthread -I. -Wall

binaries=pthread pthread_int pthread_lock_cond pthread_struct
all: $(binaries)

pthread_mult: pthread.c pthread_int.c
$(CC) $(CFLAGS) $^ -o $@

clean:
$(RM) -f $(binaries) *.o
```


- Example builds multiple single file programs
  - All target
- pthread\_mult**
  - Example if multiple source files should produce a single executable
- clean target**

April 18, 2017

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


L6.30

# LOCKS



April 18, 2017 TCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma L6.31

# LOCKS



- Ensure critical section(s) are executed atomically-as a *unit*
  - 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);
```

April 18, 2017 TCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma L7.32

## LOCKS - 2

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

April 18, 2017 TCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma L7.33

## 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.

April 18, 2017 TCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma L7.34

## 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 modifying a table, row, field

April 18, 2017 TCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma L7.35

## 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);
```



April 18, 2017 TCS422: Operating Systems [Spring 2017]  
Institute of Technology, University of Washington - Tacoma L7.36

## 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 . . .
```



April 18, 2017

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

L7.37

## EVALUATING LOCK IMPLEMENTATIONS

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



April 18, 2017

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

L7.38

## BUILDING LOCKS

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

April 18, 2017

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

L7.39

## 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...

April 18, 2017

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

L7.40

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

L7.41

DIY: CORRECT?

- **Correctness requires luck... (e.g. DIY lock is incorrect)**

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

April 18, 2017

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

L7.42

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...

April 18, 2017

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

L7.43

TEST-AND-SET INSTRUCTION

- C implementation: not atomic
 - Adds a simple check to basic spin lock
 - One a single core CPU system with preemptive scheduler:
 - 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 checks that TestAndSet doesn't return 1
- Comparison is in the caller
- Single core systems are becoming scarce
- Try on a one-core VM

April 18, 2017

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

L7.44

DIY: TEST-AND-SET - 2

- Requires a preemptive scheduler on single CPU core system
- Lock is never released without a context switch
- 1-core VM: occasionally will deadlock, doesn't miscount

```
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 }
```

April 18, 2017

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

L7.45

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

April 18, 2017

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

L7.46

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 (as a unit) using the HW support CompareAndSwap instruction
 - Shared data structure updates become "wait-free"
 - Upcoming in Chapter 32

April 18, 2017

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

L7.47

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 "cmpxchg1" compare-and-exchange instruction
 - cmpxchg8b
 - cmpxchg16b

April 18, 2017

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

L7.48

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

**1-core VM:
Count is correct, no deadlock**

```
1 int spin_lock(int *lock) {
2   while (*lock == 1)
3     ; // spin
4 }
```

X86 provides "cmpxchg1" compare-and-exchange instruction

- `cmpxchg8b`
- `cmpxchg16b`

April 18, 2017

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

L7.49

TWO MORE "LOCK BUILDING" CPU INSTRUCTIONS

Cooperative instructions used together to support synchronization on RISC systems

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

April 18, 2017

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

L7.50

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

April 18, 2017

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

L7.51

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

April 18, 2017

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

L7.52

QUESTIONS



April 18, 2017

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

L6.53

DO-IT-YOURSELF LOCK

"initialized" -- global variable shared by multiple threads

Wait (client thread):

```
while(initialized == 0)
; // spin
```

Signal (parent thread): when ready...

```
initialized = 1;
```

How is this "wait" different that pthread_cond_wait() ?

- Wastes CPU cycles → effectively pegs a core at 100%
- Potential synchronization errors with changing the value of "initialized"
- Thread API is provided to advance the DO-IT-YOURSELF approach

April 18, 2017

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

L6.54