TCSS 422: OPERATING SYSTEMS

Locks, Lock Based Data Structures

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

January 31, 2018

TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Tutorial 1 Questions
- Homework 1 Questions
- Feedback from 1/29
- Ch. 28
  - Locks
- Ch. 29
  - Lock Based Data Structures

SELECTED FEEDBACK FROM 1/29

- **What happens when a thread reaches a lock that another thread is holding?**
- **Does it wait on that line of code? (lock line)**
- **Does it continually poll for the lock availability?**
  - pthread_block.c example
  - Check for PID: `ps u`
  - Trace PID: `top -d .1 -H -p <pid>`
- The thread will block (stop executing) until the lock is available

FEEDBACK - 2

- **Please give a real example of when to use:** `pthread_mutex_trylock()` and `pthread_mutex_timelock()`
- Both functions return integer 0 if lock is successfully acquired
- Both return integer error code (non-zero) otherwise
- Consider if 3 threads append text to the same file
- Only one thread is allowed to open the file for writing ***at the same time***
- Locks can be used to synchronize access to append to the file
- If a thread can not acquire the file lock, it accumulates changes in a buffer, but continues to perform useful work

FEEDBACK - 3

- **What are good habits to do when using locks to avoid bugs?**
- This is upcoming in chapter 32
- **What are good habits for fine-grained locking?**
- Want ***more*** opportunities for parallelism in code
- Method-level locking (coarse-grained) **does not** provide them
- Optimal number of locks is between 1 and n locks, where n is the number of variables being changed
- Finding optimal solution requires identifying how often variables will be modified, and in what combinations by multiple threads
- This is non-trivial → thread behavior @ runtime can be random

FEEDBACK - 4

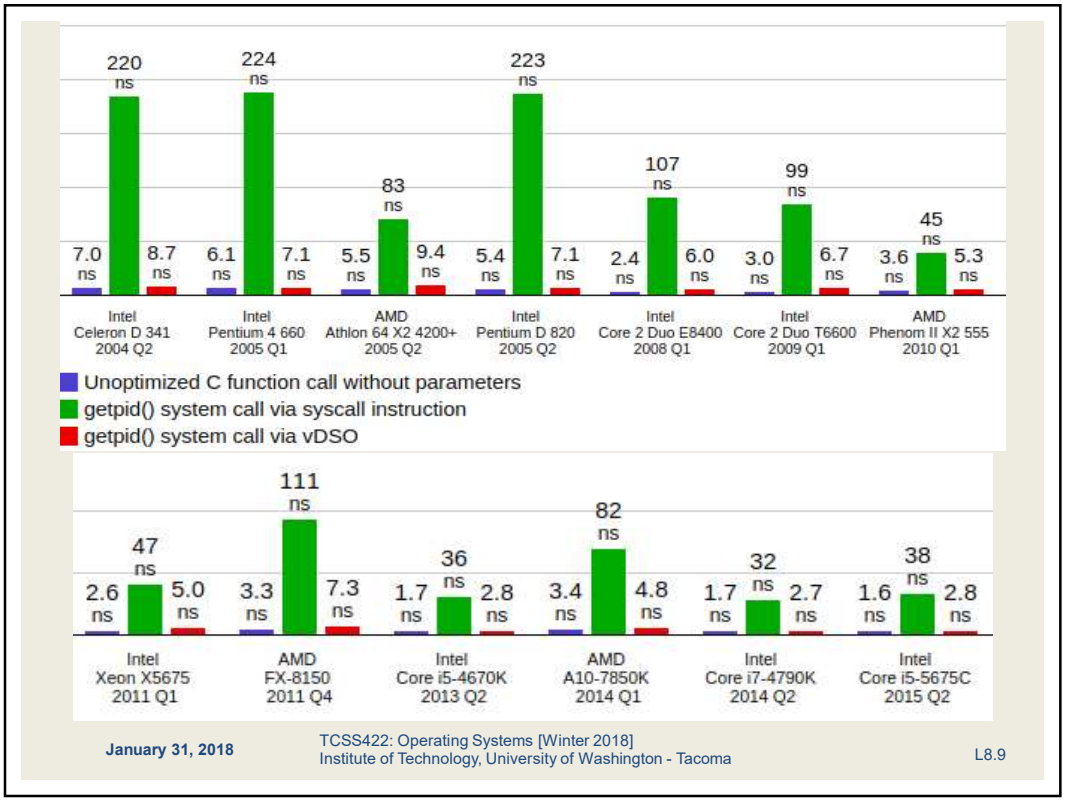
- **How much overhead does lock checking, waiting, and signaling have?**
- All of these are kernel calls
- Making just a few system calls shouldn't introduce too much overhead
- The blog, "Measurements of system call performance and overhead" provides insight:
- http://arkanis.de/weblog/2017-01-05-measurements-of-system-call-performance-and-overhead

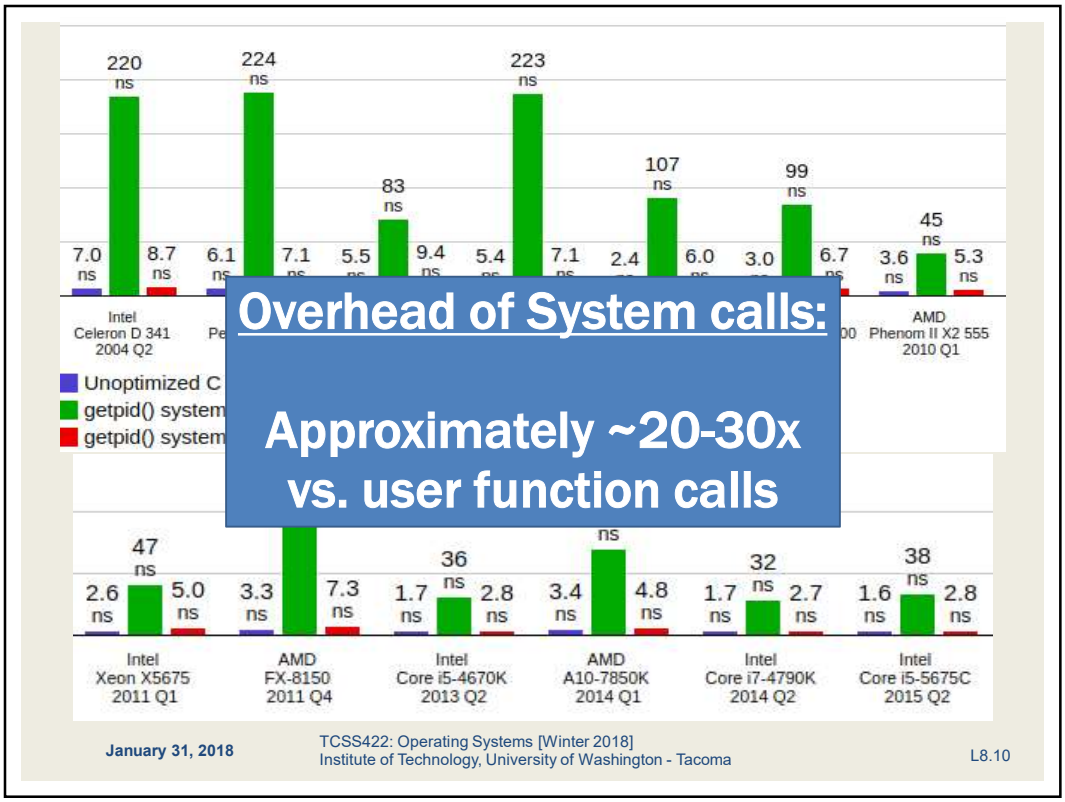
OVERHEAD OF A LINUX C SYSTEM CALL

- What is the performance overhead difference between a user function call, and a kernel system call?
- Ordinary function calls require passing data to/from the stack
- x86 64-bit systems perform kernel calls using **syscall** instruction
- Arguments placed into registers and syscall instruction transitions from ***user mode*** to ***kernel mode*** to call kernel function

OVERHEAD - 2

- Additional overhead is from CPU switching to different (protected) address space, CPU memory caches (TLB) may require updating
- Linux kernel provides optimized kernel calls via the vDSO: **V**irtually **d**ynamically linked **s**hared **o**bject (see https://en.wikipedia.org/wiki/VDSO)
- vDSO provides optimized "virtual" system calls where a selected set of ***common*** kernel space routines are exported to user space applications for faster execution
- Linux (glibc) uses virtual system calls automatically when available


220 ns
224 ns
223 ns
107 ns
99 ns
83 ns
45 ns
7.0 ns
8.7 ns
6.1 ns
7.1 ns
5.5 ns
9.4 ns
5.4 ns
7.1 ns
2.4 ns
6.0 ns
3.0 ns
6.7 ns
3.6 ns
5.3 ns
Intel Celeron D 341 2004 Q2
Intel Pentium 4 660 2005 Q1
AMD Athlon 64 X2 4200+ 2005 Q2
Intel Pentium D 820 2005 Q2
Intel Core 2 Duo E8400 2008 Q1
Intel Core 2 Duo T6600 2009 Q1
AMD Phenom II X2 555 2010 Q1
Unoptimized C function call without parameters
getpid() system call via syscall instruction
getpid() system call via vDSO
111 ns
82 ns
47 ns
36 ns
32 ns
38 ns
2.6 ns
5.0 ns
3.3 ns
7.3 ns
1.7 ns
2.8 ns
3.4 ns
4.8 ns
1.7 ns
2.7 ns
1.6 ns
2.8 ns
Intel Xeon X5675 2011 Q1
AMD FX-8150 2011 Q4
Intel Core i5-4670K 2013 Q2
AMD A10-7850K 2014 Q1
Intel Core i7-4790K 2014 Q2
Intel Core i5-5675C 2015 Q2
January 31, 2018
TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma
L8.9


Overhead of System calls:
Approximately ~20-30x
vs. user function calls
January 31, 2018
TCSS422: Operating Systems [Winter 2018]
Institute of Technology, University of Washington - Tacoma
L8.10

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  }
```


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  }
```

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

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 loc

```
3               ; // spin
4   }
```

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

1-core VM:
Count is correct, no deadlock

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

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-*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

- Fast Userspace 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

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

CHAPTER 29 – 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

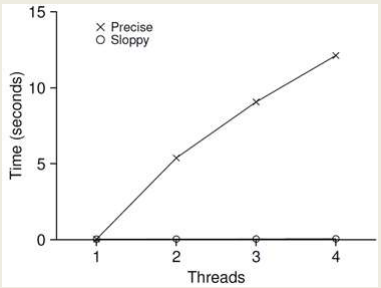
CONCURRENT COUNTER - 2

- 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?

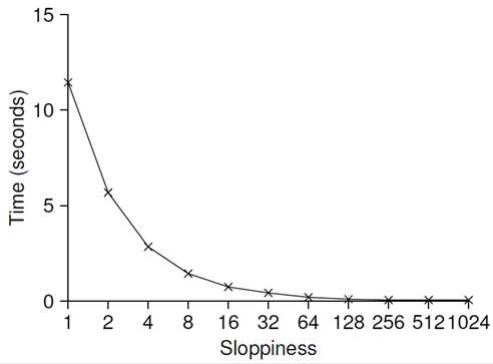
SLOPPY COUNTER - 2

- 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 …

QUESTIONS