TCSS 422: OPERATING SYSTEMS

Lock-Based Data Structures, Midterm Review

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma

April 30, 2020 TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma

OBJECTIVES – 4/30

- Questions from 4/28
- C Tutorial (Apr 30 11:59p AOE)
- Assignment 1 (May 7 11:59p AOE)
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- Midterm Review

April 30, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L10.2

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (39 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 7.30 (↑ *from* 7.21)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.92 (↑ from 5.53)

FEEDBACK FROM 4/28

- C tutorial (v.11)
- Question 6 – rephrased as:
- What is ALWAYS the ASCII integer value for the last character?

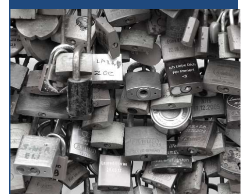
OBJECTIVES – 4/30

- Questions from 4/28
- C Tutorial (Apr 30 11:59p AOE)
- Assignment 1 (May 7 11:59p AOE)
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- Midterm Review

OBJECTIVES – 4/30

- Questions from 4/28
- C Tutorial (Apr 30 11:59p AOE)
- Assignment 1 (May 7 11:59p AOE)
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- Midterm Review

CHAPTER 29 – LOCK BASED DATA STRUCTURES



April 30, 2020
TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L10.7

OBJECTIVES – 4/30

- Questions from 4/28
- C Tutorial (Apr 30 11:59p AOE)
- Assignment 1 (May 7 11:59p AOE)
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- Midterm Review

April 30, 2020
TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma
L10.8

SLOPPY COUNTER

- Provides single logical shared counter
  - Implemented with 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
- Local counters for each CPU Core:
  - Distribute work evenly, each core has independent local counter

SLOPPY COUNTER – MAIN POINTS

- Main idea:
  RELAX synchronization requirement for counting
  - Instead of synchronizing global count variable each time:
    `counter=counter+1`
  - Synchronization occurs only every so often:
    e.g. *every 1000 counts*
- Relaxing the synchronization requirement ***drastically*** reduces locking API overhead by trading-off split-second accuracy of the counter
- Sloppy counter: trade-off accuracy for speed
  - It's sloppy because it's not so accurate (until the end)

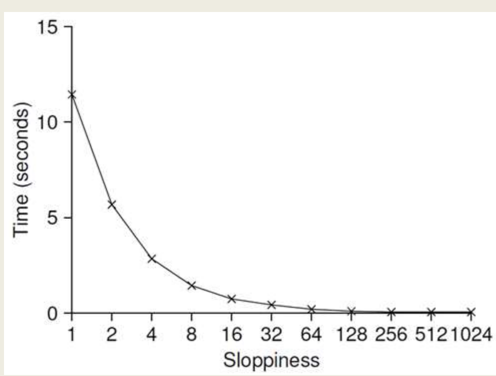
SLOPPY COUNTER - 2

- 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?

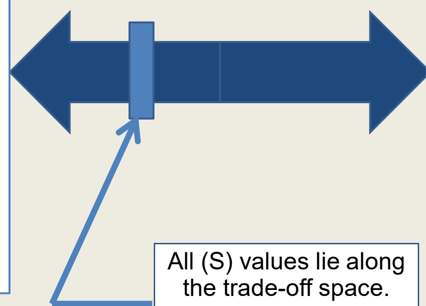


SLOPPY THRESHOLD (S) TRADE-OFF SPACE

LOW (S) VALUE

1. **Behavior 1**
2. **Behavior 2**
3. **Behavior 3**
4. **Behavior 4**

Classify: (high vs. low)
Number of global updates
Performance, Overhead, Accuracy



All (S) values lie along the trade-off space.

The (S) value changes the sloppy counter's behavior

HIGH (S) VALUE

1. **Behavior 1**
2. **Behavior 2**
3. **Behavior 3**
4. **Behavior 4**

SLOPPY COUNTER - EXAMPLE

- Example implementation
- Chapter 29
  - `sloppybasic.c`
- Also a version with CPU affinity:
  - `sloppy.c`
  - Local counters pinned to specific CPUs
  - Uses sched_setaffinity API call for CPU pinning

CONCURRENT LINKED LIST - 1

- Simplification - only basic list operations shown
- Structs and initialization:

```
// basic node structure
typedef struct __node_t {
        int key;
        struct __node_t *next;
} node_t;

// basic list structure (one used per list)
typedef struct __list_t {
        node_t *head;
        pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

(Cont.)
```


CONCURRENT LINKED LIST - 2

- Insert – adds item to list
- Everything is critical!
  - There are two unlocks

```
(Cont.)
int List_Insert(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                pthread_mutex_unlock(&L->lock);
        return -1; // fail }
        new->key = key;
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
        return 0; // success
}
(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.)

int List_Lookup(list_t *L, int key) {
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
                if (curr->key == key) {
                        pthread_mutex_unlock(&L->lock);
                        return 0; // success
                }
                curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return -1; // failure
}
```


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

```
void List_Init(list_t *L) {
        L->head = NULL;
        pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
        // synchronization not needed
        node_t *new = malloc(sizeof(node_t));
        if (new == NULL) {
                perror("malloc");
                return;
        }
        new->key = key;

        // just lock critical section
        pthread_mutex_lock(&L->lock);
        new->next = L->head;
        L->head = new;
        pthread_mutex_unlock(&L->lock);
}
```


CCL – SECOND IMPLEMENTATION - 2

- Lookup

(Cont.)

```
int List_Lookup(list_t *L, int key) {
        int rv = -1;
        pthread_mutex_lock(&L->lock);
        node_t *curr = L->head;
        while (curr) {
                if (curr->key == key) {
                        rv = 0;
                        break;
                }
                curr = curr->next;
        }
        pthread_mutex_unlock(&L->lock);
        return rv; // now both success and failure
}
```

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**

```
typedef struct __node_t {
        int value;
        struct __node_t *next;
} node_t;

typedef struct __queue_t {
        node_t *head;
        node_t *tail;
        pthread_mutex_t headLock;
        pthread_mutex_t tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
        node_t *tmp = malloc(sizeof(node_t));
        tmp->next = NULL;
        q->head = q->tail = tmp;
        pthread_mutex_init(&q->headLock, NULL);
        pthread_mutex_init(&q->tailLock, NULL);
}

(Cont.)
```


CONCURRENT QUEUE - 2

- **Add to queue**

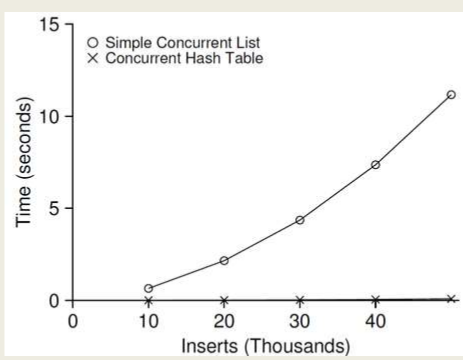
```
(Cont.)
void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        tmp->value = value;
        tmp->next = NULL;

        pthread_mutex_lock(&q->tailLock);
        q->tail->next = tmp;
        q->tail = tmp;
        pthread_mutex_unlock(&q->tailLock);
}
(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

```c
#define BUCKETS (101)

typedef struct __hash_t {
        list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
        int i;
        for (i = 0; i < BUCKETS; i++) {
                List_Init(&H->lists[i]);
        }
}

int Hash_Insert(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
        int bucket = key % BUCKETS;
        return List_Lookup(&H->lists[bucket], key);
}
```


Which is a major advantage of using concurrent data structures in your programs?

Locks are encapsulated within data structure code ensuring thread safety.

Lock granularity tradeoff already optimized inside data structurew

Multiple threads can more easily share data

All of the above

None of the above

LOCK-FREE DATA STRUCTURES

- Lock-free data structures in Java
- Java.util.concurrent.atomic package
- Classes:
  - AtomicBoolean
  - AtomicInteger
  - AtomicIntegerArray
  - AtomicIntegerFieldUpdater
  - AtomicLong
  - AtomicLongArray
  - AtomicLongFieldUpdater
  - AtomicReference
- See: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/package-summary.html

TCSS 422 WILL RETURN AT ~2:30PM

MIDTERM REVIEW

April 30, 2020 | TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma | L10.31

OBJECTIVES – 4/30

- Questions from 4/28
- C Tutorial (Apr 30 11:59p AOE)
- Assignment 1 (May 7 11:59p AOE)
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- Midterm Review

April 30, 2020 | TCSS422: Operating Systems [Spring 2020]
School of Engineering and Technology, University of Washington - Tacoma | L10.32

MIDTERM

- Tuesday May 5th
- ONLINE via Canvas (for 3 hrs 12:30 – 3:30p)
- Additional hour provided in case of internet issues, etc.
- Open book, note, internet
- Individual work

- ***Preparation:***
- Practice quiz: CPU scheduling (***to be posted***)
  - Auto grading w/ multiple attempts allowed as study aid
- Practice THURSDAY – first hour of lecture
  - Series of problems presented with some time to solve
  - Will then work through solutions
- Second hour – new material not on midterm

FIFO EXAMPLE

- Operation of CPU schedulers can be visualized with timing graphs.

- The graph below depicts a FIFO scheduler where three jobs arrive in the sequence A, B, C, where job A runs for 10 time slices, job B for 5 time slices, and job C for 10 time slices.

```
     |
FIFO |AAAAAAAAAABBBBBCCCCCCCCCC
     |________________________________________
     0          10   15        25
```

Q1- SHORTEST JOB FIRST (SJF) SCHEDULER

- Draw a scheduling graph for the SJF scheduler without preemption for the following jobs. Draw vertical lines for key events and be sure to label the X-axis times as in the example.

Job	Arrival Time	Job Length
A	T=0	25
B	T=5	10
C	T=10	15

```
        |
        |
SJF     |
        |
        |____________________________________________________________
        0
```


Q1 – SJF - 2

What is the response time (RT) and turnaround time (TT) for jobs A, B, and C?

RT Job A: ________________ TT Job A: ________________

RT Job B: ________________ TT Job B: ________________

RT Job C: ________________ TT Job C: ________________

What is the average response time for all jobs? _____________

What is the average turnaround time for all jobs? ___________

Q2 – SHORTEST TIME TO COMPLETION FIRST (STCF) SCHEDULER

Draw a scheduling graph for the STCF scheduler with preemption for the following jobs.

Draw vertical lines for key events and be sure to label the X-axis times as in the example.

Job	Arrival Time	Job Length
A	T=0	25
B	T=5	10
C	T=10	15

```
        |
        |
CPU     |
        |
        |_____________________________________________________________
        0
```


Q2 – STCF - 2

- What is the response time (RT) and turnaround time (TT) for jobs A, B, and C?

RT Job A: ________________ TT Job A: ________________

RT Job B: ________________ TT Job B: ________________

RT Job C: ________________ TT Job C: ________________

- What is the average response time for all jobs? ___________
- What is the average turnaround time for all jobs? __________

Q3 - OPERATING SYSTEM APIs

1. Provide a definition for what is a blocking API call

2. Provide a definition for a non-blocking API call

3. Provide an example of a blocking API call.
Consider APIs used to manage processes and/or threads.

4. Provide an example of a non-blocking API call.
Consider APIs used to manage processes and/or threads.

Q4 – OPERATING SYSTEM APIs - II

1. When implementing memory synchronization for a multi-threaded program list one **advantage** of combining the use of a condition variable with a lock variable via the Linux C thread API calls: `pthread_mutex_lock()` and `pthread_cond_wait()`

2. When implementing memory synchronization for a multi-threaded program using locks, list one **disadvantage** of using blocking thread API calls such as the Linux C thread API calls for: `pthread_mutex_lock()` and `pthread_cond_wait()`

3. List (2) factors that cause Linux blocking API calls to introduce **overhead** into programs:

Q5 – PERFECT MULTITASKING OPERATING SYSTEM

In a perfect-multi-tasking operating system, every process of the same priority will always receive exactly $1/n^{th}$ of the available CPU time. Important CPU improvements for multi-tasking include: (1) fast context switching to enable jobs to be swapped in-and-out of the CPU very quickly, and (2) the use of a timer interrupt to preempt running jobs without the user voluntarily yielding the CPU. These innovations have enabled major improvements towards achieving a coveted "Perfect Multi-Tasking System".

List and describe two challenges that remain complicating the full realization of a Perfect Multi-Tasking Operating System. In other words, what makes it very difficult for all jobs (for example, 10 jobs) of the same priority to receive **EXACTLY** the same runtime on the CPU? Your description must explain why the challenge is a problem for achieving perfect multi-tasking.

Q6 – ROUND-ROBIN SCHEDULER

Show a scheduling graph for a Round-Robin (RR) scheduler with job preemption where newly arriving jobs will immediately run. Assume a time slice of 3 timer units. Draw vertical lines for key events and be sure to label the X-axis times as in the example.

Job	Arrival Time	Job Length
A	T=0	25
B	T=5	10
C	T=10	15

```
        |
        |
RR      |
        |
        |______________________________________________________
        0
```

Q6 – RR SCHEDULER - 2

Using the graph, from time t=10 until all jobs complete at t=50, evaluate Jain's Fairness Index:

Jain's fairness index is expressed as:

$$\mathcal{J}(x_1, x_2, \ldots, x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} {x_i}^2}$$

Where n is the number of jobs, and x_i is the time share of each process Jain's fairness index=1 for best case fairness, and 1/n for worst case fairness.

For the time window from t=10 to t=50, what percentage of the CPU time is allocated to each of the jobs A, B, and C?

Job A: ________ Job B: ________ Job C: ________

With these values, calculate Jain's fairness index from t=10 to t=50.

April 30, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L10.43

Q7 – SLOPPY COUNTER

Below is a tradeoff space graph similar to those we've shown in class. Based on the sloppy counter threshold (S), add numbers on the **left** or **right** side of the graph for each of the following tradeoffs:

1. High number of Global Updates
2. High Performance
3. High Overhead
4. High Accuracy
5. Low number of Global Updates
6. Low Performance
7. Low Overhead
8. Low Accuracy

Low sloppy threshold (S) **High sloppy threshold (S)**

|____|

April 30, 2020	TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma	L10.44

MULTI-LEVEL FEEDBACK QUEUE

- Review the bonus lecture for examples of Multi-level-feedback-queue problems (MLFQ)
- https://tinyurl.com/y8ucda5z

April 30, 2020 | TCSS422: Operating Systems [Spring 2020] School of Engineering and Technology, University of Washington - Tacoma | L10.45

QUESTIONS

WILL RETURN IN A FEW MINUTES