# TCSS 422: OPERATING SYSTEMS

## Lock-based data structures, Midterm review

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



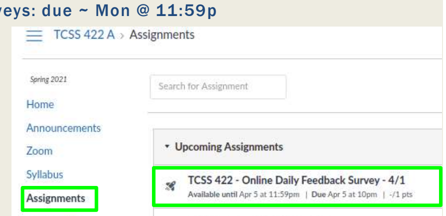
## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions

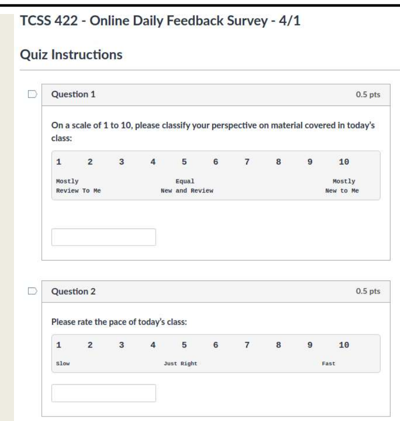


## ONLINE DAILY FEEDBACK SURVEY

- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 11:59p
- Thursday surveys: due ~ Mon @ 11:59p








## MATERIAL / PACE

- Please classify your perspective on material covered in today's class (57 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.89 (↑ - previous 6.56)**
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.85 (↑ - previous 5.73)**



## FEEDBACK

- ***What's difference between thread_wait and lock?***
  - What's difference between **pthread_cond_wait()** and **pthread_mutex_lock()**?
  - Pthread_mutex_lock() tries to obtain a lock protecting a critical section of code
    - The lock is either available (YES), or unavailable (NO)
    - If the lock is unavailable, then the API call will BLOCK indefinitely *i.e. forever*
    - When the lock is available, it is random which thread obtains it next
  - Pthread_cond_wait() adds signaling mechanism to manage locks
    - Other threads can wake up waiting threads
    - Order is based on a FIFO wait queue
    - Thread waiting longest obtains the lock first
    - State variable is used with condition variable to test if it is OKAY to proceed - ***If not we go back to sleep***

## FEEDBACK - 2

- ***What is a blocking API call?***
- Blocking API calls are system calls that interrupt execution of a thread until some resource can be obtained
- Examples from pthread API:
- pthread_mutex_lock() - block until lock is available
- pthread_cond_wait() - block until woken up-FIFO order
- pthread_join() - parent thread blocks until child thread completes
- ***What other blocking API calls have we discussed?***

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.7

## OBJECTIVES – 5/4

- Questions from 4/29
- **C Tutorial - Pointers, Strings, Exec in C**
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.8

## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- **Assignment 1 – May 11**
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.9

## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- **Quiz 2 – CPU Scheduling Algorithms**
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.10

## QUIZ 2 - CPU SCHEDULING ALGORITHMS

- Quiz posted on Canvas
- Due Wednesday May 5 @ 11:59p
- Provides CPU scheduling practice problems
  - FIFO, SJF, STCF, RR, MLFQ (Ch. 7 & 8)
- Unlimited attempts allowed
- Multiple choice and fill-in the blank
- Quiz automatically scored by Canvas
  - Please report any grading problems

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.11

## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- **Chapter 29: Lock Based Data Structures**
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.12

# CHAPTER 29 – LOCK BASED DATA STRUCTURES

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.13

## LOCK-BASED CONCURRENT DATA STRUCTURES

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

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.14

## COUNTER STRUCTURE W/O LOCK

- Synchronization weary --- not thread safe

```
typedef struct __counter_t {
        int value;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
}

void increment(counter_t *c) {
        c->value++;
}

void decrement(counter_t *c) {
        c->value--;
}

int get(counter_t *c) {
        return c->value;
}
```

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.15

## CONCURRENT COUNTER

```
typedef struct __counter_t {
        int value;
        pthread_lock_t lock;
} counter_t;

void init(counter_t *c) {
        c->value = 0;
        Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value++;
        Pthread_mutex_unlock(&c->lock);
}

```

- Add lock to the counter
- Require lock to change data

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.16

## CONCURRENT COUNTER - 2

- Decrease counter
- Get value

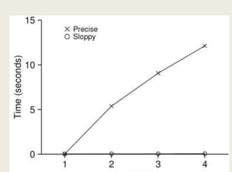
```
(Cont.)
void decrement(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        c->value--;
        Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
        Pthread_mutex_lock(&c->lock);
        int rc = c->value;
        Pthread_mutex_unlock(&c->lock);
        return rc;
}
```

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.17

## CONCURRENT COUNTERS - PERFORMANCE

- iMac: four core Intel 2.7 GHz i5 CPU
- Each thread increments counter 1,000,000 times




Synchronized counter scales poorly.

May 4, 2021 | TCSS422: Operating Systems [Spring 2021] School of Engineering and Technology, University of Washington - Tacoma | L11.18

## PERFECT SCALING

- Achieve (N) performance gain with (N) additional resources

- Throughput:
- Transactions per second (tps)

- 1 core
- N = 100 tps

- 10 cores (x10)
- N = 1000 tps (x10)



## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions



## 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 – MAIN POINTS

- Idea of Sloppy Counter is to ***RELAX*** the 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

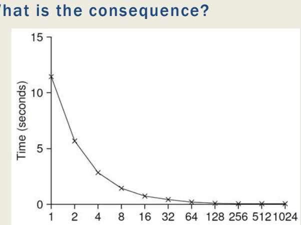
- Update threshold (S) = 5
- Synchronized across four CPU cores
- Threads update local CPU counters

| 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 COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity



## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - **Concurrent Structures: Linked List,** Queue, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions



## 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?



## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, **Queue**, Hash Table
- 2nd hour: Midterm Review
  - Practice Questions



## 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.)
```



## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, **Hash Table**
- 2nd hour: Midterm Review
  - Practice Questions



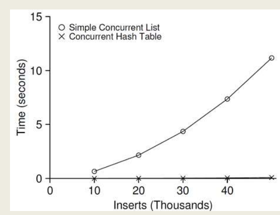
## CONCURRENT HASH TABLE

- 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

```
    #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

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

## 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



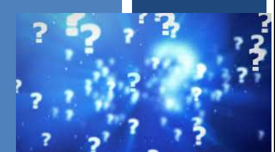
## WE WILL RETURN AT 5:00PM



## OBJECTIVES – 5/4

- Questions from 4/29
- C Tutorial - Pointers, Strings, Exec in C
- Assignment 1 – May 11
- Quiz 2 – CPU Scheduling Algorithms
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, **Queue**, Hash Table
- **2nd hour: Midterm Review**
  - Practice Questions





## MIDTERM REVIEW



## MIDTERM

- Thursday May 6th
- ONLINE via Canvas (for 3.5 hrs 3:40 – 7:10p)
- Test designed to take less than 2 hours
- Additional time provided in case of internet issues, etc.
- Open book, note, internet
- Individual work
- Coverage: all content up through Chapter 29, sloppy counter
- ***Preparation:***
- Practice quiz: Quiz 2: CPU scheduling (***posted***)
  - Auto grading w/ multiple attempts allowed as study aid
- Practice– second hour of lecture
  - Series of problems presented with some time to solve
  - Will then work through solutions



## 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 <u>multi-threaded program</u> 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 <u>multi-threaded program</u> 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 **<u>overhead</u>** 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:

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

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.



## 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)**



## MULTI-LEVEL FEEDBACK QUEUE

- Review the bonus lecture for examples of Multi-level-feedback-queue problems (MLFQ)
- **https://tinyurl.com/ky7usnjb**



## QUESTIONS

## QUESTIONS

## TCSS 422

## OFFICE HOURS

## *PLEASE SAY HELLO*

## OFFICE HOURS

## HAVE STEPPED OUT

## WILL RETURN SHORTLY