TCSS 422: OPERATING SYSTEMS

Linux Thread API, Lock Implementations, Lock-based data structures,

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

May 2, 2023 | TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma

1

OBJECTIVES – 5/2

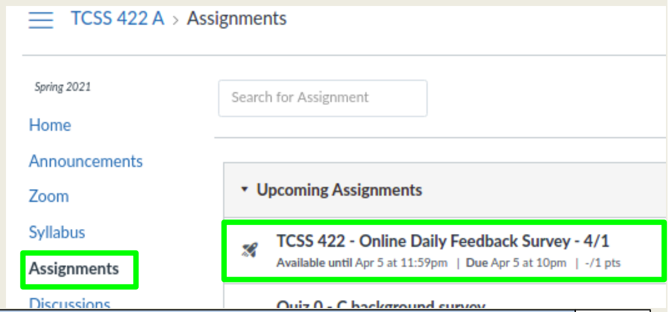
- **Questions from 4/27**
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Practice Midterm – 2^{nd} hour

May 2, 2023 | TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L11.2

2

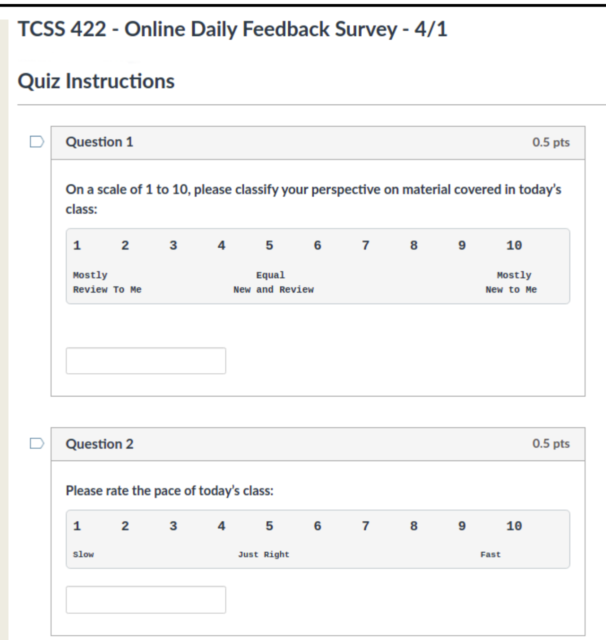
ONLINE DAILY FEEDBACK SURVEY

- 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



May 2, 2023 | TCSS422: Computer Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L11.3

3



May 2, 2023 | TCSS422: Computer Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L11.4

4

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (43 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.98 (↓ - previous 7.30)**

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 6.07 (↑ - previous 5.70)**

FEEDBACK FROM 4/27

OBJECTIVES – 5/2

- Questions from 4/27
- **C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28**
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Practice Midterm – 2nd hour

7

OBJECTIVES – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- **Assignment 1 - Due Tue May 9**
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Practice Midterm – 2nd hour

8

OBJECTIVES – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Practice Midterm – 2nd hour

QUIZ 2

- Canvas Quiz – Practice CPU Scheduling Problems

- Posted in Canvas
- Unlimited attempts permitted
- Provides CPU scheduling practice problems
  - FIFO, SJF, STCF, RR, MLFQ (Ch. 7 & 8)
- Multiple choice and fill-in the blank
- Quiz automatically scored by Canvas
  - Please report any grading problems

- Due Tuesday May 2nd at 11:59pm

- Link:
- https://canvas.uw.edu/courses/1642522/assignments/8316759

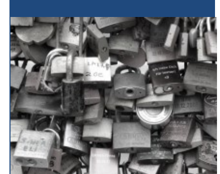
OBJECTIVES – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- **Chapter 28: Locks**
- Chapter 29: Lock Based Data Structures
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Practice Midterm – 2nd hour

May 2, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma | L11.11

11

CHAPTER 28 – LOCKS



May 2, 2023 | TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - T oma | L11.12

12

"LOCK BUILDING" CPU INSTRUCTIONS ON ARM PROCESSORS

- Two instructions used together to support synchronization on RISC systems
- These instructions are not on x86 processors
- They are on RISC CPUs: 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

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
            *ptr = value;
            return 1; // success!
    } else {
            return 0; // failed to update
    }
}
```

- 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

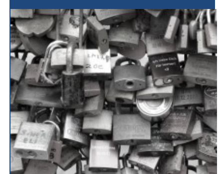
```c
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- Provides a two instruction lock

OBJECTIVES – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- **Chapter 29: Lock Based Data Structures**
  - Approximate Counter (Sloppy Counter)
  - Concurrent Structures: Linked List, Queue, Hash Table
- Practice Midterm – 2nd hour

CHAPTER 29 – LOCK BASED DATA STRUCTURES



17

LOCK-BASED CONCURRENT DATA STRUCTURES

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

18

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;
}
```


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

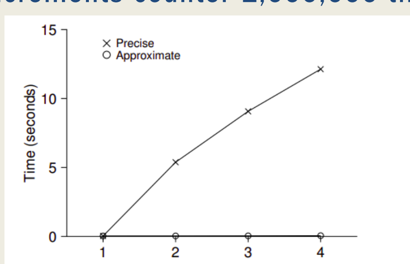
CONCURRENT COUNTER - 2

- 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;
}
```


CONCURRENT COUNTERS - PERFORMANCE

- Concurrent counter is considered a “precise counter”
- iMac: four core Intel 2.7 GHz i5 CPU
- Each thread increments counter 1,000,000 times



Precise counter scales poorly.

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)

- ***Is parallel counting with a shared counter an embarrassingly parallel problem?***

23

OBJECTIVES – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - **Approximate Counter (Sloppy Counter)**
  - Concurrent Structures: Linked List, Queue, Hash Table
- Practice Midterm – 2nd hour

24

APPROXIMATE (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
    - Update threshold (*S*) – *referred to as sloppiness threshold:* How often to push local values to global counter
    - 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?

May 2, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L11.25

25

APPROXIMATE COUNTER – MAIN POINTS

- Idea of the Approximate 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
- Approximate counter: trade-off accuracy for speed
  - It's approximate because it's not so accurate (until the end)

May 2, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L11.26

26

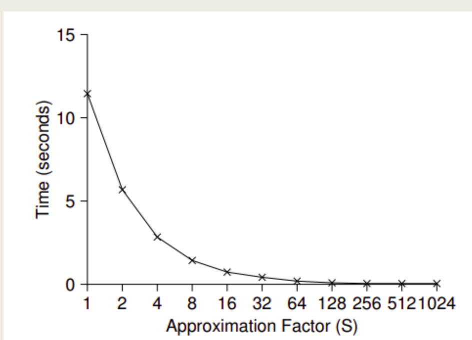
APPROXIMATE COUNTER - 2

- 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?



APPROXIMATE COUNTER - EXAMPLE

- Example implementation – sloppybasic.c
- Also with CPU affinity

29

When poll is active, respond at **pollev.com/wesleylloyd641**
Text **WESLEYLLOYD641** to **22333** once to join

Which of the following is NOT a problem as a result of having a low S-value for the approximate counter (Sloppy Counter) threshold?

- The counter overhead is very high.
- The counter implementation performs a very large number of LOCK/UNLOCK API calls.
- The global counter value is highly accurate.
- The counter performs very few local to global counter updates.
- 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**

30

OBJECTIVES – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - **Concurrent Structures: Linked List**, Queue, Hash Table
- Practice Midterm – 2nd hour

31

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.)
```


32

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 – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, **Queue**, Hash Table
- Practice Midterm – 2nd hour

39

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

40

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 – 5/2

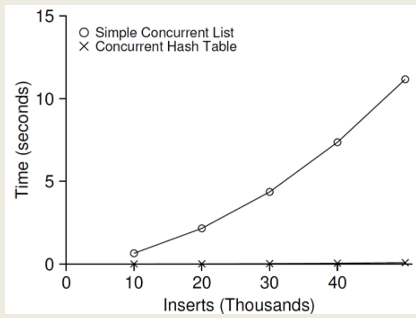
- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, **Hash Table**
- Practice Midterm – 2nd hour

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

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

47

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

48

OBJECTIVES – 5/2

- Questions from 4/27
- C Tutorial - Pointers, Strings, Exec in C - Due Fri Apr 28
- Assignment 1 - Due Tue May 9
- Quiz 1 (Due Thur Apr 27) – Quiz 2 (Due Tue May 2)
- Chapter 28: Locks
- Chapter 29: Lock Based Data Structures
  - Sloppy Counter
  - Concurrent Structures: Linked List, Queue, Hash Table
- **Practice Midterm – 2nd hour**

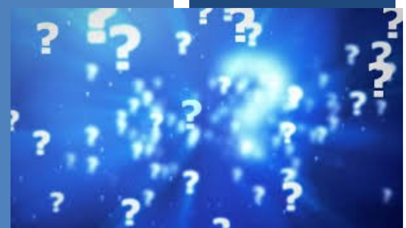
May 2, 2023 | TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L11.49

49

WE WILL RETURN AT 5:05PM

May 2, 2023 | TCSS422: Operating Systems [Spring 2023]
School of Engineering and Technology, University of Washington - Tacoma | L11.50

50



MIDTERM REVIEW

51

MIDTERM

- Thursday May 4th
- In Class in BHS 106 (2.0 hrs 3:40 – 5:40p)
- Test designed to take less than 2 hours
- Two pages of notes, double-sided, any-size paper permitted
- No book, other notes, cell phones, or internet
- Basic calculators OK
- 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

52

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? ___________

55

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
```


56

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.

61

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
```


62

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

NOTE: In the class, we mentioned how it is not clear at time t=13 if job A or B will be run.

The solution on the next page assumes job A will be run.

But a solution where job B runs next at time t=13 is also okay because the problem does not specify a rule.

```
        |
        |
RR      |
        |
        |_____________________________________________________________
        0
```


63

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.

64

Q6 - II

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

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)**

|____|

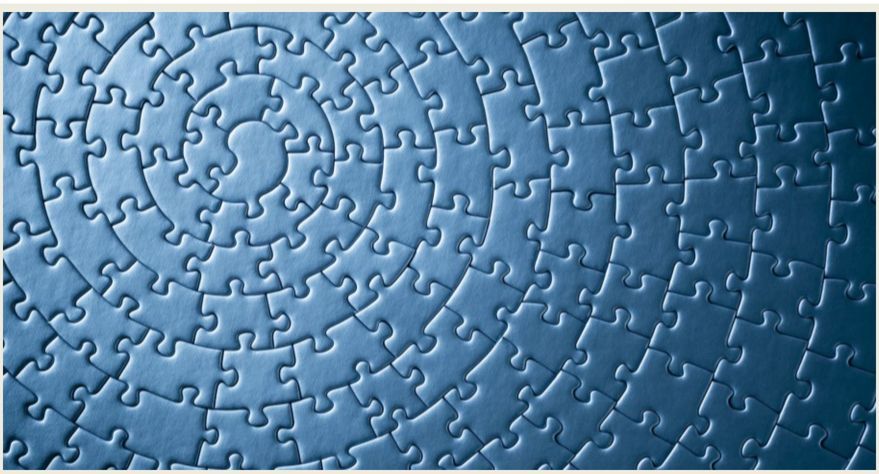
MULTI-LEVEL FEEDBACK QUEUE

- Review the bonus lecture for scheduling examples including several Multi-level-feedback-queue problems (MLFQ)
- https://tinyurl.com/4sepy582

May 2, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L11.67

67

SOLUTIONS



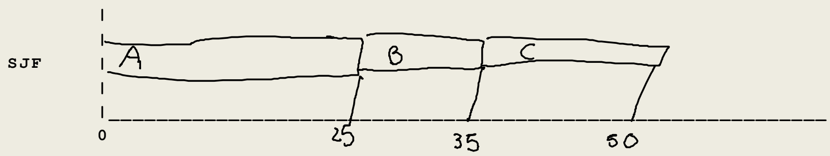
May 2, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L11.68

68

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



69

Q1 – SJF - 2

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

RT Job A: 0

TT Job A: 25

RT Job B: 25 - 5 = 20

TT Job B: 35 - 5 = 30

RT Job C: 35 - 10 = 25

TT Job C: 50 - 10 = 40

What is the average response time for all jobs? $\frac{0+20+25}{3} = \frac{45}{3} = 15$

What is the average turnaround time for all jobs? $\frac{25+30+40}{3} = \frac{95}{3}$ = $31.\overline{66}$

70

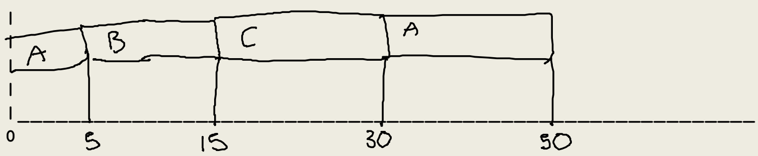
Q2 – SHORTEST TIME TO COMPLETION FIRST (STCF) SCHEDULER

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~~ 20
B	T=5	~~10~~ ~~5~~ 0
C	T=10	15

CPU



71

Q2 – STCF - 2

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

RT Job A: 0 | TT Job A: 50

RT Job B: 0 | TT Job B: 15 – 5 = 10

RT Job C: 15 – 10 = 5 | TT Job C: 30 – 10 = 20

- What is the average response time for all jobs? $\frac{0+0+5}{3} = \frac{5}{3} = 1.6\bar{6}$
- What is the average turnaround time for all jobs? $\frac{50+10+20}{3} = \frac{80}{3}$ → $26.6\bar{6}$

72

Q3 - OPERATING SYSTEM APIs

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

pthread_mutex_lock() — PROGRAMS FREEZES AND WAITS UNTIL A RESOURCE BECOMES AVAILABLE

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

PERFORMS ITS ACTION IMMEDIAETLY W/O WAIT

3. Provide an example of a blocking API call.

Consider APIs used to manage processes and/or threads.

wait() pthread_cond_wait() pthread_join

4. Provide an example of a non-blocking API call.

Consider APIs used to manage processes and/or threads.

pthread_mutex_unlock() fork() execvpe()

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()`

FIFO wait queue - ADDS threads in PRIORITY ORDER when WAITING

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()`

DEADLOCK CAN OCCUR which FREEZES the thread when the LOCK IS NEVER AVAILABLE

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

LONG WAITS when RESOURCE IS UNAVAILABLE

Context switch USER→KERNEL TO RUN the API

FINE GRAINED LOCK GRANULARITY - use APIs too much

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.

perfect fairness – jobs of the same priority

- jobs w/ different lengths + different arrival times
  hs to perfectly balance their runtime
- overhead from time tracking: measuring runtime can be inaccurate, and there is a cost for making measurements (measurements may not be precise)
- Context switching: how do account for overhead of c/s when balancing runtime — a busy system may have more context switches

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.

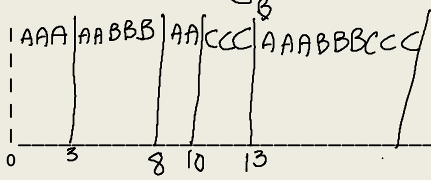
Job	Arrival Time	Job Length
A	T=0	25
B	T=5	10
C	T=10	15

NOTE: In the class, we mentioned how it is not clear at time t=13 if job A or B will be run.

The solution on the next page assumes job A will be run.

But a solution where job B runs next at time t=13 is also okay because the problem does not specify a rule.

RR



77

Q6 – ROUND-ROBIN SCHEDULER

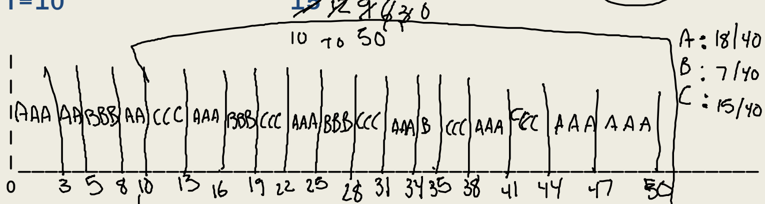
Show a scheduling graph for a Round-Robin (RR) scheduler with job preemption where newly arriving jobs will immediately run. Assume a time slice of 3 timer units. Draw vertical lines for key events and be sure to label the X-axis times as in the example.

ARRIVING JOBS ARE ADDED TO THE BACK OF THE RUNQUEUE AND THE JOB PTR WILL JUMP TO THE MOST RECENTLY ADDED JOB – AND CONTINUES IN RR FASHION

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

RUNQUEUE: ABC

RR



78

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: 18/40= .45 Job B: 7/40= .175 Job C: 15/40= .375

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

79

Q6 - II

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

worst case $\frac{1}{3}$ = .333

perfect 1

$(.45 + .175 + .375) = (1)^2 = 1$

$n \cdot \sum_{i=1}^{n} x_i^2 \rightarrow 3 \cdot ((.45)^2 + (.175)^2 + (.375)^2)$

$3 \cdot (.2025 + .030625 + .140625)$

$3 \cdot (.37375)$

1.12125

$\rightarrow \frac{1}{1.12125}$

$\rightarrow .8918617$

~ 89.2%

80

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) — 1 3 4 6

High sloppy threshold (S) — 2 5 7 8

|____|

May 2, 2023	TCSS422: Operating Systems [Spring 2023] School of Engineering and Technology, University of Washington - Tacoma	L11.81

81

QUESTIONS

82