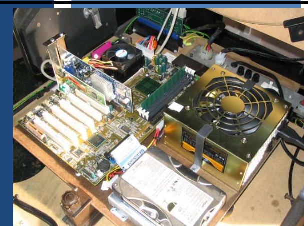
TCSS 422: OPERATING SYSTEMS



Locks API, Introduction to Locks, Lock-Based Data Structures

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

October 24, 2018 | TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

FEEDBACK FROM 10/22

- **What is the purpose of a void * pointer for pthread_create() ?**
  - It is used to store a memory location to a function
  - Thread's code pointer will reference this
- How do programmers balance coarse vs. fine-grained locking with applications that have UIs?
  - Does the UI have shared memory that is operated on in parallel by multiple threads?
  - Model-View-Controller:
    Can separate Controller and Model from View

October 24, 2018 | TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma | L9.2

FEEDBACK - 2

- Which is easier to implement fine-grained locking? Or coarse grained locking for parallel programming?
- If a thread is stuck in a lock state (*assume the thread is blocked waiting for the lock*) and the parent process (thread) does not call join, will the thread remain blocked after the parent process exits?
  - Who holds the lock? Does the parent? Some other thread?
  - If a thread existing causes the termination of the program, the program is likely dead.

FEEDBACK - 3

- Can you go over how to make MAKE files?
- Can the 2 pages of notes for the exam be typed/printer.

OBJECTIVES

- Quiz 2 Review
- Program 1 – MASH Shell (Friday 10/26)
- Midterm – (Wed 10/31)

- <u>Multi-threaded Programming</u>
- Chapter 28 – Introduction to Locks
- Chapter 29 – Lock-based Data Structures
- Chapter 30 -

October 24, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L9.5

CHAPTER 28 – LOCKS

October 24, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L9.6

LOCKS - 4

- Program can have many mutex (lock) variables to "serialize" many critical sections
- Locks are also used to protect data structures
  - Prevent multiple threads from changing the same data simultaneously
  - Programmer can make sections of code "granular"
    - Fine grained – means just one grain of sand at a time through an hour glass
  - Similar to relational database transactions
    - DB transactions prevent multiple users from modifying a table, row, field

October 24, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L9.7

FINE GRAINED?

- Is this code a good example of "*fine grained parallelism*"?

```
pthread_mutex_lock(&lock);
a = b++;
b = a * c;
*d = a + b +c;
FILE * fp = fopen ("file.txt", "r");
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
ListNode *node = mylist->head;
Int i=0
while (n
  node->
  node->subheading = str2;
  node->desc = str3;
  node->end = *e;
  node = node->next;
  i++
}
e = e - i;
pthread_mutex_unlock(&lock);
```

Example of coarse-grained parallelism

October 24, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L9.8

FINE GRAINED PARALLELISM

```
pthread_mutex_lock(&lock_a);
pthread_mutex_lock(&lock_b);
a = b++;
pthread_mutex_unlock(&lock_b);
pthread_mutex_unlock(&lock_a);

pthread_mutex_lock(&lock_b);
b = a * c;
pthread_mutex_unlock(&lock_b);

pthread_mutex_lock(&lock_d);
*d = a + b +c;
pthread_mutex_unlock(&lock_d);

FILE * fp = fopen ("file.txt", "r");
pthread_mutex_lock(&lock_e);
fscanf(fp, "%s %s %s %d", str1, str2, str3, &e);
pthread_mutex_unlock(&lock_e);

ListNode *node = mylist->head;
int i=0 . . .
```

EVALUATING LOCK IMPLEMENTATIONS

- Correctness
  - Does the lock work?
  - Are critical sections mutually exclusive? (atomic-*as a unit*?)

- Fairness
  - Are threads competing for a lock have a fair chance of acquiring it?
- Overhead

BUILDING LOCKS

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

HISTORICAL IMPLEMENTATION

- To implement mutual exclusion
  - Disable interrupts upon entering critical sections

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

- Any thread could disable system-wide interrupt
  - What if lock is never released?
- On a multiprocessor processor each CPU has its own interrupts
  - Do we disable interrupts for all cores simultaneously?
- While interrupts are disabled, they could be lost
  - If not queued…

SPIN LOCK IMPLEMENTATION

- Operate without atomic-*as a unit* assembly instructions
- "Do-it-yourself" Locks
- Is this lock implementation: Correct? Fair? Performant?

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 → lock is available, 1 → held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
            ;  // spin-wait (do nothing)
    mutex->flag = 1;  // now SET it !
}

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


DIY: CORRECT?

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

Thread1	Thread2
call `lock()`	
`while (flag == 1)`	
interrupt: switch to Thread 2	
	call `lock()`
	`while (flag == 1)`
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1;` // set flag to 1 (too!)	

- Here both threads have "acquired" the lock simultaneously

DIY: PERFORMANT?

```
void lock(lock_t *mutex)
{
  while (mutex->flag == 1);    // while lock is unavailable, wait…
  mutex->flag = 1;
}
```

- What is wrong with while(<cond>); ?
- Spin-waiting wastes time actively waiting for another thread
- while (1); will "peg" a CPU core at 100%
  - Continuously loops, and evaluates mutex->flag value…
  - Generates heat…

TEST-AND-SET INSTRUCTION

- C implementation: not atomic
  - Adds a simple check to basic spin lock
  - One a single core CPU system with preemptive scheduler:
  - Try this…

```
int TestAndSet(int *ptr, int new) {
    int old = *ptr;   // fetch old value at ptr
    *ptr = new;       // store 'new' into ptr
    return old;       // return the old value
}
```

- lock() method checks that TestAndSet doesn't return 1
- Comparison is in the caller
- Single core systems are becoming scarce
- Try on a one-core VM

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

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ;        // spin-wait
}

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


SPIN LOCK EVALUATION

- Correctness:
  - Spin locks guarantee: critical sections won't be executed simultaneously by (2) threads
- Fairness:
  - No fairness guarantee. Once a thread has a lock, nothing forces it to relinquish it…
- Performance:
  - Spin locks perform "busy waiting"
  - Spin locks are best for short periods of waiting
  - Performance is slow when multiple threads share a CPU
    - Especially for long periods

COMPARE AND SWAP

- Checks that the lock variable has the expected value FIRST, before changing its value
  - If so, make assignment
  - Return value at location
- Adds a comparison to TestAndSet
- Useful for wait-free synchronization
  - Supports implementation of shared data structures which can be updated atomically (*as a unit*) using the HW support CompareAndSwap instruction
  - Shared data structure updates become "wait-free"
  - Upcoming in Chapter 32

October 24, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L9.19

COMPARE AND SWAP

- Compare and Swap

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
```

- Spin loc

```
        ; // spin
}
```

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

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

October 24, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L9.20

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

```
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
    }
}
```

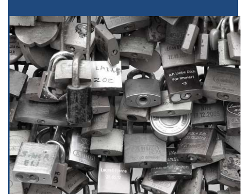
- LL instruction loads pointer value (ptr)
- SC only stores if the load link pointer has not changed
- Requires HW support
  - C code is psuedo code

LL/SC LOCK - 2

```
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;
}
```

- Two instruction lock

CHAPTER 29 – LOCK BASED DATA STRUCTTURES



OBJECTIVES

- Chapter 29
  - 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

```c
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

```c
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

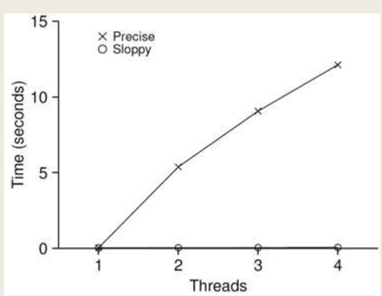
CONCURRENT COUNTER - 2

- 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

- 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?

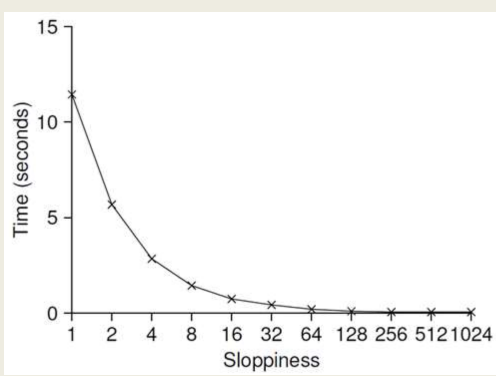
SLOPPY COUNTER - 2

- 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:

```
// 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

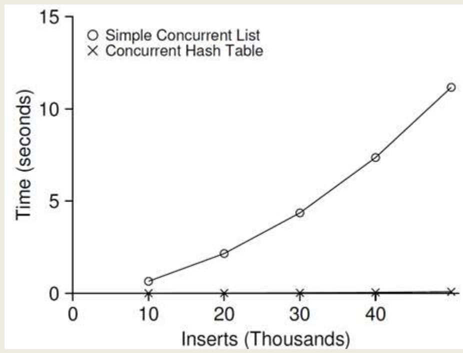
```
(Cont.)
void Queue_Enqueue(queue_t *q, int value) {
        node_t *tmp = malloc(sizeof(node_t));
        assert(tmp != NULL);

        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

```
#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);
}
```

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/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html

CHAPTER 30 – CONDITION VARIABLES

CONDITION VARIABLES

- There are many cases where a thread wants to wait for another thread before proceeding with execution
- Consider when a precondition must be fulfilled before it is meaningful to proceed …

CONDITION VARIABLES - 2

- Support a signaling mechanism to alert threads when preconditions have been satisfied
- Eliminate busy waiting
- Alert one or more threads to "consume" a result, or respond to state changes in the application
- Threads are placed on an **explicit queue** (FIFO) to wait for signals
- **Signal**: wakes one thread
  broadcast wakes all (ordering by the OS)

CONDITION VARIABLES - 3

- Condition variable

```
pthread cond t c;
```

  - Requires initialization

- Condition API calls

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);    // wait()
pthread_cond_signal(pthread_cond_t *c);                      // signal()
```

- wait() accepts a mutex parameter
  - Releases lock, puts thread to sleep
- signal()
  - Wakes up thread, awakening thread acquires lock

CONDITION VARIABLES - QUESTIONS

- **Why would we want to put waiting threads on a queue… why not use a stack?**
  - Queue (FIFO), Stack (LIFO)
  - Using condition variables eliminates busy waiting by putting threads to "sleep" and yielding the CPU.
- **Why do we want to not busily wait for the lock to become available?**
- A program has 10-threads, where 9 threads are waiting. The working thread finishes and broadcasts that the lock is available. **What happens next?**

MATRIX GENERATOR

Matrix generation example

Chapter 30
signal.c

MATRIX GENERATOR

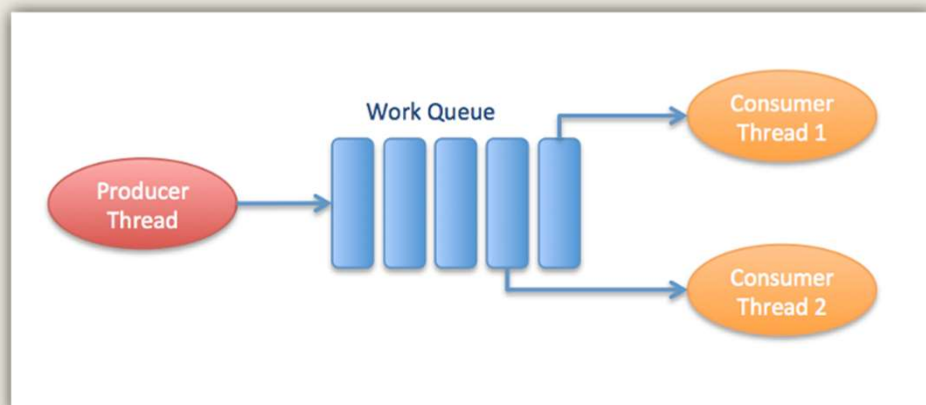
- The main thread, and worker thread (generates matrices) share a single matrix pointer.
- What would happen if we don't use a condition variable to coordinate exchange of the lock?
- Let's try "nosignal.c"

SUBTLE RACE CONDITION: WITHOUT A WHILE

```
void thr_exit() {
        done = 1;
        Pthread_cond_signal(&c);
}

void thr_join() {
        if (done == 0)
                Pthread_cond_wait(&c);
}
```

- Parent thread calls thr_join() and executes the comparison
- The context switches to the child
- The child runs thr_exit() and signals the parent, but the parent is not waiting yet.
- The signal is lost
- The parent deadlocks

PRODUCER / CONSUMER



PRODUCER / CONSUMER

- Producer
  - Produces items – consider the child matrix maker
  - Places them in a buffer
    - Example: the buffer is only 1 element (single array pointer)
- Consumer
  - Grabs data out of the buffer
  - Our example: parent thread receives dynamically generated matrices and performs an operation on them
    - Example: calculates average value of every element (integer)
- Multithreaded web server example
  - Http requests placed into work queue; threads process

PRODUCER / CONSUMER - 2

- Producer / Consumer is also known as **Bounded Buffer**

- Bounded buffer
  - Similar to piping output from one Linux process to another
  - grep pthread signal.c | wc –l
  - Synchronized access:
    sends output from grep → wc as it is produced
  - File stream

PUT/GET ROUTINES

- Buffer is a one element shared data structure (int)
- Producer "puts" data
- Consumer "gets" data
- Shared data structure requires synchronization

```
int buffer;
int count = 0;    // initially, empty

void put(int value) {
        assert(count == 0);
        count = 1;
        buffer = value;
}

int get() {
        assert(count == 1);
        count = 0;
        return buffer;
}
```


PRODUCER / CONSUMER - 3

- Producer adds data
- Consumer removes data (busy waiting)
- Will this code work (spin locks) with 2-threads?
  1. Producer 2. Consumer

```
void *producer(void *arg) {
        int i;
        int loops = (int) arg;
        for (i = 0; i < loops; i++) {
                put(i);
        }
}

void *consumer(void *arg) {
        int i;
        while (1) {
                int tmp = get();
                printf("%d\n", tmp);
        }
}
```

PRODUCER / CONSUMER - 3

- The shared data structure needs synchronization!

Producer

```
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // p1
        if (count == 1)                          // p2
            Pthread_cond_wait(&cond, &mutex);    // p3
        put(i);                                  // p4
        Pthread_cond_signal(&cond);              // p5
        Pthread_mutex_unlock(&mutex);            // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // c1
```

October 24, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L9.63

PRODUCER/CONSUMER - 4

```
        if (count == 0)                          // c2
            Pthread_cond_wait(&cond, &mutex);    // c3
        int tmp = get();                         // c4
        Pthread_cond_signal(&cond);              // c5
        Pthread_mutex_unlock(&mutex);            // c6
        printf("%d\n", tmp);
    }
}
```

Consumer

- This code as-is works with just:
  - (1) Producer
  - (1) Consumer
- If we scale to (2+) consumer's it fails
  - How can it be fixed ?

October 24, 2018 | TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma | L9.64

EXECUTION TRACE:
NO WHILE, 1 PRODUCER, 2 CONSUMERS

- Two threads

Legend
c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in …
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	… and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

PRODUCER/CONSUMER SYNCHRONIZATION

- When producer threads awake, they do not check if there is any data in the buffer…
  - Need while, not if
- What if T_p puts a value, wakes T_{c1} whom consumes the value
- Then T_p has a value to put, but T_{c1}'s signal on &cond wakes T_{c2}
- There is nothing for T_{c2} consume, so T_{c2} sleeps
- T_{c1}, T_{c2}, and T_p all sleep forever
- T_{c1} needs to wake T_p to T_{c2}

EXECUTION TRACE:
WHILE, 1 CONDITION, 1 PRODUCER, 2 CONSUMERS

Legend
c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	**Oops! Woke T_{c2}**

EXECUTION TRACE – 2
WHILE, 1 CONDITION, 1 PRODUCER, 2 CONSUMERS

- T_{c2} runs, no data to consume

Legend
c1/p1- lock
c2/p2- check var
c3/p3- wait
c4- put()
p4- get()
c5/p5- signal
c6/p6- unlock

T_{c1}	State	T_{c2}	State	T_p	State	Count	Comment
...	...	...	...	...	...	...	*(cont.)*
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	**Everyone asleep ...**

TWO CONDITIONS

- Use two condition variables: empty & full
  - One condition handles the producer
  - the other the consumer

```
cond_t empty, full;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal( &full);
        Pthread_mutex_unlock(&mutex);
    }
}
```


FINAL PRODUCER/CONSUMER

- Change buffer from int, to int buffer[MAX]
- Add indexing variables

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```

FINAL P/C - 2

```
cond_t empty, full
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // p1
        while (count == MAX)                     // p2
            Pthread_cond_wait(&empty, &mutex);   // p3
        put(i);                                  // p4
        Pthread_cond_signal (&full);             // p5
        Pthread_mutex_unlock(&mutex);            // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);              // c1
        while (count == 0)                       // c2
            Pthread_cond_wait( &full, &mutex);   // c3
        int tmp = get();                         // c4
```


FINAL P/C - 3

```
(Cont.)
        Pthread_cond_signal(&empty);             // c5
        Pthread_mutex_unlock(&mutex);            // c6
        printf("%d\n", tmp);
    }
}
```

- Producer: only sleeps when buffer is full
- Consumer: only sleeps if buffers are empty

COVERING CONDITIONS

- A condition that covers ***all*** cases (conditions):
- Excellent use case for **pthread_cond_broadcast**
- Consider memory allocation:
  - When a program deals with huge memory allocation/deallocation on the heap
  - Access to the heap must be managed when memory is scarce

    PREVENT: Out of memory:
    - queue requests until memory is free
  - Which thread should be woken up?

COVERING CONDITIONS - 2

```
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...;                // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);        //
    Pthread_mutex_unlock(&m);
}
```

Check available memory

Broadcast

COVER CONDITIONS - 3

- Broadcast awakens all blocked threads requesting memory
- Each thread evaluates if there's enough memory: (bytesLeft < size)
  - Reject: requests that cannot be fulfilled- go back to sleep
    - *Insufficient memory*
  - Run: requests which can be fulfilled
    - with newly available memory!
- Overhead
  - Many threads may be awoken which can't execute

QUESTIONS