TCSS 422: OPERATING SYSTEMS

Lock-based Data Structures, Condition Variables

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

OBJECTIVES

- Assignment 1 – 2/15
- Midterm 2/13 (???)
- Feedback 2/6

- **Parallel programming with P-threads cont'd**
- Chapter 29 – Lock-Based Data Structures
- Chapter 30 – Condition Variables
- Practice midterm questions

FEEDBACK FROM 2/6

- **Is `pthread_mutex_trylock()` just `pthread_mutex_timelock()` with a default time?**
- No. Pthread_mutex_trylock() is closer to pthread_mutex_lock()
- From the manpages:
- pthread_mutex_trylock() function is equivalent to pthread_mutex_lock(), except that if the mutex object referenced by mutex is currently locked (by any thread, including the current thread), the call shall return immediately
- It is a non-blocking API that tries to obtain the lock

FEEDBACK - 2

- **Threading use cases:**
- **Is it more common to create threads in advance of them being needed, or to create them when conditions to use them are met (*on demand*)?**
- Many server applications will tend to utilize thread pools, preinitialized sets of threads that are repurposed to process different requests in a system
  - Latency from thread creation may impact multiple users by slowing down the server
- Client applications are may be more likely to create threads on demand
  - Latency from thread creation only impacts one user

FEEDBACK - 3

- **Counters …?**
- **We ended on Wednesday Feb 6th introducing the *sloppy counter*, and now resume this example**

CHAPTER 29 – LOCK BASED DATA STRUCTURES

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

```
1       typedef struct __counter_t {
2               int value;
3       } counter_t;
4
5       void init(counter_t *c) {
6               c->value = 0;
7       }
8
9       void increment(counter_t *c) {
10              c->value++;
11      }
12
13      void decrement(counter_t *c) {
14              c->value--;
15      }
16
17      int get(counter_t *c) {
18              return c->value;
19      }
```


CONCURRENT COUNTER

```
1       typedef struct __counter_t {
2               int value;
3               pthread_lock_t lock;
4       } counter_t;
5
6       void init(counter_t *c) {
7               c->value = 0;
8               Pthread_mutex_init(&c->lock, NULL);
9       }
10
11      void increment(counter_t *c) {
12              Pthread_mutex_lock(&c->lock);
13              c->value++;
14              Pthread_mutex_unlock(&c->lock);
15      }
16
```

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

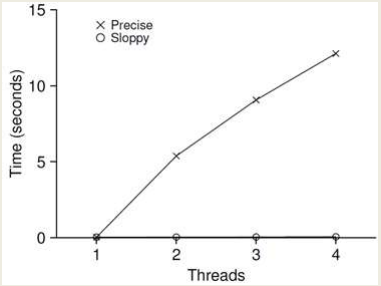
CONCURRENT COUNTER - 2

- Decrease counter
- Get value

```
(Cont.)
17      void decrement(counter_t *c) {
18              Pthread_mutex_lock(&c->lock);
19              c->value--;
20              Pthread_mutex_unlock(&c->lock);
21      }
22
23      int get(counter_t *c) {
24              Pthread_mutex_lock(&c->lock);
25              int rc = c->value;
26              Pthread_mutex_unlock(&c->lock);
27              return rc;
28      }
```


CONCURRENT COUNTERS - PERFORMANCE

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



Traditional vs. sloppy counter
Sloppy Threshold (S) = 1024

Synchronized counter scales poorly.

PERFECT SCALING

- Achieve (N) performance gain with (N) additional resources
- Throughput:
- Transactions per second
- 1 core
- N = 100 tps
- 10 core
- N = 1000 tps

SLOPPY COUNTER

- Provides single logical shared counter
  - Implemented using local counters for each ~CPU core
    - 4 CPU cores = 4 local counters & 1 global counter
    - Local counters are synchronized via local locks
  - Global counter is updated periodically
    - Global counter has lock to protect global counter value
    - Sloppiness threshold (*S*): Update threshold of global counter with local values
    - Small (S): more updates, more overhead
    - Large (S): fewer updates, more performant, less synchronized
- Why this implementation? Why do we want counters local to each CPU Core?

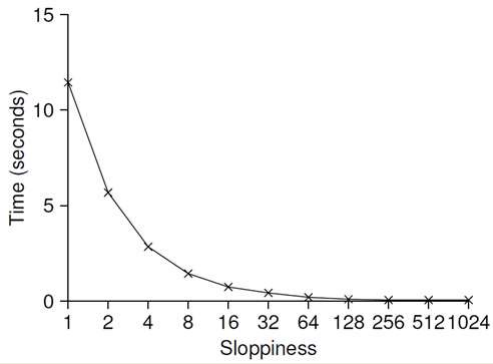
SLOPPY COUNTER - 2

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

Time	L_1	L_2	L_3	L_4	G
0	0	0	0	0	0
1	0	0	1	1	0
2	1	0	2	1	0
3	2	0	3	1	0
4	3	0	3	2	0
5	4	1	3	3	0
6	5 → 0	1	3	4	5 (from L_1)
7	0	2	4	5 → 0	10 (from L_4)

THRESHOLD VALUE S

- Consider 4 threads increment a counter 1000000 times each
- Low S → What is the consequence?
- High S → What is the consequence?



SLOPPY COUNTER - EXAMPLE

- Example implementation
- Also with CPU affinity

CONCURRENT LINKED LIST - 1

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

```
1       // basic node structure
2       typedef struct __node_t {
3               int key;
4               struct __node_t *next;
5       } node_t;
6
7       // basic list structure (one used per list)
8       typedef struct __list_t {
9               node_t *head;
10              pthread_mutex_t lock;
11      } list_t;
12
13      void List_Init(list_t *L) {
14              L->head = NULL;
15              pthread_mutex_init(&L->lock, NULL);
16      }
17
(Cont.)
```

CONCURRENT LINKED LIST - 2

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

```
(Cont.)
18      int List_Insert(list_t *L, int key) {
19              pthread_mutex_lock(&L->lock);
20              node_t *new = malloc(sizeof(node_t));
21              if (new == NULL) {
22                      perror("malloc");
23                      pthread_mutex_unlock(&L->lock);
24              return -1; // fail
26              new->key = key;
27              new->next = L->head;
28              L->head = new;
29              pthread_mutex_unlock(&L->lock);
30              return 0; // success
31      }
(Cont.)
```


CONCURRENT LINKED LIST - 3

- Lookup – checks list for existence of item with key
- Once again everything is critical
  - Note - there are also two unlocks

```
(Cont.)
32
32      int List_Lookup(list_t *L, int key) {
33              pthread_mutex_lock(&L->lock);
34              node_t *curr = L->head;
35              while (curr) {
36                      if (curr->key == key) {
37                              pthread_mutex_unlock(&L->lock);
38                              return 0; // success
39                      }
40                      curr = curr->next;
41              }
42              pthread_mutex_unlock(&L->lock);
43              return -1; // failure
44      }
```

CONCURRENT LINKED LIST

- First Implementation:
  - Lock **everything** inside Insert() and Lookup()
  - If malloc() fails lock must be released
    - Research has shown "***exception-based control flow***" to be error prone
    - 40% of Linux OS bugs occur in rarely taken code paths
    - Unlocking in an exception handler is considered a poor coding practice
    - There is nothing specifically wrong with this example however
- Second Implementation …

CCL – SECOND IMPLEMENTATION

- Init and Insert

```
1       void List_Init(list_t *L) {
2               L->head = NULL;
3               pthread_mutex_init(&L->lock, NULL);
4       }
5
6       void List_Insert(list_t *L, int key) {
7               // synchronization not needed
8               node_t *new = malloc(sizeof(node_t));
9               if (new == NULL) {
10                      perror("malloc");
11                      return;
12              }
13              new->key = key;
14
15              // just lock critical section
16              pthread_mutex_lock(&L->lock);
17              new->next = L->head;
18              L->head = new;
19              pthread_mutex_unlock(&L->lock);
20      }
21
```

CCL – SECOND IMPLEMENTATION - 2

- Lookup

```
(Cont.)
22      int List_Lookup(list_t *L, int key) {
23              int rv = -1;
24              pthread_mutex_lock(&L->lock);
25              node_t *curr = L->head;
26              while (curr) {
27                      if (curr->key == key) {
28                              rv = 0;
29                              break;
30                      }
31                      curr = curr->next;
32              }
33              pthread_mutex_unlock(&L->lock);
34              return rv; // now both success and failure
35      }
```


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

```
1       typedef struct __node_t {
2               int value;
3               struct __node_t *next;
4       } node_t;
5
6       typedef struct __queue_t {
7               node_t *head;
8               node_t *tail;
9               pthread_mutex_t headLock;
10              pthread_mutex_t tailLock;
11      } queue_t;
12
13      void Queue_Init(queue_t *q) {
14              node_t *tmp = malloc(sizeof(node_t));
15              tmp->next = NULL;
16              q->head = q->tail = tmp;
17              pthread_mutex_init(&q->headLock, NULL);
18              pthread_mutex_init(&q->tailLock, NULL);
19      }
20
(Cont.)
```

CONCURRENT QUEUE - 2

- Add to queue

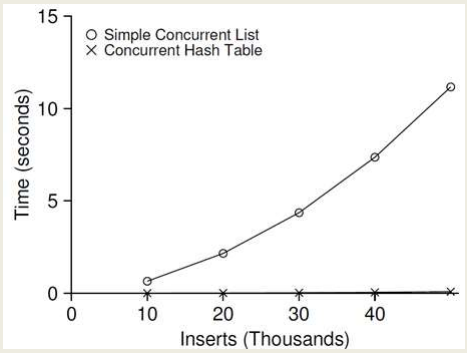
```
(Cont.)
21      void Queue_Enqueue(queue_t *q, int value) {
22              node_t *tmp = malloc(sizeof(node_t));
23              assert(tmp != NULL);
24
25              tmp->value = value;
26              tmp->next = NULL;
27
28              pthread_mutex_lock(&q->tailLock);
29              q->tail->next = tmp;
30              q->tail = tmp;
31              pthread_mutex_unlock(&q->tailLock);
32      }
(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

```
1       #define BUCKETS (101)
2
3       typedef struct __hash_t {
4               list_t lists[BUCKETS];
5       } hash_t;
6
7       void Hash_Init(hash_t *H) {
8               int i;
9               for (i = 0; i < BUCKETS; i++) {
10                      List_Init(&H->lists[i]);
11              }
12      }
13
14      int Hash_Insert(hash_t *H, int key) {
15              int bucket = key % BUCKETS;
16              return List_Insert(&H->lists[bucket], key);
17      }
18
19      int Hash_Lookup(hash_t *H, int key) {
20              int bucket = key % BUCKETS;
21              return List_Lookup(&H->lists[bucket], key);
22      }
```

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

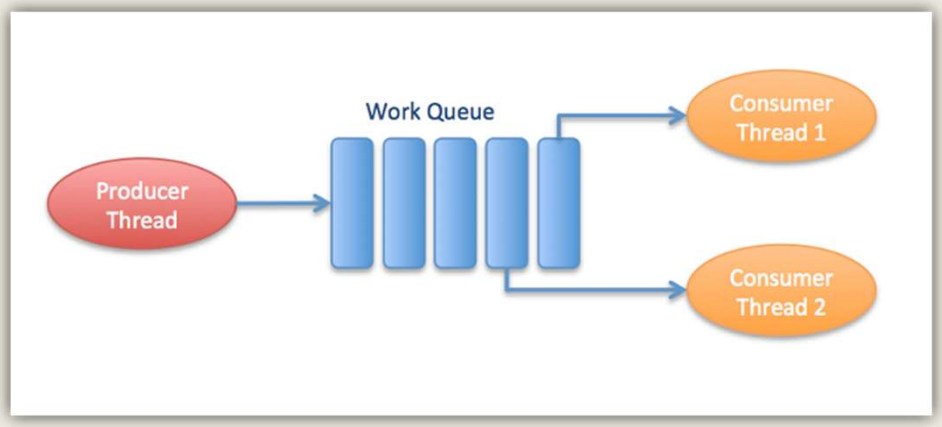
- The main thread, and worker thread (generates matrices) share a single matrix pointer.
- What would happen if we don't use a condition variable to coordinate exchange of the lock?
- Let's try "nosignal.c"

SUBTLE RACE CONDITION: WITHOUT A WHILE

```
1       void thr_exit() {
2               done = 1;
3               Pthread_cond_signal(&c);
4       }
5
6       void thr_join() {
7               if (done == 0)
8                       Pthread_cond_wait(&c);
9       }
```

- 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

```
1       int buffer;
2       int count = 0;   // initially, empty
3
4       void put(int value) {
5               assert(count == 0);
6               count = 1;
7               buffer = value;
8       }
9
10      int get() {
11              assert(count == 1);
12              count = 0;
13              return buffer;
14      }
```


PRODUCER / CONSUMER - 3

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

```
1       void *producer(void *arg) {
2               int i;
3               int loops = (int) arg;
4               for (i = 0; i < loops; i++) {
5                       put(i);
6               }
7       }
8
9       void *consumer(void *arg) {
10              int i;
11              while (1) {
12                      int tmp = get();
13                      printf("%d\n", tmp);
14              }
15      }
```

PRODUCER / CONSUMER - 3

- The shared data structure needs synchronization!

```
1       cond_t cond;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i;
6           for (i = 0; i < loops; i++) {
7               Pthread_mutex_lock(&mutex);            // p1
8               if (count == 1)                        // p2
9                   Pthread_cond_wait(&cond, &mutex);  // p3
10              put(i);                                // p4
11              Pthread_cond_signal(&cond);            // p5
12              Pthread_mutex_unlock(&mutex);          // p6
13          }
14      }
15
16      void *consumer(void *arg) {
17          int i;
18          for (i = 0; i < loops; i++) {
19              Pthread_mutex_lock(&mutex);            // c1
```

Producer

PRODUCER/CONSUMER - 4

```
20              if (count == 0)                        // c2
21                  Pthread_cond_wait(&cond, &mutex);  // c3
22              int tmp = get();                       // c4
23              Pthread_cond_signal(&cond);            // c5
24              Pthread_mutex_unlock(&mutex);          // c6
25              printf("%d\n", tmp);
26          }
27      }
```

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 ?

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

```
1   cond_t empty, full;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);
8           while (count == 1)
9               Pthread_cond_wait(&empty, &mutex);
10          put(i);
11          Pthread_cond_signal( &full);
12          Pthread_mutex_unlock(&mutex);
13      }
14  }
15
```


FINAL PRODUCER/CONSUMER

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

```
1   int buffer[MAX];
2   int fill = 0;
3   int use = 0;
4   int count = 0;
5
6   void put(int value) {
7       buffer[fill] = value;
8       fill = (fill + 1) % MAX;
9       count++;
10  }
11
12  int get() {
13      int tmp = buffer[use];
14      use = (use + 1) % MAX;
15      count--;
16      return tmp;
17  }
```

FINAL P/C - 2

```
1   cond_t empty, full
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);              // p1
8           while (count == MAX)                     // p2
9               Pthread_cond_wait(&empty, &mutex);   // p3
10          put(i);                                  // p4
11          Pthread_cond_signal (&full);             // p5
12          Pthread_mutex_unlock(&mutex);            // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);              // c1
20          while (count == 0)                       // c2
21              Pthread_cond_wait( &full, &mutex);   // c3
22          int tmp = get();                         // c4
```


FINAL P/C - 3

```
(Cont.)
23          Pthread_cond_signal(&empty);             // c5
24          Pthread_mutex_unlock(&mutex);            // c6
25          printf("%d\n", tmp);
26      }
27  }
```

- 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

```
1       // how many bytes of the heap are free?
2       int bytesLeft = MAX_HEAP_SIZE;
3
4       // need lock and condition too
5       cond_t c;
6       mutex_t m;
7
8       void *
9       allocate(int size) {
10          Pthread_mutex_lock(&m);
11          while (bytesLeft < size)
12              Pthread_cond_wait(&c, &m);
13          void *ptr = ...;              // get mem from heap
14          bytesLeft -= size;
15          Pthread_mutex_unlock(&m);
16          return ptr;
17      }
18
19      void free(void *ptr, int size) {
20          Pthread_mutex_lock(&m);
21          bytesLeft += size;
22          Pthread_cond_signal(&c);      //
23          Pthread_mutex_unlock(&m);
24      }
```

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