CO3

Session - 3

COURSE NAME: OPERATING SYSTEMS
COURSE CODE: 23CS2104R/A

LOCKS, LOCKED DATA STRUCTURES, CONDITION VARIABLES, MUTEX.

AIM OF THE SESSION

To familiarize students with the basic concept of locks.

INSTRUCTIONAL OBJECTIVES

This Session is designed to:

1. Demonstrate what is meant by Lock.
2. Describe the types of Locks.
3. Describe the Locked Data Structures.
4. Describe the Concurrent Counters with Locks.

LEARNING OUTCOMES

At the end of this session, you should be able to:

1. Defines what is a lock.
2. Describe Test and Set and Compare and Swap.
3. Summarize the Role of Mutex Locks.

SYNCHRONIZATION HARDWARE

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

LOCKS:THE BASIC IDEA

- Ensure that any **critical section** executes as if it were a single atomic instruction.
  - An example: the canonical update of a shared variable

```
balance = balance + 1;
```

- Add some code around the critical section

lock_t mutex; // some globally-allocated lock 'mutex'
…
lock(&mutex);
balance = balance + 1; unlock(&mutex);

CONTD..

- Lock variable holds the state of the lock.
  - available (or unlocked or free)
    - No thread holds the lock.
  - acquired (or locked or held)
    - Exactly one thread holds the lock and presumably is in a critical section.

THE SEMANTICS OF THE LOCK()

- **lock()**
  - **Try to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock.
  - **Enter** the critical section.
    - This thread is said to be the owner of the lock.
  - Other threads are prevented from entering the critical section while the first thread that holds the lock is in there.

EVALUATING LOCKS – BASIC CRITERIA

- **Mutual exclusion**
  - Does the lock work, preventing multiple threads from entering a critical section?
- **Fairness**
  - Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation)
- **Performance**
  - The time overheads added by using the lock

CONTROLLING INTERRUPTS

- **Disable Interrupts** for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems.

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

LOCKS

- Test and Set
- Compare and Swap
- Load-Linked and Store-Conditional
- Fetch-And-Add
- Ticket Lock
- Futex Lock

TEST AND SET (ATOMIC EXCHANGE)

- An instruction to support the creation of simple locks
- **return**(testing) old value pointed to by the ptr.
- Simultaneously **update**(setting) said value to new.
- This sequence of operations is performed atomically.

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

A SIMPLE SPIN LOCK USING TEST-AND-SET

```
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 (do nothing)
}

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

COMPARE-AND-SWAP

- Test whether the value at the address(ptr) is equal to expected.
  - If so, update the memory location pointed to by ptr with the new value.
  - In either case, return the actual value at that memory location.

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

COMPARE-AND-SWAP

- Compare-and-Swap hardware atomic instruction (C-style)

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

- Spin lock with compare-and-swap

LOAD-LINKED AND STORE-CONDITIONAL

```
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
    }
}
```

The store-conditional only succeeds if no intermittent store to the address has taken place.

- **success**: return 1 and update the value at ptr to value.
- **fail**: the value at ptr is not updated and 0 is returned.

FETCH-AND-ADD

- Atomically increment a value while returning the old value at a particular address.

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

- Fetch-And-Add Hardware atomic instruction (C-style)

TICKET LOCK

- **Ticket lock** can be built with fetch-and add.
  - Ensure progress for all threads. à fairness

```
typedef struct __lock_t {
        int ticket;
        int turn;
} lock_t;

void lock_init(lock_t *lock) {
        lock->ticket = 0;
        lock->turn   = 0;
}

void lock(lock_t *lock) {
        int myturn = FetchAndAdd(&lock->ticket);
        while (lock->turn != myturn)
                ; // spin
 }

void unlock(lock_t *lock) {
        lock->turn = lock->turn + 1;
}
```

USING QUEUES: SLEEPING INSTEAD OF SPINNING

- **Queue** to keep track of which threads are waiting to enter the lock.
- park()
  - Put a calling thread to sleep
- unpark(threadID)
  - Wake a particular thread as designated by threadID.

WAKEUP/WAITING RACE

- In case of releasing the lock (*thread A*) just before the call to park(
- (*thread B*) à Thread B would sleep forever (potentially).
- **Solaris** solves this problem by adding a third system call: setpark().
  - By calling this routine, a thread can indicate it *is about to* park.
    - If it happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping.

FUTEX

- Linux provides a futex (is similar to Solaris's parkand unpark). More functionality goes into the kernel.
  - futex_wait(address, expected)
    - Put the calling thread to sleep
    - If the value at address is not equal to expected, the call returns immediately.
  - futex_wake(address)
    - ¢ Wake one thread that is waiting on the queue.

TWO-PHASE LOCKS

- A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.
  - **First phase**
    - The lock spins for a while, *hoping that* it can acquire the lock.
      - If the lock is not acquired during the first spin phase, a second phase is entered,
  - **Second phase**
    - The caller is put to sleep.
    - The caller is only woken up when the lock becomes free later.

LOCK-BASED CONCURRENT DATA STRUCTURES

- Adding locks to a data structure makes the structure **thread safe**.
  - How locks are added determine both the correctness and performance of the data structure.

EXAMPLE: CONCURRENT COUNTERS WITHOUT LOCKS

```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;
}
```

EXAMPLE: CONCURRENT COUNTERS WITH LOCKS

```
typedef struct __counter_t {
    int             value;
    pthread_mutex_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);
}

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;
}
```

SLOPPY COUNTER

- The sloppy counter works by representing …
  - A single **logical counter** via numerous local physical counters, on per CPU core
  - A single **global counter**
  - There are **locks**:
    - One for each local counter and one for the global counter
- Example: on a machine with four CPUs
  - Four local counters
  - One global counter

THE BASIC IDEA OF SLOPPY COUNTING

- When a thread running on a core wishes to increment the counter.
  - It increments its local counter.
  - Each CPU has its local counter:
    - Threads across CPUs can update local counters *without contention*.
    - Thus counter updates are scalable.
  - The local values are periodically transferred to the global counter.
    - Acquire the global lock
    - Increment it by the local counter's value
    - The local counter is then reset to zero.

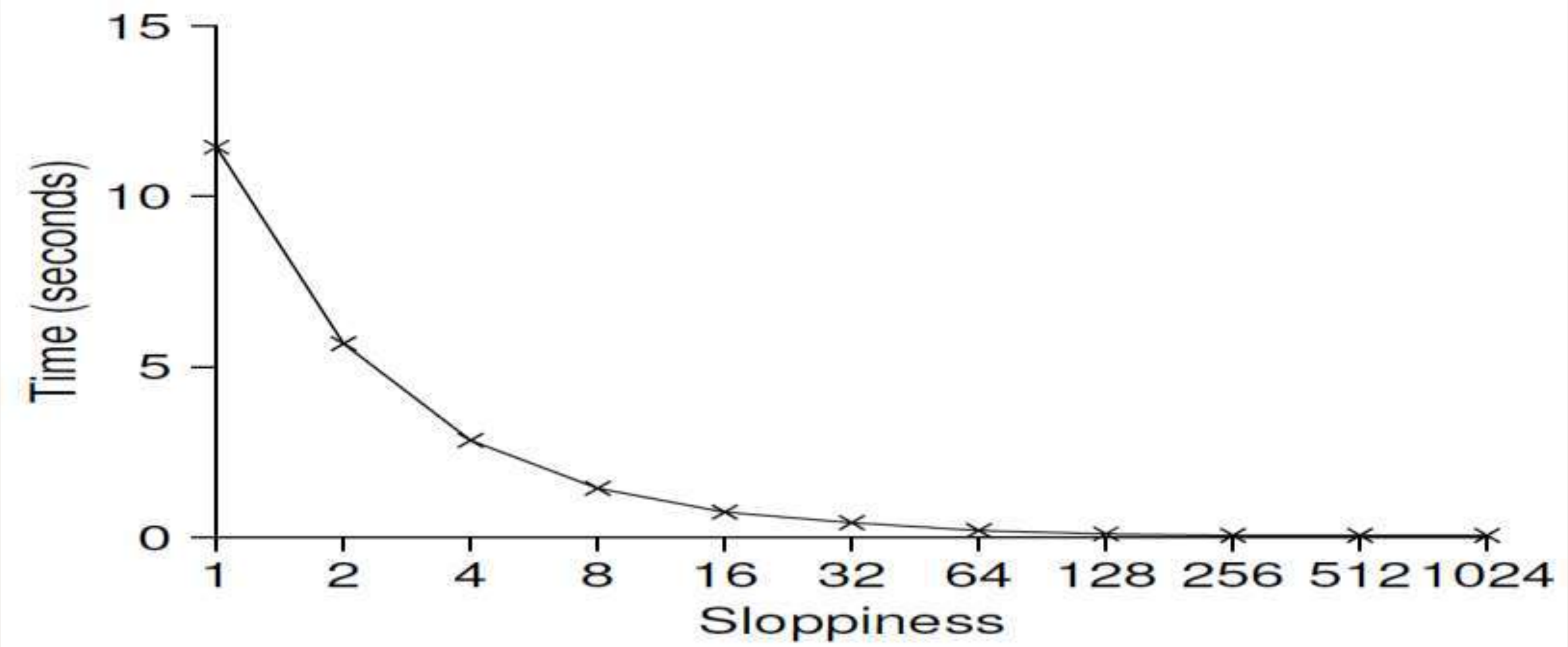
SLOPPY COUNTER EXAMPLE

- **Tracing the Sloppy Counters**
  - The threshold S is set to 5.
  - There are threads on each of four CPUs
  - Each thread updates their local counters $L_1 \ldots L_4$.

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	1	3	4	5 (from L_1)
7	0	2	4	5	10 (from L_4)

IMPORTANCE OF THE THRESHOLD VALUE *S*

- Each four threads increments a counter 1 million times on four CPUs.
  - Low S, Performance is **poor**, The global count is always quite **accurate**.
  - High S, Performance is **excellent**, The global count **lags**.



CONCURRENT LINKED LIST:

```
// 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);
}

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
}

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 LISTS (CONT.)

- The code **acquires** a lock in the insert routine upon entry.
- The code **releases** the lock upon exit.
  - If malloc() happens to *fail*, the code must also release the lock before failing the insert.
  - This kind of exceptional control flow has been shown to be quite error prone.
  - **Solution:** The lock and release *only surround* the actual critical section in the insert code

CONCURRENT LINKED LIST: REFACTORED 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);
}

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
}
```

SCALING LINKED LIST

Hand-over-hand locking (lock coupling)

Add **a lock per node** of the list instead of having a single lock for the entire list.

- When traversing the list,
  - First grab the next node's lock.
  - And then releases the current node's lock.
- Enable a high degree of concurrency in list operations.
  - However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal can be *prohibitive*.
  - Perhaps a hybrid (where you grab a new lock every so many nodes) approach?

MICHAEL AND SCOTT CONCURRENT QUEUES

- Queues uses enqueue / dequeue operations only
- There are two locks.
  - One for the **head** of the queue.
  - One for the **tail**.
  - The goal of these two locks is to enable concurrency of enqueue and
    - dequeue operations.
- Add a dummy node
  - Allocated in the queue initialization code
  - Enable the separation of head and tail operations

CONCURRENT QUEUES (CONT.)

```
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
             pthread_mutex_t          headLock;
                                      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 QUEUES (CONT.)

(Cont.)

```
void   Queue_Enqueue(queue_t *q, int value) {
           node_t *tmp = malloc(sizeof(node_t));
           assert(tmp != NULL && "Malloc failed");

           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 QUEUES (CONT.)

(Cont.)

```
int   Queue_Dequeue(queue_t *q, int *value) {
        pthread_mutex_lock(&q->headLock);
        node_t *tmp = q->head;
        node_t *newHead = tmp->next;
        if (newHead == NULL) {
                pthread_mutex_unlock(&q->headLock);  return -1;
                // queue was empty
        }
        *value = newHead->value;
        q->head = newHead;
        pthread_mutex_unlock(&q->headLock);
        free(tmp);
        return 0;
}
```

CONCURRENT HASH TABLE

- Focus on a simple hash table
  - The hash table does not resize, simplest hash, slow searches
  - Built using the concurrent lists
  - It uses a lock per hash bucket each of which is represented by a list.

KL UNIVERSITY

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);
}
```

CONDITION VARIABLES

- There are many cases where a thread wishes to <u>check</u> whether a
- **condition** is true before continuing its execution.
- Example: A parent thread might wish to check whether a child thread has *completed*.
    - This is often called a join().

HOW TO WAIT FOR A CONDITION

- Condition variable
  - **Waiting** on the condition
    - An explicit queue that threads can put themselves on when some state of execution is not as desired.
  - **Signaling** on the condition
    - Some other thread, *when it changes said state*, can wake one of those waiting threads and allow them to continue.

DEFINITION AND ROUTINES

- Declare condition variable

```
pthread cond t c;
```

- Proper initialization is required.

Operation (the POSIX calls)

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

DEFINITION AND ROUTINES

- The wait() call takes a mutex as a parameter.
  - ¢ The wait() call **release the lock** and put the calling thread to sleep.
  - ¢ When the thread wakes up, it must **re-acquire the lock**.

PARENT WAITING FOR THE CHILD

```
int done = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void   thr_exit() {   Pthread_mutex_lock(&m);
           done = 1;
           Pthread_cond_signal(&c);
           Pthread_mutex_unlock(&m);

}

void   *child(void *arg) {
           printf("child\n");
           thr_exit();
           return NULL;
}

void   thr_join() {   Pthread_mutex_lock(&m);
           while (done == 0)
                    Pthread_cond_wait(&c,
           Pthread_mutex_unlock(&m);                       &m);

}
```

PARENT WAITING FOR CHILD: USE A CONDITION VARIABLE

(cont.)

```
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

PARENT WAITING FOR CHILD: USE A CONDITION VARIABLE

- **Parent:**
  - Create the child thread and continue running itself.
  - Call into thr_join() to wait for the child thread to complete.
    - ¢ Acquire the lock
    - ¢ Check if the child is done
    - ¢ Put itself to sleep by calling wait()
    - ¢ Release the lock

PARENT WAITING FOR CHILD: USE A CONDITION VARIABLE

- **Child:**
  - Print the message “child”
  - Call thr_exit() to wake the parent thread
    - ¢ Grab the lock
    - ¢ Set the state variable done
    - ¢ Signal the parent thus waking it.

THE PRODUCER / CONSUMER (BOUND BUFFER) PROBLEM

- **Producer**
  - Produce data items
  - Wish to place data items in a buffer
- **Consumer**
  - Grab data items out of the buffer consume them in some way
- Example: Multi-threaded web server
  - A producer puts HTTP requests in to a work queue
  - Consumer threads take requests out of this queue and process them

BOUNDED BUFFER

- A bounded buffer is used when you pipe the output of one program into another.
  - Example: grep foo file.txt | wc –l
    - ¢ The grepprocess is the producer.
    - ¢ The wc process is the consumer.
    - ¢ Between them is an in-kernel bounded buffer.
  - Bounded buffer is a Shared resource à **Synchronized access** is required.

THE SINGLE BUFFER PRODUCER/CONSUMER SOLUTION

- Use two condition variables and while
  - **Producer** threads wait on the condition empty, and signals fill.
  - **Consumer** threads wait on fill and signal empty.

```
cond_t empty, fill;
mutex_t mutex;

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

        }
}
```

THE SINGLE BUFFER PRODUCER/CONSUMER SOLUTION

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

THE FINAL PRODUCER/CONSUMER SOLUTION(MULTIPLE BUFFERS)

- More **concurrency** and **efficiency** à Add more buffer slots.
  - Allow concurrent production or consuming to take place.
  - Reduce context switches.

```
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;
}
```

The Final Put and Get Routines

THE FINAL PRODUCER/CONSUMER SOLUTION (CONT.)

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i,loops = (int) arg;
    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(&fill);                          // p5
                Pthread_mutex_unlock(&mutex);                // p6
    }
}

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

THE FINAL PRODUCER/CONSUMER SOLUTION (CONT.)

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

The Final Working Solution (Cont.)

- p2: **A producer** only sleeps if all buffers are currently filled.
- c2: **A consumer** only sleeps if all buffers are currently empty.
- **Note**: The Producer Consumer Problem **with Multiple Buffer Slots** is practically demonstrated in the next Chapter using semaphores and mutex.

MUTEX LOCK

- A lock variable provides the simplest synchronization mechanism for processes.
  1. Its a **software mechanism** implemented in user mode, i.e. no support required from the Operating System.
  2. Its a busy waiting solution (keeps the CPU busy even when its technically waiting).
  3. It can be used for more than two processes.
  4. When Lock = 0 implies critical section is vacant (initial value) and Lock = 1 implies critical section occupied.

- Basic operations on a **lock:**
- **acquire:** mark the lock as owned by the current thread; if some other thread already owns the lock then first wait until the lock is free.
- **release:** mark the lock as free (it must currently be owned by the calling thread).

```
do {
    acquire lock
      critical section
    release lock
      remainder section
  } while
```

Mutex Locks

- **mutex** lock (**mut**ual **ex**clusion)

lock

Process 1

```
while (true){
   acquire(&lock);
   /*critical section */
   release(&lock);
   /*  remainder section */
  }
```

Process 2

```
while (true){
   acquire(&lock);
   /*critical section */
   release(&lock);
   /*  remainder section */
  }
```

lock

Process 1

```
while (true){

   while (lock != 0);
   lock = 1;

   /*critical section */

   lock = 0;

   /*  remainder section */
  }
```

Process 2

```
while (true){

   while (lock != 0);
   lock = 1;

   /*critical section */

   lock = 0;

   /*  remainder section */
  }
```

THANK YOU

Team – Operating System