

Concurrency and Synchronisation

Learning Outcomes

- Understand concurrency is an issue in operating systems and multithreaded applications
- Know the concept of a *critical region*.
- Understand how mutual exclusion of critical regions can be used to solve concurrency issues
 - Including how mutual exclusion can be implemented correctly and efficiently.
- Be able to identify a *producer consumer bounded buffer* problem.

Textbook

- Sections 2.3 - 2.3.7 & 2.5

Concurrency Example

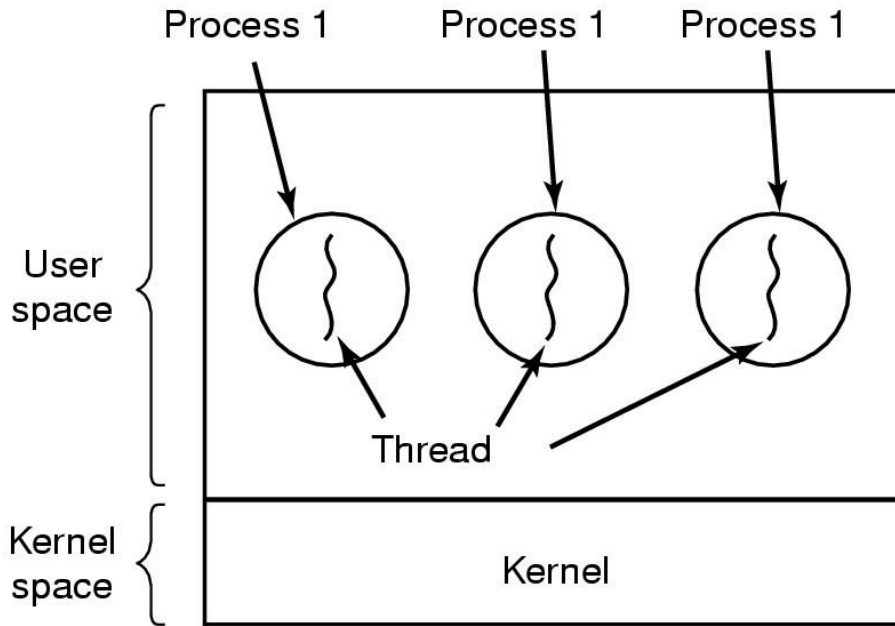
count is a global variable shared between two threads, t is a local variable.
After increment and decrement complete, what is the value of count?

```
void increment ()  
{  
    int t;  
    t = count;  
    t = t + 1;  
    count = t;  
}
```

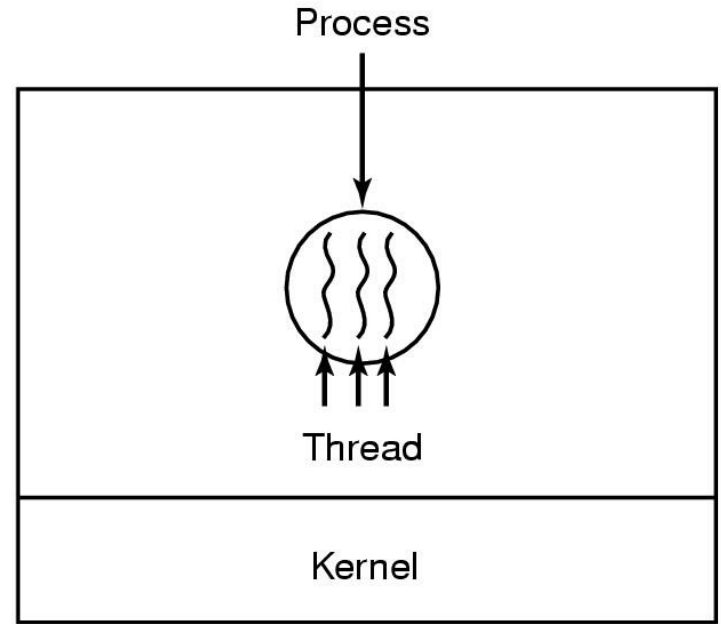
```
void decrement ()  
{  
    int t;  
    t = count;  
    t = t - 1;  
    count = t;  
}
```

We have a
race
condition

Where is the concurrency?



(a)

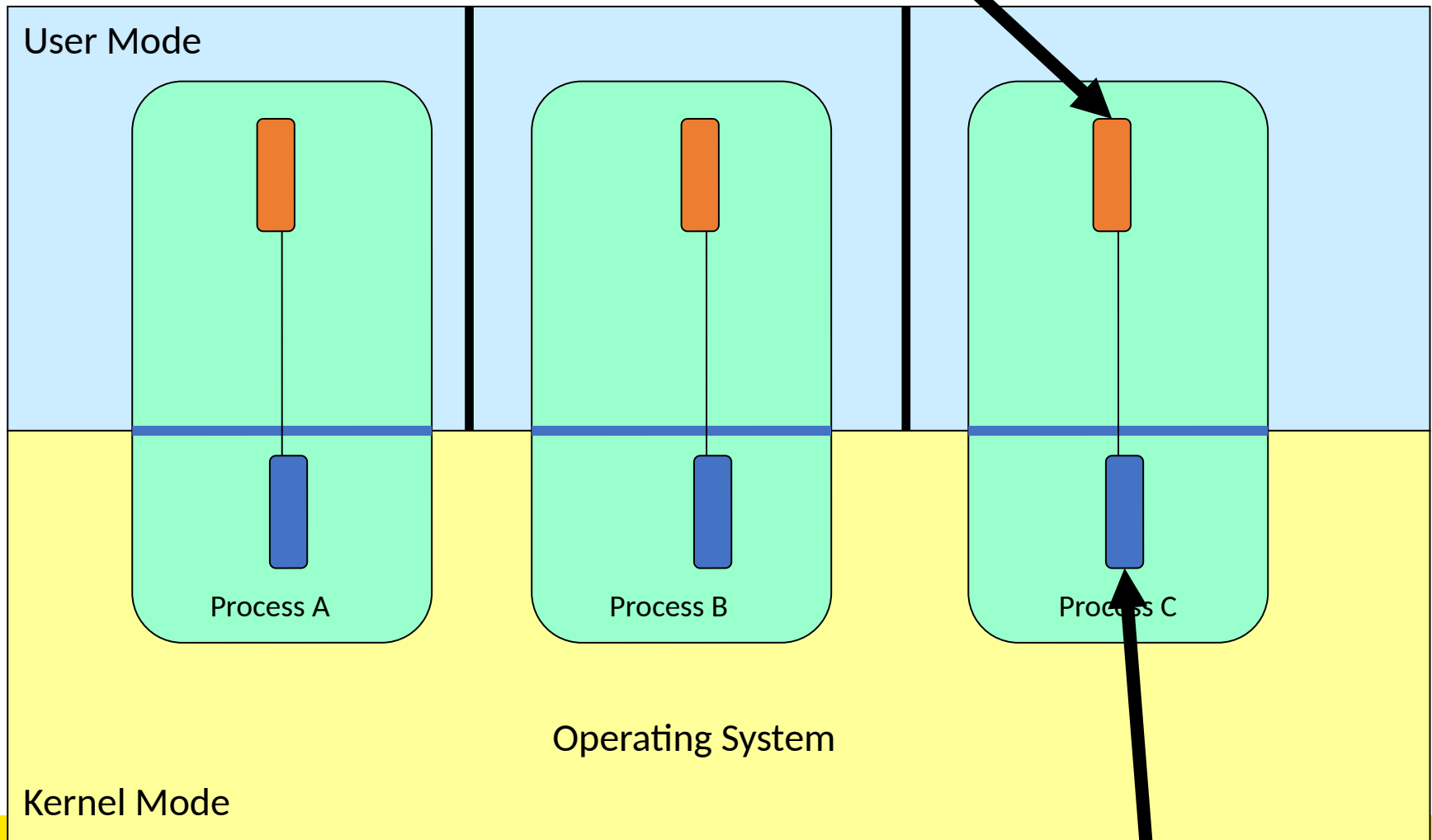


(b)

- (a) Three processes each with one thread
- (b) One process with three threads

There is in-kernel concurrency even for single-threaded processes

Process's user-level stack and execution state



Process's in-kernel stack and execution state

Critical Region

- We can control access to the shared resource by controlling access to the code that accesses the resource.
⇒ *A critical region* is a region of code where shared resources are accessed.
 - Variables, memory, files, etc...
- Uncoordinated entry to the critical region results in a race condition
⇒ Incorrect behaviour, deadlock, lost work,...

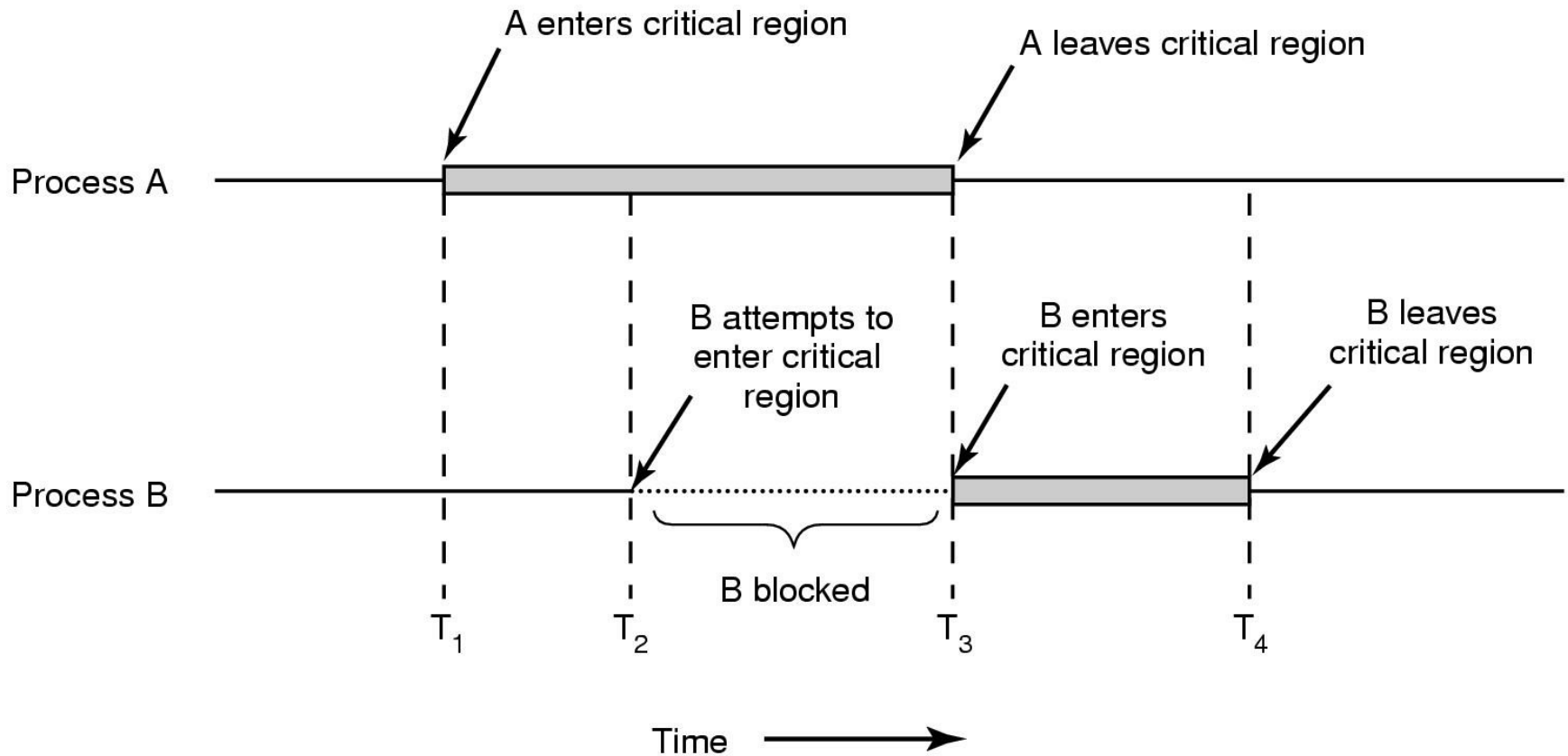
Identifying critical regions

- Critical regions are regions of code that:
 - Access a shared resource,
 - and correctness relies on the shared resource not being concurrently modified by another thread/process/entity.

```
void increment ()
{
    int t;
    t = count;
    t = t + 1;
    count = t;
}
```

```
void decrement ()
{
    int t;
    t = count;
    t = t - 1;
    count = t;
}
```

Accessing Critical Regions



Mutual exclusion using critical regions

Example critical regions

```
struct node {  
    int data;  
    struct node *next;  
};  
struct node *head;
```

```
void init(void)  
{  
    head = NULL;  
}
```

- Simple last-in-first-out queue implemented as a linked list.

```
void insert(struct *item)  
{  
    item->next = head;  
    head = item;  
}
```

```
struct node *remove(void)  
{  
    struct node *t;  
    t = head;  
    if (t != NULL) {  
        head = head->next;  
    }  
    return t;  
}
```

Example Race

```
void insert(struct *item)
{
    item->next = head;
    head = item;
}
```

```
void insert(struct *item)
{
    item->next = head;
    head = item;
}
```

Example critical regions

```
struct node {  
    int data;  
    struct node *next;  
};  
struct node *head;
```

```
void init(void)  
{  
    head = NULL;  
}
```

- Critical sections

```
void insert(struct *item)  
{  
    item->next = head;  
    head = item;  
}
```

```
struct node *remove(void)  
{  
    struct node *t;  
    t = head;  
    if (t != NULL) {  
        head = head->next;  
    }  
    return t;  
}
```

Critical Regions Solutions

- We seek a solution to coordinate access to critical regions.
 - Also called critical sections
- Conditions required of any solution to the critical region problem
 1. Mutual Exclusion:
 - No two processes simultaneously in critical region
 2. Progress
 - No process running outside its critical region may block another process
 3. Non-Starvation
 - No process waits forever to enter its critical region
 4. Generality
 - A solution that only works for 2 CPUs isn't a good solution.

A solution?

- A lock variable
 - If lock == 1,
 - somebody is in the critical section and we must wait
 - If lock == 0,
 - nobody is in the critical section and we are free to enter


A solution?

```
while(TRUE) {  
    while(lock == 1)  
        ;  
    lock = 1;  
    critical();  
    lock = 0  
    non_critical();  
}
```

```
while(TRUE) {  
    while(lock == 1)  
        ;  
    lock = 1;  
    critical();  
    lock = 0  
    non_critical();  
}
```

A problematic execution sequence

```
while(TRUE) {  
  
    while(lock == 1)  
        ;  
    lock = 1;  
  
    critical();  
    lock = 0  
    non_critical();  
}  
  
while(TRUE) {  
    while(lock == 1)  
        ;  
  
    lock = 1;  
    critical();  
  
    lock = 0  
    non_critical();  
}
```



Observation

- Can we test for these problematic cases?
- Unfortunately, it is usually easier to show something does not work, than it is to prove that it does work.
 - Easier to provide a counter example
 - Ideally, we'd like to prove, or at least informally demonstrate, that our solutions work.
- Some of our problematic sequences are quite unlikely
 - e.g. Timer interrupt arrives exactly after we read the lock variable.
 - Testing for concurrency errors is really tricky.

Mutual Exclusion by Taking Turns

```
while (TRUE) {  
    while (turn != 0)  
        /* loop */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(Process 0.)

```
while (TRUE) {  
    while (turn != 1)  
        /* loop */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(Process 1.)

Proposed solution to critical region problem

Mutual Exclusion by Taking Turns

- Works due to *strict alternation*
 - Each process takes turns
- Cons
 - This is not a good approach
 - Busy waiting
 - Process may wait to enter the critical section while the other process is doing non-critical work.
 - With many processes, this is even worse.
- Enforcing a fixed schedule in a concurrent environment doesn't work well.

Mutual Exclusion by Disabling Interrupts

- Before entering a critical region, disable interrupts
- After leaving the critical region, enable interrupts

```
while(TRUE) {  
    disable_interrupts();  
    critical();  
    enable_interrupts();  
    non_critical();  
}
```

```
while(TRUE) {  
    disable_interrupts();  
    critical();  
    enable_interrupts();  
    non_critical();  
}
```

Mutual Exclusion by Disabling Interrupts

- Pros
 - simple
- Cons
 - Only available in the kernel
 - Delays everybody else, even with no contention
 - Slows interrupt response time
 - Does not work on a multi-core processor

Core Problem: Read and Update

- We don't want a fixed schedule
- If multiple threads (e.g. A, B, C) can enter the critical section, thread A needs to do two things:
 - Read the current state
 - To know that B/C have not gone into their critical sections.
 - Change the state
 - To inform B/C that A is entering the critical section.
- This can be in memory by “Peterson's algorithm”
 - See wikipedia for details.
 - Doesn't work on modern hardware.

Hardware Support for mutual exclusion

- Test and set instruction
 - Test memory cell X and set memory cell X
 - Can be used to implement lock variables correctly
 - It loads the value of the lock
 - If lock == 0,
 - set the lock to 1
 - return the result 0 - we acquire the lock
 - If lock == 1
 - return 1 - another thread/process has the lock
 - Hardware guarantees that the instruction executes atomically.
 - Both the read-lock part and the set-lock part.
 - Atomically: As an indivisible unit.

Mutual Exclusion via Test-and-Set

enter_region:

```
TSL r0, lock      | copy lock to r0, 1 to lock
CMP r0, #0        | compute whether lock was 0
JNE enter_region  | if not, loop again
RET               | done, return to caller
```

leave_region:

```
MOVE lock, #0     | store 0 (unlocked) in lock
```

Entering and leaving a critical region using the TSL instruction

Simple Test-and-Set Lock

- Pros

- Simple (easy to see it is correct)
- Available at user-level
 - To any number of processors
 - To implement any number of lock variables

- Cons

- Busy waits (also termed a *spin lock*)
 - Consumes CPU
- Starvation might be possible when a process leaves its critical section and more than one process is waiting.

Variants of Test-and-Set

Other synchronisation operations are provided by modern processors

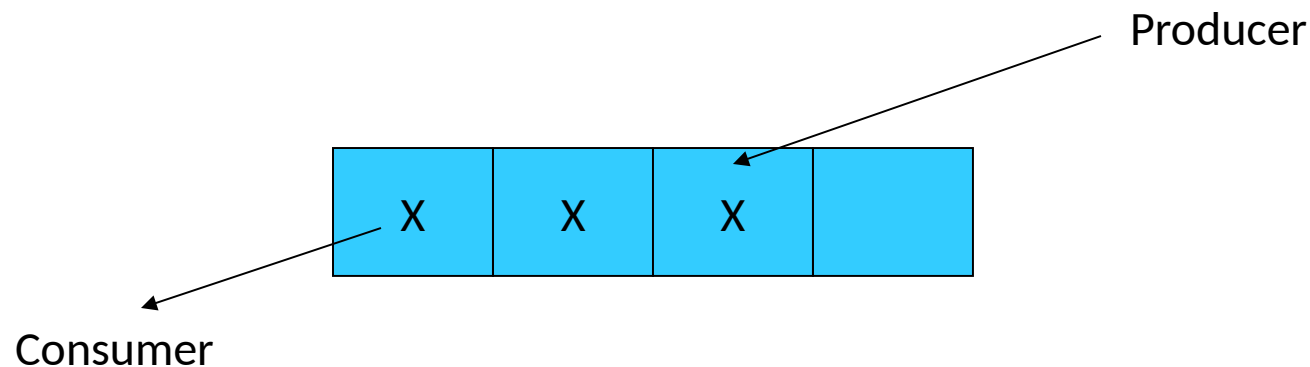
- Compare-and-Swap
 - Check the contents of X is Y, and if so write Z
- Load-Link, Store-Exclusive
 - The store fails if the linked memory address has been accessed
- Atomic Arithmetic
 - e.g. Atomic Increment by 1.

Tackling the Busy-Wait Problem

- Sleep / Wakeup
 - If a user-level thread cannot take the lock, instead of spinning, it can make a “sleep” system call to wait without wasting CPU.
 - When a thread releases the lock, it may need to call the “wakeup” system call to unblock other processes.
 - Accidentally calling “wakeup” when other threads are not sleeping has no effect.

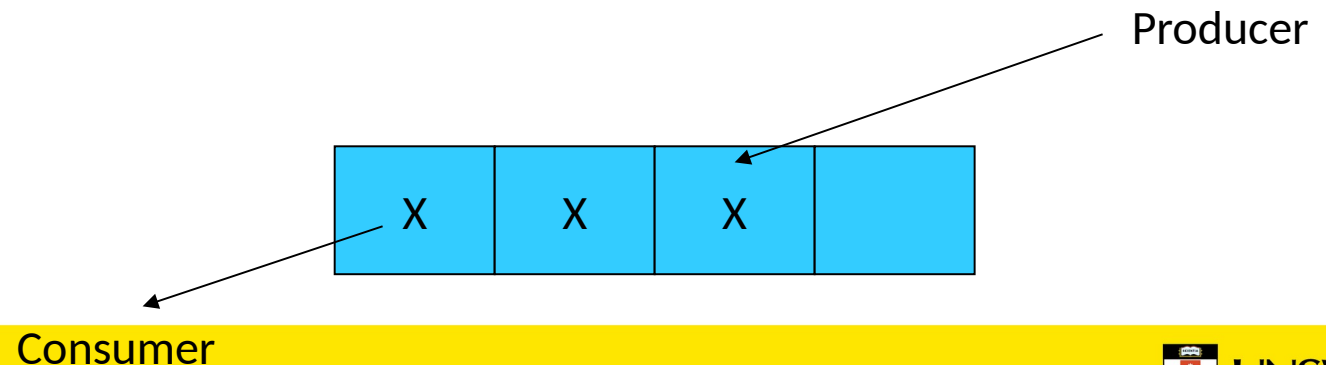
The Producer-Consumer Problem

- Also called the *bounded buffer* problem
- A producer produces data items and stores the items in a buffer
- A consumer takes the items out of the buffer and consumes them.



Issues

- We must keep an accurate count of items in buffer
 - Producer
 - should sleep when the buffer is full,
 - and wakeup when there is empty space in the buffer
 - The consumer can call wakeup when it consumes the first entry of the full buffer
 - Consumer
 - should sleep when the buffer is empty
 - and wake up when there are items available
 - Producer can call wakeup when it adds the first item to the buffer



Pseudo-code for producer and consumer

```
int count = 0;
#define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce()
        if (count == N)
            sleep(prod);
        insert_item();
        count++;
        if (count == 1)
            wakeup(con);
    }
}
```

```
con() {
    while(TRUE) {
        if (count == 0)
            sleep(con);
        remove_item();
        count--;
        if (count == N-1)
            wakeup(prod);
    }
}
```

Problems

```
int count = 0;
#define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce()
        if (count == N)
            sleep(prod);
        insert_item();
        count++;
        if (count == 1)
            wakeup(con);
    }
}
```

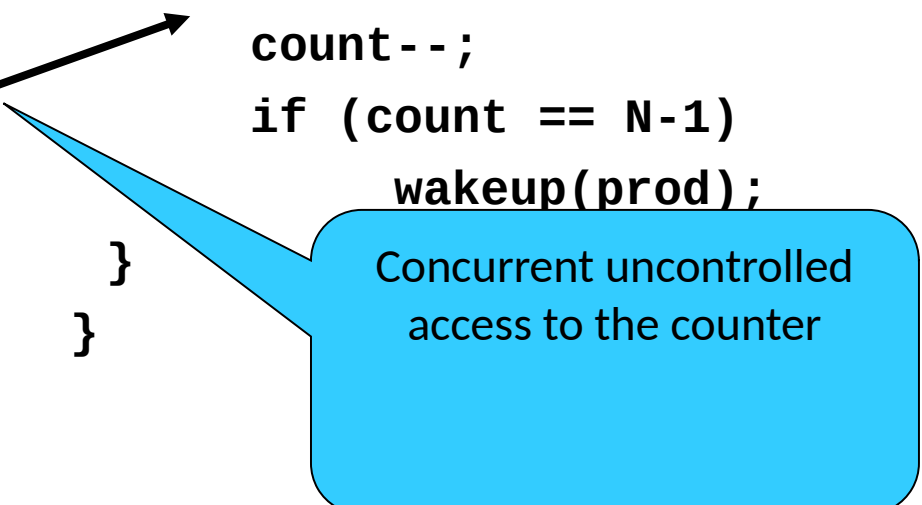
```
con() {
    while(TRUE) {
        if (count == 0)
            sleep(con);
        remove_item();
        count--;
        if (count == N-1)
            sleep(prod);
    }
}
```

Concurrent uncontrolled access to the buffer

Problems

```
int count = 0;
#define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce()
        if (count == N)
            sleep(prod);
        insert_item();
        count++;
        if (count == 1)
            wakeup(con);
    }
}
```

```
con() {
    while(TRUE) {
        if (count == 0)
            sleep(con);
        remove_item();
        count--;
        if (count == N-1)
            wakeup(prod);
    }
}
```



Concurrent uncontrolled access to the counter

Proposed Solution

- Lets use a locking primitive based on test-and-set to protect the concurrent access

Proposed solution?

```
int count = 0;
lock_t buf_lock;
#define N 4 /* buf size */
prod() {
    while(TRUE) {
        item = produce()
        if (count == N)
            sleep(prod);
        acquire_lock(buf_lock)
        insert_item();
        count++;
        release_lock(buf_lock)
        if (count == 1)
            wakeup(con);
    }
}
```

```
con() {
    while(TRUE) {
        if (count == 0)
            sleep(con);
        acquire_lock(buf_lock)
        remove_item();
        count--;
        release_lock(buf_lock);
        if (count == N-1)
            wakeup(prod);
    }
}
```

Problematic execution sequence

```
prod() {  
  while(TRUE) {  
    item = produce()  
    if (count == N)  
      sleep(prod);  
    acquire_lock(buf_lock)  
    insert_item();  
    count++;  
    release_lock(buf_lock)  
    if (count == 1)  
      wakeup(con);  
  }  
}
```

```
con() {  
  while(TRUE) {  
    if (count == 0)
```

wakeup without a
matching sleep is lost

```
    sleep(con);  
    acquire_lock(buf_lock)  
    remove_item();  
    count--;  
    release_lock(buf_lock);  
    if (count == N-1)  
      wakeup(prod);  
  }  
}
```

Problem

- The test for *some condition* and actually going to sleep needs to be atomic
- The following does not work:

```
acquire_lock(buf_lock)
if (count == N)
    sleep();
release_lock(buf_lock)
```

The lock is held while asleep
⇒ count will never change

```
acquire_lock(buf_lock)
if (count == 1)
    wakeup();
release_lock(buf_lock)
```

Today

- Concurrency.
- Critical sections and mutual exclusion.
- Test-and-set operations and locks.
- The producer/consumer problem.
 - More on that later, when we return to concurrency management.