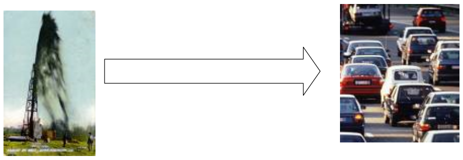
Synchronization

COS 450 - Fall 2018

1

Producer - Consumer

Remember the **Producer** and **Consumer** scenario...



...it had a hidden **problem**

2

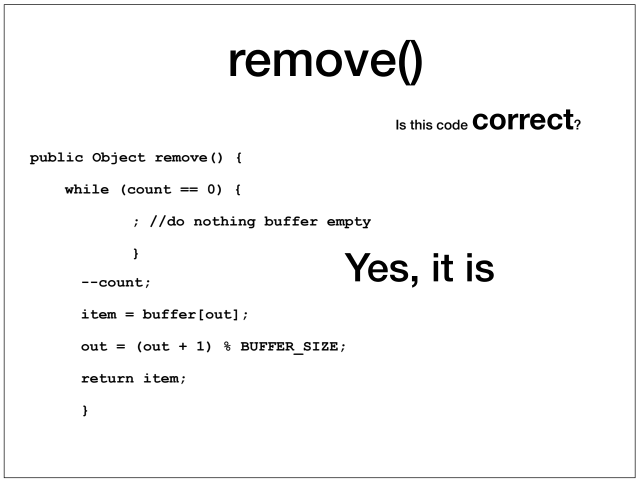
insert()

Is this code **correct**?

```
public void insert(Object item) {
    while (count == BUFFER_SIZE) {
            ; //do nothing buffer full
            }
      ++count;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      }
```

Yes, it is

3



remove()

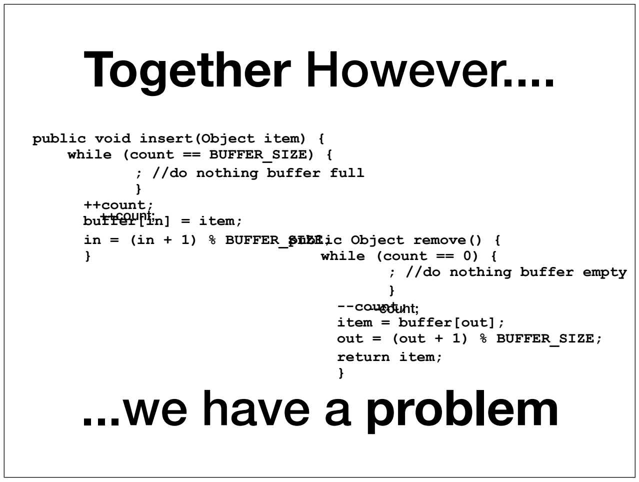
Is this code **correct**?

```
public Object remove() {
    while (count == 0) {
            ; //do nothing buffer empty
            }
      --count;
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      return item;
      }
```

Yes, it is

4

Similar process happens with remove that happened with insert.



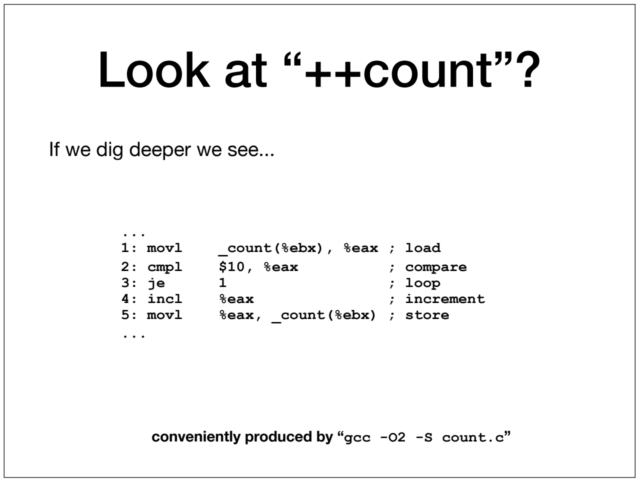
Together However....

```
public void insert(Object item) {
    while (count == BUFFER_SIZE) {
            ; //do nothing buffer full
            }
      ++count;
        ++count;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      }
```

```
public Object remove() {
    while (count == 0) {
            ; //do nothing buffer empty
            }
      --count;
           count;
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      return item;
      }
```

...we have a **problem**

5



Look at “++count”?

If we dig deeper we see...

```
...
1: movl    _count(%ebx), %eax ; load
2: cmpl    $10, %eax          ; compare
3: je      1                  ; loop
4: incl    %eax               ; increment
5: movl    %eax, _count(%ebx) ; store
...
```

conveniently produced by “`gcc -O2 -S count.c`”

6

7

...and “--count”

If we dig deeper we see...

```
...
1: movl    _count(%ebx), %eax  ; load
2: testl   %eax, %eax          ; compare
3: je      1                   ; loop
4: decl    %eax                ; decrement
5: movl    %eax, _count(%ebx)  ; store
...
```

conveniently produced by “`gcc -O2 -S count.c`”

8

When they run **concurrently**

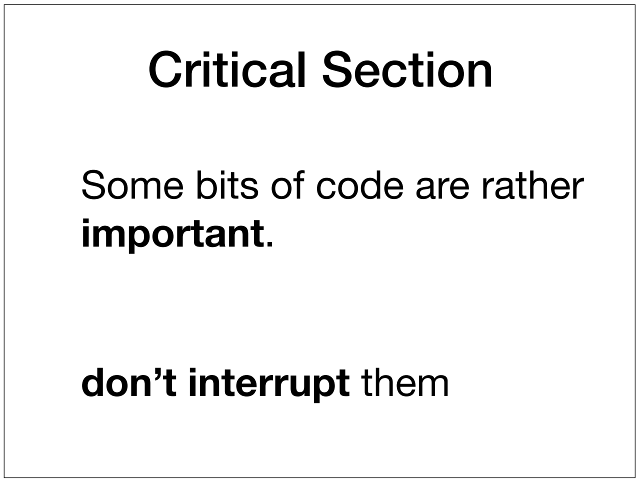
What is count?

we might see something like...

```
A1: movl    _count(%ebx), %eax ; load
A2: cmpl    $10, %eax          ; compare
A3: je      1                  ; loop
A4: incl    %eax               ; increment
B1: movl    _count(%ebx), %eax  ; load
B2: testl   %eax, %eax          ; compare
B3: je      1                   ; loop
B4: decl    %eax                ; decrement
B5: movl    %eax, _count(%ebx)  ; store
A5: movl    %eax, _count(%ebx) ; store
```

9

What is count?

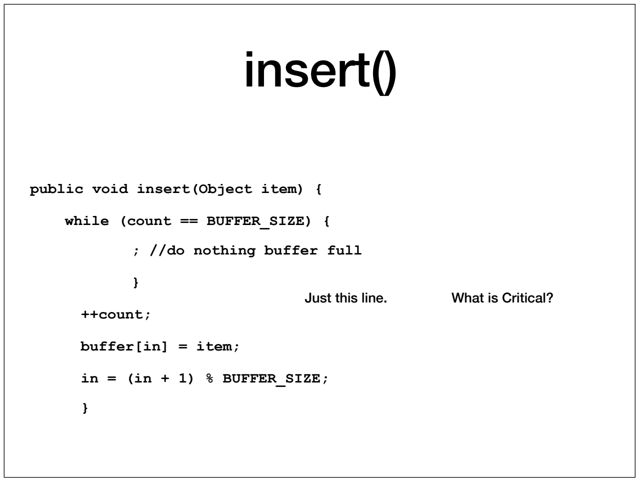


Critical Section

Some bits of code are rather **important**.

don’t interrupt them

10



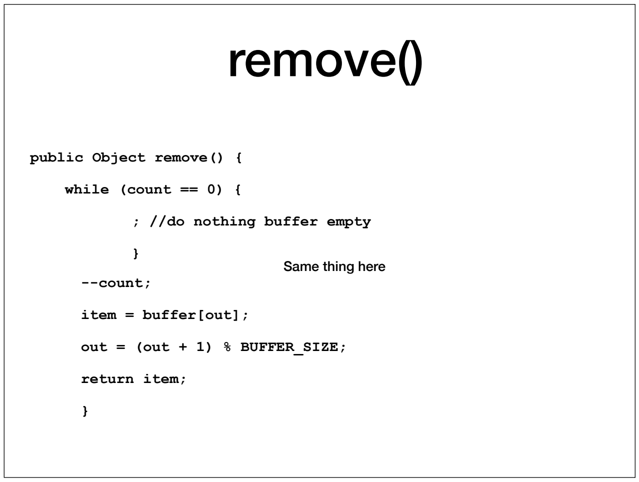
insert()

```
public void insert(Object item) {
    while (count == BUFFER_SIZE) {
            ; //do nothing buffer full
            }
      ++count;
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      }
```

Just this line.

What is Critical?

11



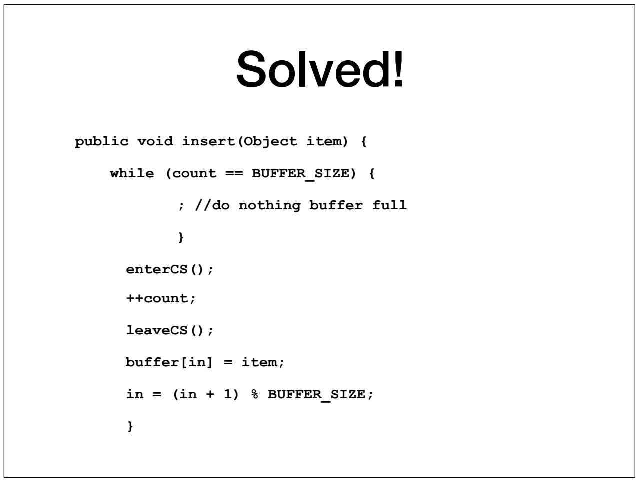
remove()

```
public Object remove() {
    while (count == 0) {
            ; //do nothing buffer empty
            }
      --count;
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      return item;
      }
```

Same thing here

12

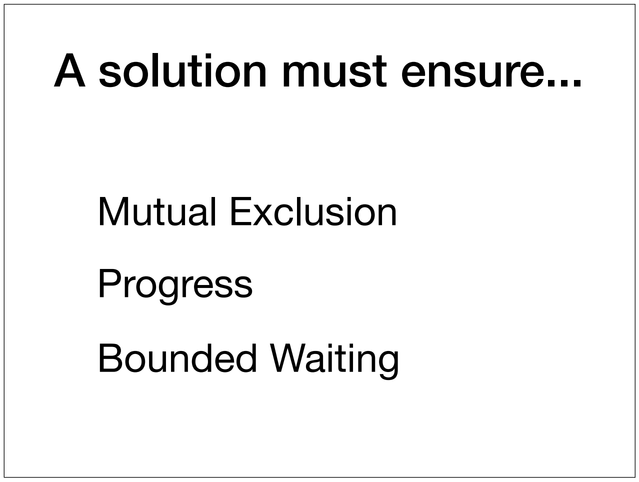
Similar process happens with remove that happened with insert.



Solved!

```
public void insert(Object item) {
    while (count == BUFFER_SIZE) {
           ; //do nothing buffer full
           }
      enterCS();
      ++count;
      leaveCS();
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;
      }
```

13



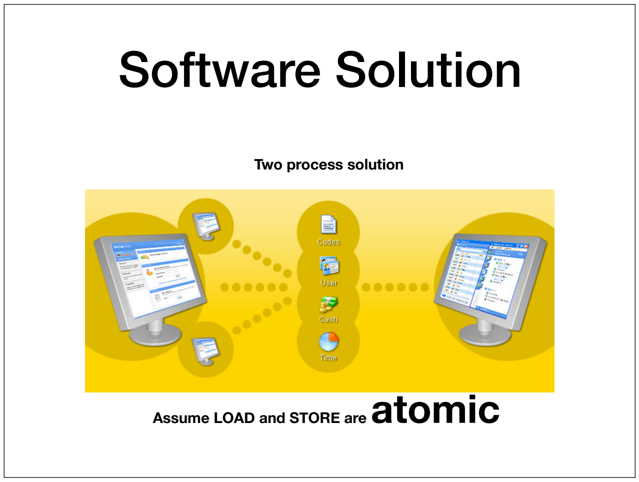
A solution must ensure...

Mutual Exclusion

Progress

Bounded Waiting

14



Software Solution

Two process solution

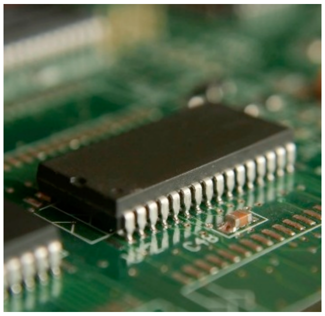
Assume LOAD and STORE are **atomic**

15

Peterson’s Solution in textbook

16

Hardware Solutions



17

Get and Set

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
       Thread.yield();

    criticalSection();
    lock.set(false);
    remainderSection();
}
```

18

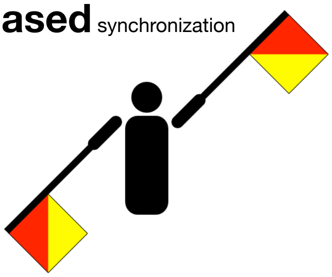
Swap

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
       lock.swap(key);
    }
    while (key.get() == true);

    criticalSection();
    lock.set(false);
    remainderSection();
}
```

19

Semaphores

an **integer based** synchronization mechanism



20

Operations

Semaphores have **two operations** defined on them...

acquire

release

21

Semaphore Use

```
Semaphore S = new Semaphore()

S.acquire();

   // critical section

S.release();
```

...this is a simple **mutex lock** or **binary semaphore**

22

Multiple Resources

```
Semaphore S = new Semaphore(10)

S.acquire();

   // critical section

S.release();
```

...here we can enter the **critical section multiple times**

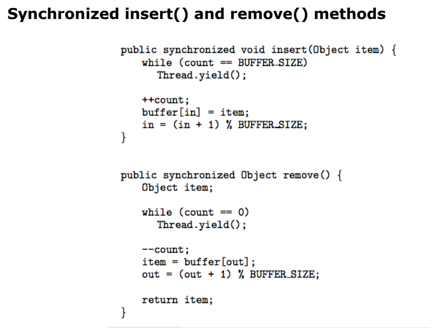
23

Monitors

language based mutex

...in Java, “**synchronized**” keyword

24



Synchronized insert() and remove() methods

```
public synchronized void insert(Object item) {
   while (count == BUFFER_SIZE)
     Thread.yield();

   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}


public synchronized Object remove() {
   Object item;

   while (count == 0)
     Thread.yield();

   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   return item;
}
```

Implementation Details

busy waiting (spinlock)
while (canEnter()) { }

wait and notify
while(canEnter()) { wait(); }

25

When a thread invokes **wait():**

1. The thread releases the object lock;
2. The state of the thread is set to Blocked;
3. The thread is placed in the **wait set** for the object.

When a thread invokes **notify()**:

1. An arbitrary thread T from the wait set is selected;
2. T is moved from the wait to the entry set;
3. The state of T is set to Runnable.

26

```
public synchronized void insert(Object item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;

   notify();
}

public synchronized Object remove() {
   Object item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException e) { }
   }

   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   notify();

   return item;
}
```

27

Classic Synchronization Problems

28

Bounded Buffer

Readers-Writers

Dining Philosophers

Bounded Buffer

29

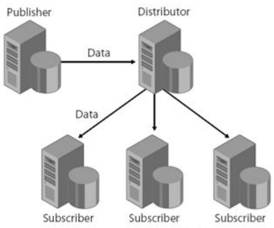
Multiple processes share a **common memory** buffer.

Readers-Writers

30

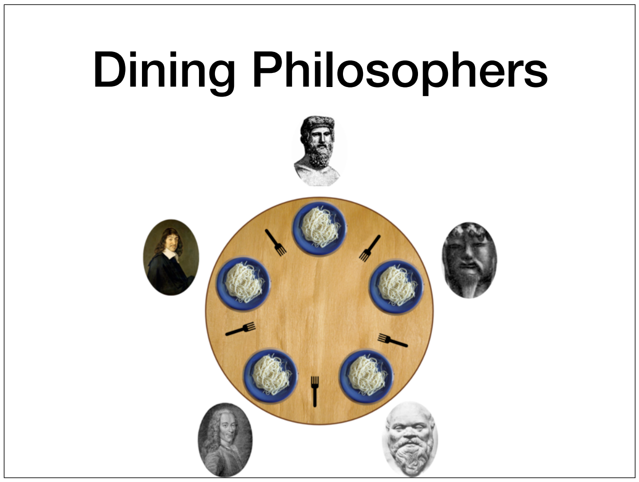
Many can read

Only one can write



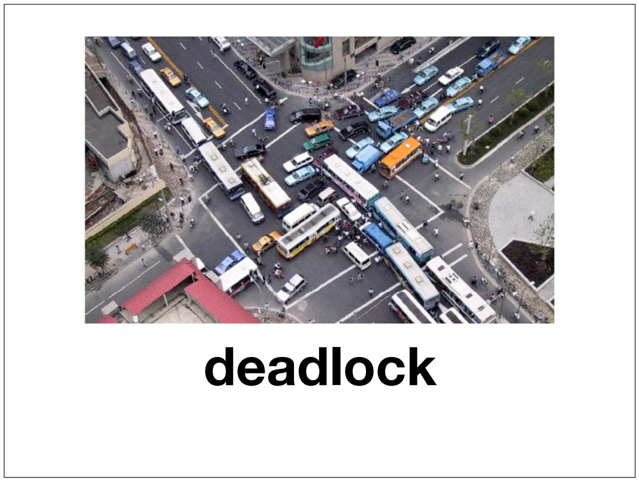
Server roles in replication

31



Dining Philosophers

32



deadlock

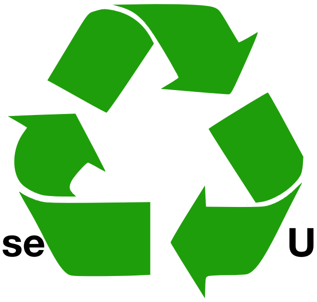
33

the problem is...

processes **compete** for resources

34

how a process uses a resource...



Request

Release

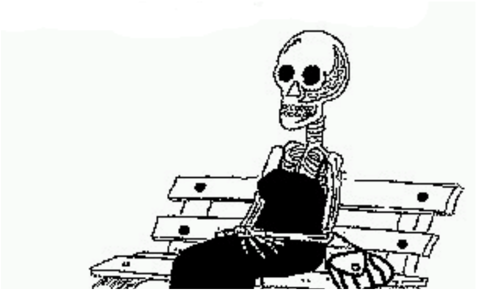
Use

35

deadlock can only **exist** if...

36

Mutual Exclusion



Hold and Wait

37

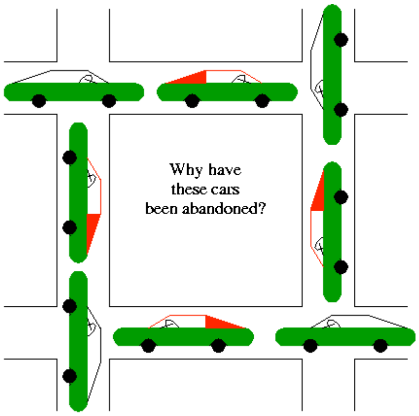
No Preemption

38

Circular Wait

39

40



41

How to **handle** deadlock

Prevent - 4 conditions

Avoid - safe states

Detect - after the fact

Ignore - it's the administrator's problem

42

Prevention

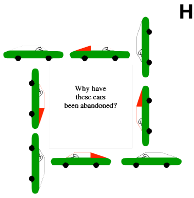
don't let the conditions exist that cause deadlock...

Mutual Exclusion

Hold and Wait

No Preemption

Circular Wait



43

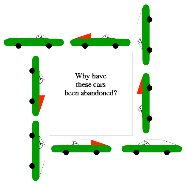
Avoidance

keep from going into a state that may allow deadlock...

Safe States

Unsafe States

...using Banker's, Safety, Resource-Request algorithms.



44

45

Ignore

Deadlock

Mutual Exclusion

Hold and Wait

No Preemption (of resources)

Circular Wait

46

Synchronization

End of Section

47