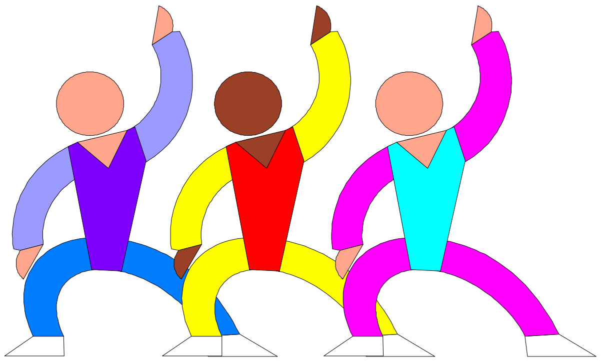
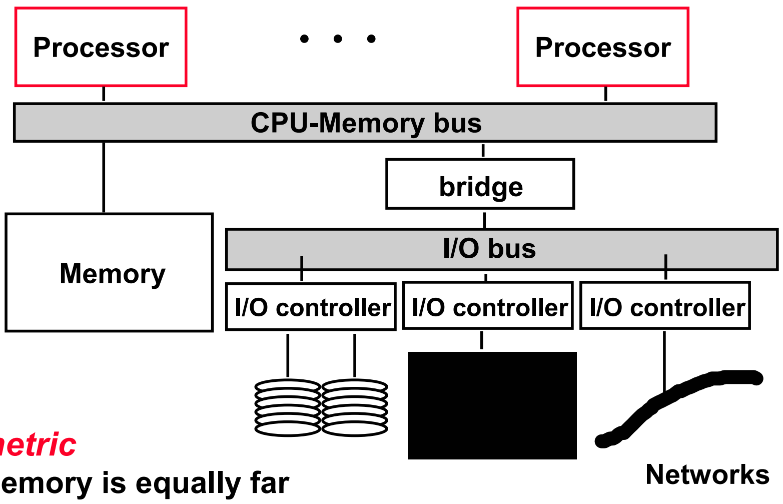
Symmetric Multiprocessors: Synchronization and Sequential Consistency



1

Symmetric Multiprocessors

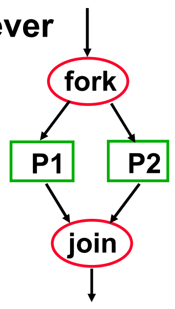


symmetric

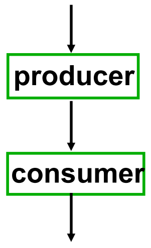
- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

Synchronization

The need for synchronization arises whenever there are parallel processes in a system *(even in a uniprocessor system)*



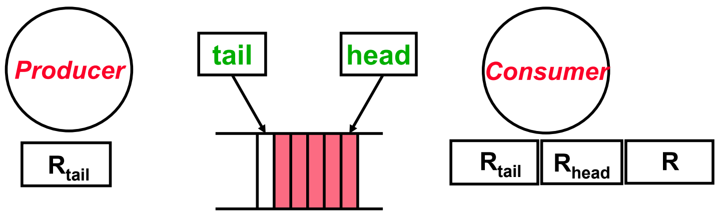
***Forks and Joins*: In parallel programming a parallel process may want to wait until several events have occurred**



***Producer-Consumer*: A consumer process must wait until the producer process has produced data**

***Exclusive use of a resource*: Operating system has to ensure that only one process uses a resource at a given time**

A Producer-Consumer Example



Producer posting Item x:

```
        R_tail ← M[tail]
1 →     M[<R_tail>] ← x
        R_tail ← <R_tail> + 1
2 →     M[tail] ← <R_tail>
```

Consumer:

```
3 ↘     R_head ← M[head]
spin:   R_tail ← M[tail]
        if <R_head> == <R_tail>
4 →     R ← M[<R_head>]
        R_head ← <R_head> + 1
        M[head] ← <R_head>
        process(R)
```

The program is written assuming instructions are executed in order.

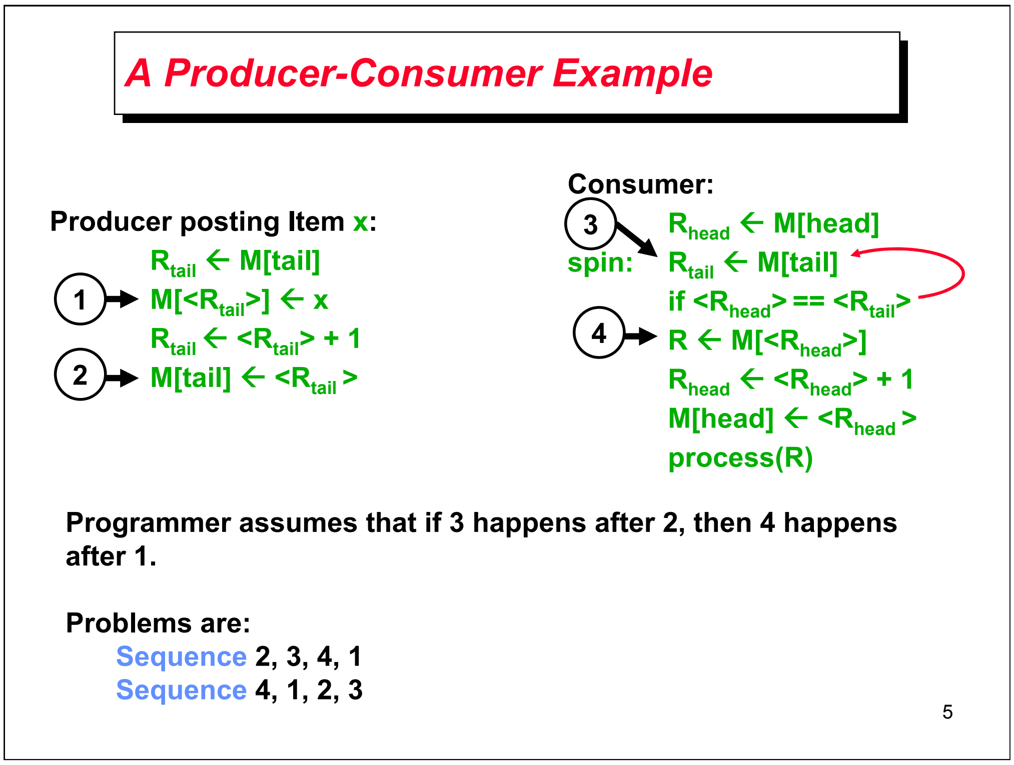
Possible problems?

4

What is the problem?

Suppose the tail pointer gets updated before the item x is stored?

Suppose R is loaded before x has been stored?



A Producer-Consumer Example

Producer posting Item x:

```
        $R_{tail} \leftarrow M[tail]$
1 →     $M[<R_{tail}>] \leftarrow x$
        $R_{tail} \leftarrow <R_{tail}> + 1$
2 →     $M[tail] \leftarrow <R_{tail}>$
```

Consumer:

```
3 ↘     $R_{head} \leftarrow M[head]$
spin:   $R_{tail} \leftarrow M[tail]$
        if $<R_{head}> == <R_{tail}>$
4 →     $R \leftarrow M[<R_{head}>]$
        $R_{head} \leftarrow <R_{head}> + 1$
        $M[head] \leftarrow <R_{head}>$
        process(R)
```

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

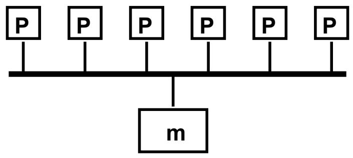
Problems are:

Sequence **2, 3, 4, 1**
Sequence **4, 1, 2, 3**

5

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

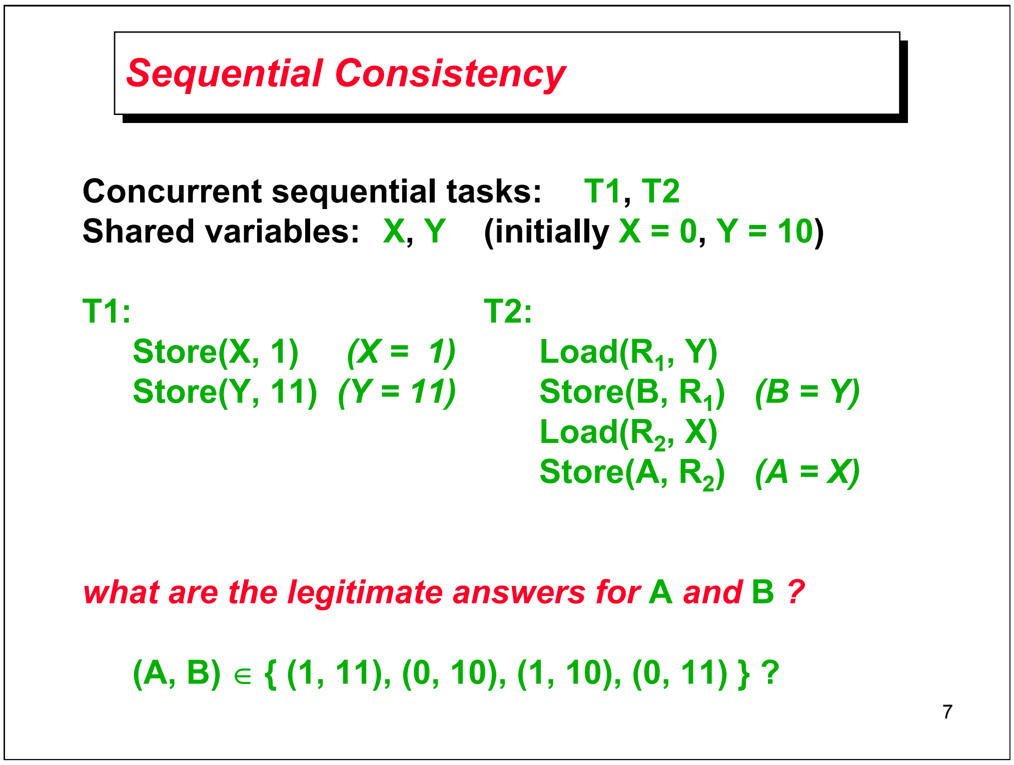
Sequential Consistency: A Memory Model



"A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =
arbitrary *order-preserving interleaving*
of memory references of sequential programs



Sequential Consistency

Concurrent sequential tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

```
T1:                              T2:
    Store(X, 1)    (X = 1)           Load(R_1, Y)
    Store(Y, 11)   (Y = 11)          Store(B, R_1)   (B = Y)
                                     Load(R_2, X)
                                     Store(A, R_2)   (A = X)
```

what are the legitimate answers for A and B ?

(A, B) $\in$ { (1, 11), (0, 10), (1, 10), (0, 11) } ?

7

(0, 11) is not legit.

Sequential Consistency

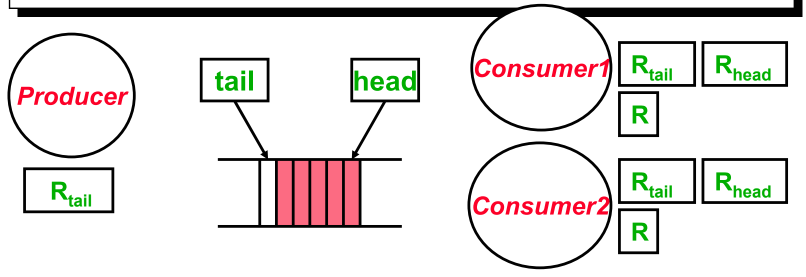
Sequential consistency imposes additional memory ordering constraints in addition to those imposed by uniprocessor program dependencies

What are these in our example ?

Does (can) a system with caches, write buffers, or out-of-order execution capability provide a *sequentially consistent* view of the memory ?

More on this later

Multiple Consumer Example



Producer posting Item x:

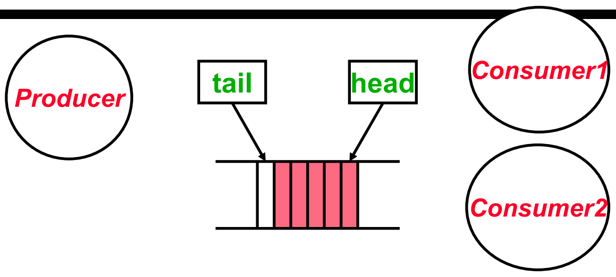
```
$R_{tail} \leftarrow M[tail]$
$M[<R_{tail}>] \leftarrow x$
$R_{tail} \leftarrow <R_{tail}> + 1$
$M[tail] \leftarrow <R_{tail}>$
```

Consumer:

```
       $R_{head} \leftarrow M[head]$
spin:  $R_{tail} \leftarrow M[tail]$
       if $<R_{head}> == <R_{tail}>$
       $R \leftarrow M[<R_{head}>]$
       $R_{head} \leftarrow <R_{head}> + 1$
       $M[head] \leftarrow <R_{head}>$
       process(R)
```

What is wrong with this code?

Multiple Consumer Example



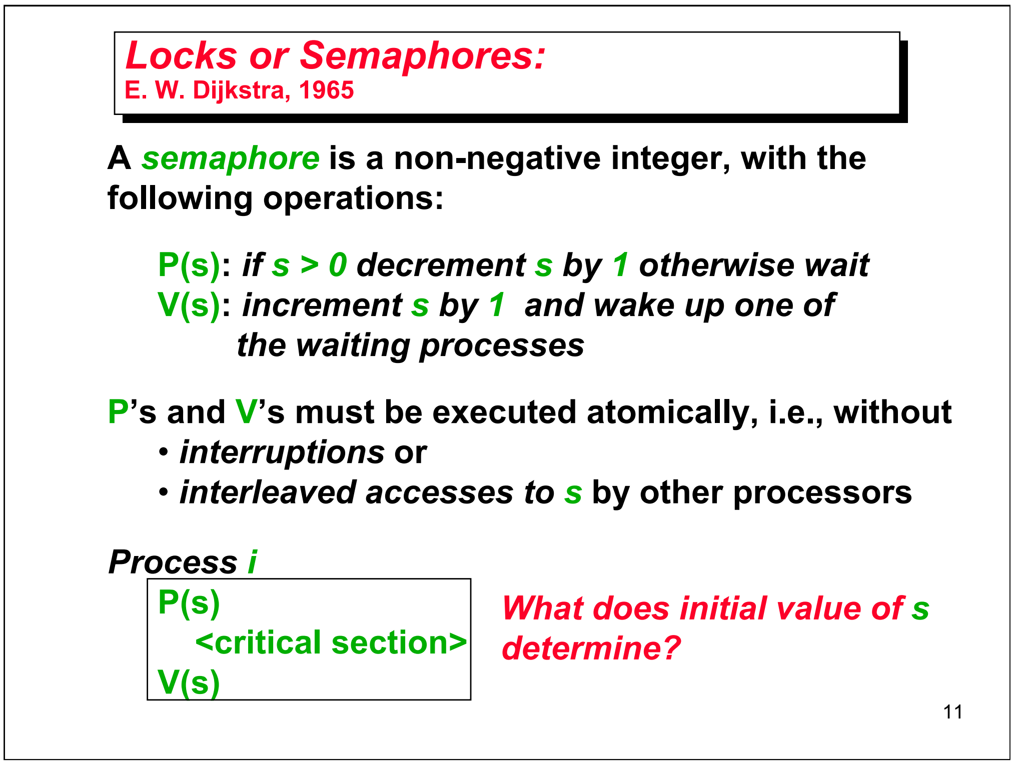
Producer posting Item x:

```
$R_{tail} \leftarrow M[tail]$
$M[<R_{tail}>] \leftarrow x$
$R_{tail} \leftarrow <R_{tail}> + 1$
$M[tail] \leftarrow <R_{tail}>$
```

Critical Section:
Needs to be executed atomically
by one consumer ⇒ locks

Consumer:

```
       $R_{head} \leftarrow M[head]$
spin:  $R_{tail} \leftarrow M[tail]$
       if $<R_{head}> == <R_{tail}>$
       $R \leftarrow M[<R_{head}>]$
       $R_{head} \leftarrow <R_{head}> + 1$
       $M[head] \leftarrow <R_{head}>$
       process(R)
```



Locks or Semaphores:
E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

> **P(s): *if s > 0 decrement s by 1 otherwise wait***
> **V(s): *increment s by 1 and wake up one of the waiting processes***

P's and V's must be executed atomically, i.e., without
- ***interruptions* or**
- ***interleaved accesses to s* by other processors**

Process i

```
P(s)
   <critical section>
V(s)
```

What does initial value of s determine?

11

The maximum number of processes in the critical section.

A sempahore is a visual system for sending information based on 2 flags held

In each hand.

Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

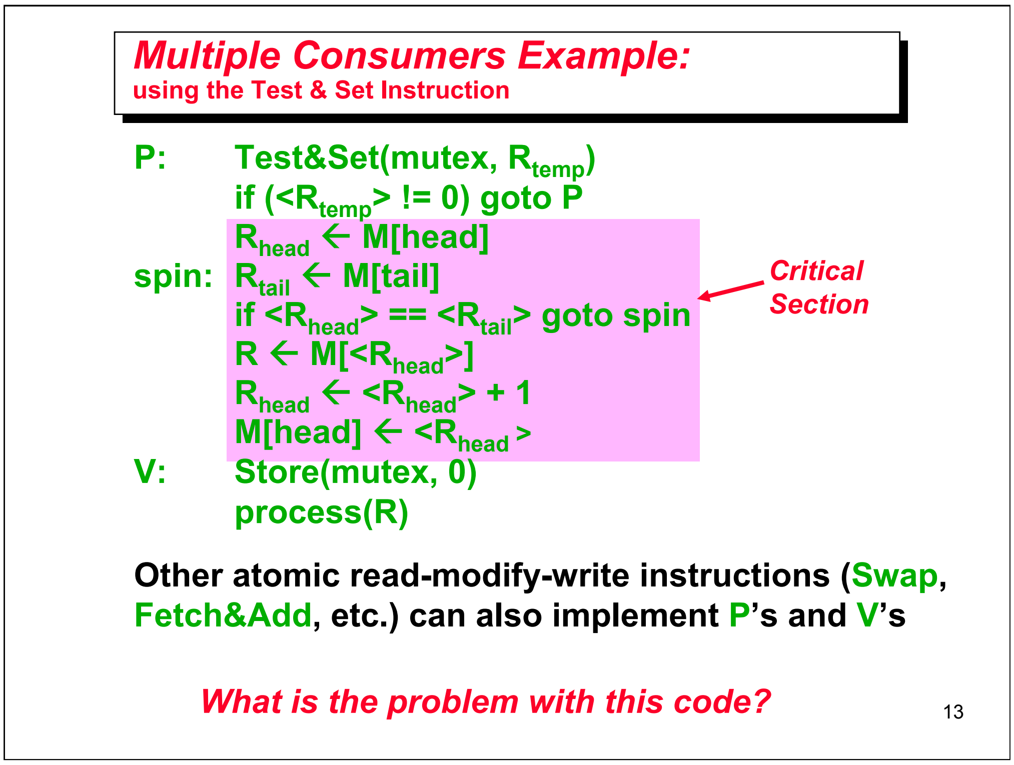
atomic read-modify-write instructions

Examples: ***(a is a memory address, R is a register)***

```
Test&Set(a, R):
   R ← M[a];
   if <R>==0 then
     M[a] ← 1;
```

```
Fetch&Add(a, $R_V$, R):
   R ← M[a];
   M[a] ← <R> + <$R_V$>;
```

```
Swap(a, R):
   $R_t$ ← M[a];
   M[a] ← <R>;
   R ← <$R_t$>;
```



Multiple Consumers Example:
using the Test & Set Instruction

```
P:      Test&Set(mutex, $R_{temp}$)
        if (<$R_{temp}$> != 0) goto P
        $R_{head} \leftarrow$ M[head]
spin:   $R_{tail} \leftarrow$ M[tail]
        if <$R_{head}$> == <$R_{tail}$> goto spin
        R $\leftarrow$ M[<$R_{head}$>]
        $R_{head} \leftarrow$ <$R_{head}$> + 1
        M[head] $\leftarrow$ <$R_{head}$ >
V:      Store(mutex, 0)
        process(R)
```

Critical Section

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P's and V's

What is the problem with this code?

13

What if the process stops or is swapped out while in the critical section?

Nonblocking Synchronization

```
Compare&Swap(a, $R_t$, $R_s$):     *implicit arg* - status
        if (<$R_t$> == M[a])
            then        M[a] $\leftarrow$ <$R_s$>;
                        $R_t \leftarrow$ <$R_s$>;
                        status $\leftarrow$ success;
            else        status $\leftarrow$ fail;

    try:    $R_{head} \leftarrow$ M[head]
    spin:   $R_{tail} \leftarrow$ M[tail]
            if <$R_{head}$> == <$R_{tail}$> goto spin
            R $\leftarrow$ M[<$R_{head}$>]
            $R_{newhead} \leftarrow$ <$R_{head}$> + 1
            Compare&Swap(head, $R_{head}$, $R_{newhead}$)
            if (status == fail) goto try
            process(R)
```

Load-reserve & Store-conditional

Non-blocking Synchronization

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve(R, a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional(a, R):
    if <flag, adr> == <1, a>
    then  cancel other procs'
          reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-reserve($R_{head}$, head)
spin:   $R_{tail}$ ← M[tail]
        if <$R_{head}$> == <$R_{tail}$> goto spin
        R ← M[<$R_{head}$>]
        $R_{head}$ ← <$R_{head}$> + 1
        Store-conditional(head, $R_{head}$)
        if (status == fail) goto try
        process(R)
```

Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2. Initially, both c1 and c2 are 0 *(not busy)*

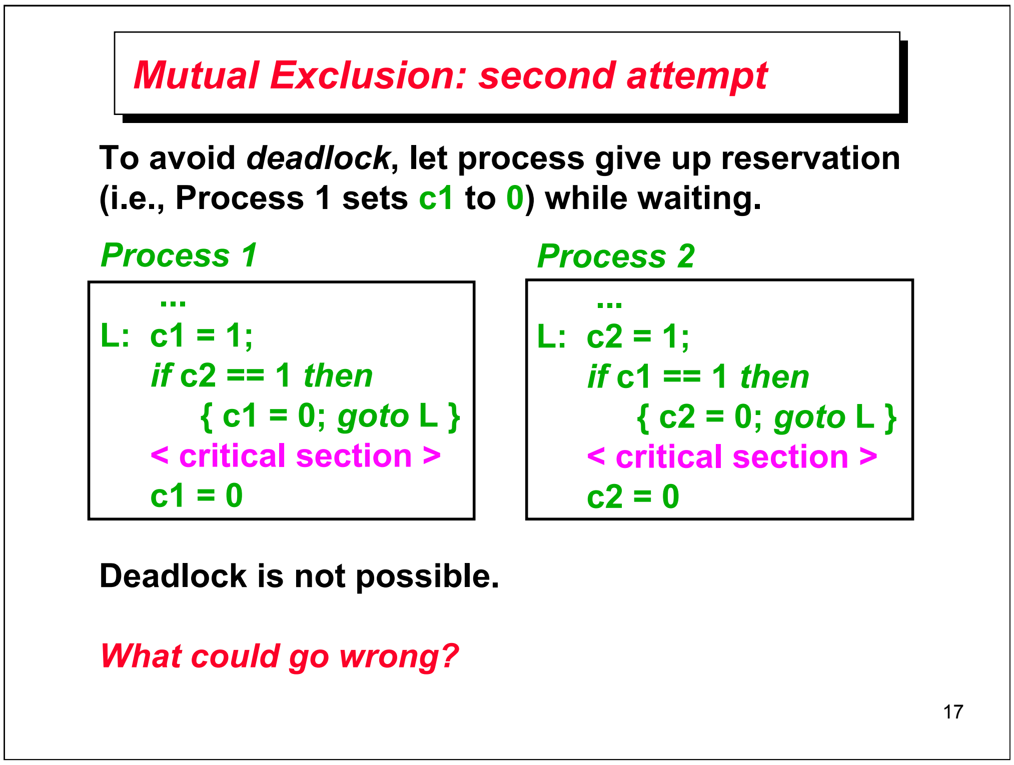
Process 1

```
     ...
     c1 = 1;
L:   if c2 == 1 then go to L
     < critical section >
     c1 = 0;
```

Process 2

```
     ...
     c2 = 1;
L:   if c1 == 1 then go to L
     < critical section >
     c2 = 0;
```

What is wrong? ______________________________



Mutual Exclusion: second attempt

To avoid *deadlock*, let process give up reservation (i.e., Process 1 sets c1 to 0) while waiting.

Process 1

```
     ...
L:  c1 = 1;
    if c2 == 1 then
        { c1 = 0; goto L }
    < critical section >
    c1 = 0
```

Process 2

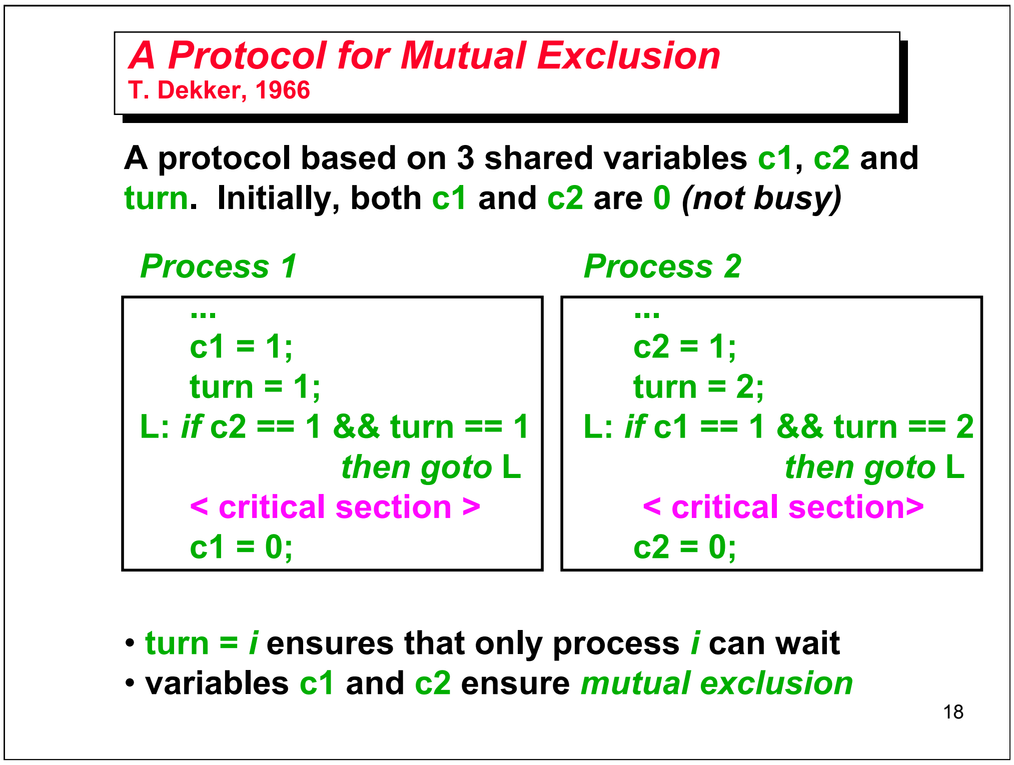
```
     ...
L:  c2 = 1;
    if c1 == 1 then
        { c2 = 0; goto L }
    < critical section >
    c2 = 0
```

Deadlock is not possible.

What could go wrong?

17

This is the most promising solution, but alas, we still have a problem with *bounded waiting* . Suppose Process *j* continually reenters its entry protocol after leaving its exit protocol, while Process *i* is waiting. It is *possible* That Process *j* will repeatedly reach the while test when Process *i* has temporarily cleared its flag. We cannot place a bound on how many times this could happen.



A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn. Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
     ...
     c1 = 1;
     turn = 1;
L: if c2 == 1 && turn == 1
                then goto L
     < critical section >
     c1 = 0;
```

Process 2

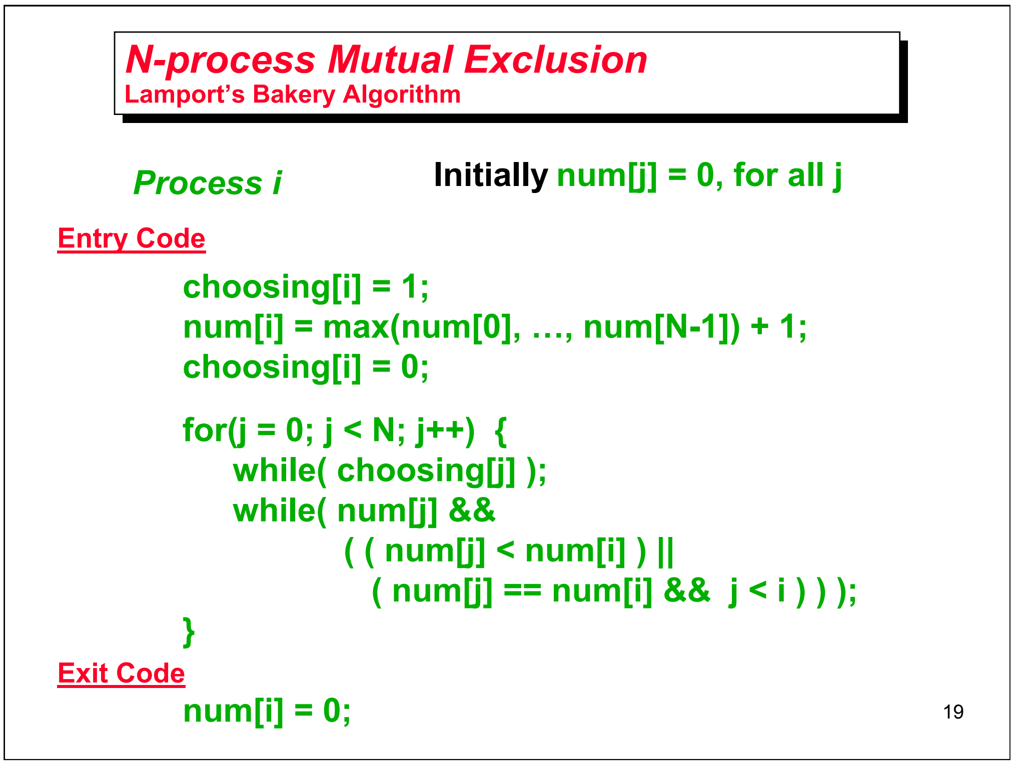
```
     ...
     c2 = 1;
     turn = 2;
L: if c1 == 1 && turn == 2
                then goto L
      < critical section>
     c2 = 0;
```

- **turn = *i* ensures that only process *i* can wait**
- **variables c1 and c2 ensure *mutual exclusion***

18

Take a number approach used in bakeries.

Never seen one in bakeries, but the RMV uses one.



N-process Mutual Exclusion

Lamport's Bakery Algorithm

Process i **Initially** num[j] = 0, for all j

Entry Code

```
choosing[i] = 1;
num[i] = max(num[0], …, num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++)  {
     while( choosing[j] );
     while( num[j] &&
                ( ( num[j] < num[i] ) ||
                   ( num[j] == num[i] &&  j < i ) ) );
}
```

Exit Code

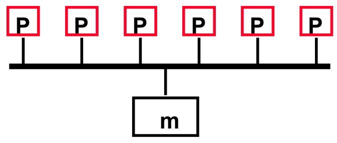
```
num[i] = 0;
```

19

Wait if the process is currently choosing

Wait if the process has a number and comes ahead of us.

Implementation Issues



Implementation of SC is complicated by two issues

- *Out-of-order execution capability*

Load(a); Load(b)	*yes*
Load(a); Store(b)	*yes if* a $\neq$ b
Store(a); Load(b)	*yes if* a $\neq$ b
Store(a); Store(b)	*yes if* a $\neq$ b

- *Caches*

  Caches can prevent the effect of a store from being seen by other processors

Memory Fences:
Instructions to serialize memory accesses

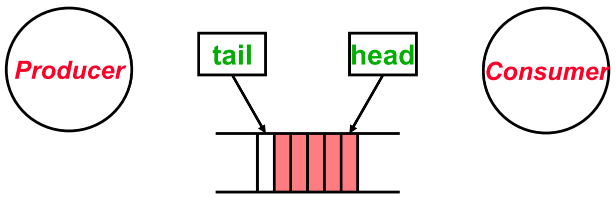
Processors with *relaxed or weak memory models* (i.e., permit Loads and Stores to different addresses to be reordered) need *memory fence* instructions to force serialization of memory accesses

Processors with relaxed memory models:

Sparc V8 (TSO, PSO): Membar
PowerPC (WO): Sync, EIEIO

Memory fences are expensive operations, however, one pays for serialization only when it is required

Using Memory Fences



Producer posting Item x:

```
$R_{tail} \leftarrow M[tail]$
$M[<R_{tail}>] \leftarrow x$
$membar_{SS}$
$R_{tail} = <R_{tail}> + 1$
$M[tail] \leftarrow <R_{tail}>$
```

What does this ensure?

Consumer:

```
        $R_{head} \leftarrow M[head]$
spin:   $R_{tail} \leftarrow M[tail]$
        if $<R_{head}> == <R_{tail}>$
        $membar_{LL}$
        $R \leftarrow M[<R_{head}>]$
        $R_{head} \leftarrow <R_{head}> + 1$
        $M[head] \leftarrow <R_{head}>$
        process(R)
```

What does this ensure?

22

Ensures that tail pointer is not updated before
X has been stored.

Ensures that R is not loaded before x has been stored.