Process Synchronization

Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers. We can do so by having an integer **count** that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /*  produce an item and put in
  nextProduced  */
      while (count == BUFFER_SIZE)
            ; // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
              ; // do nothing
              nextConsumed =  buffer[out];
               out = (out + 1) %
BUFFER_SIZE;
                  count--;
               /*  consume the item in
nextConsumed
}
```

Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2

- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}

  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}

Solution to Critical-Section Problem

Requirements:

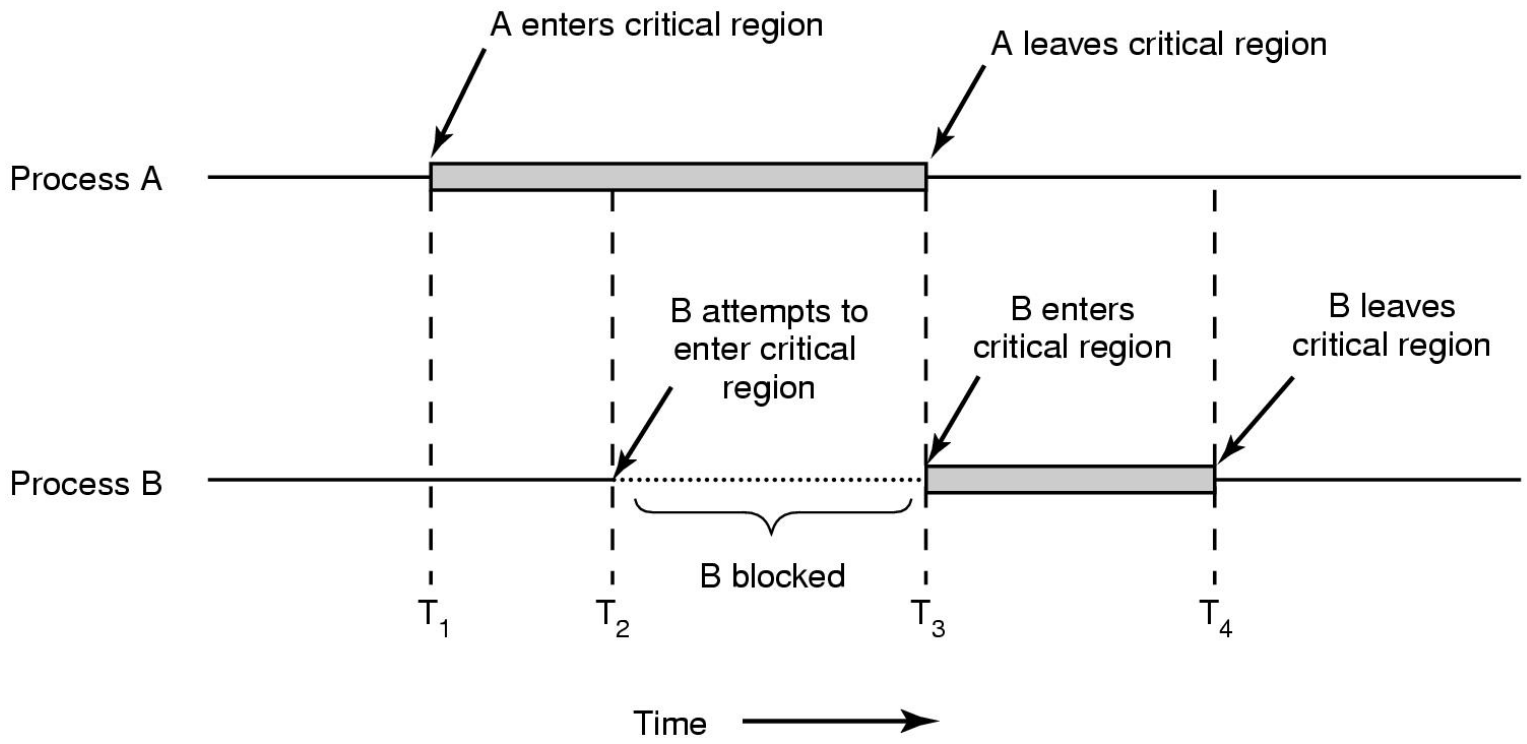
1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the **N** processes

Properties of a good solution

- Mutual exclusion
- No assumptions about speeds or number of CPU’s
- No process outside critical region can block other processes
- No starvation-no process waits forever to enter critical region

.

What we are trying to do



First attempts-Busy Waiting

A laundry list of proposals to achieve mutual exclusion

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Disabling Interrupts

- Idea: process disables interrupts, enters cr, enables interrupts when it leaves cr.
- Without interrupt no process switching occur.
- Problems
  - Process might never enable interrupts, crashing system
  - One process can hold up the whole system.

    (i) endless loop ii)waiting for resources.
  - Won’t work on multi-core chips as disabling interrupts only effects one CPU at a time

Lock variables

- A software solution-everyone shares a lock
  - When lock is 0, process turns it to 1 and enters cr
  - When exit cr, turn lock to 0
- Problem-Race condition

Case 1: Strict Alternation

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

(a)

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

(b)

Problems with strict alternation

- Employs busy waiting-while waiting for the cr, a process spins
- If one process is outside the cr and it is its turn, then other process has to wait until outside guy finishes both outside AND inside (cr) work

Case 2:

```
P0

While(1)
{
        flag[0]=T;
        while(flag[1]);
        critical section
        flag[0]=F;
}
```

```
P1

While(1)
{
        flag[1]=T;
        while(flag[0]);
        critical section
        flag[1]=F;
}
```

Problems with case 2

- Employs busy waiting-while waiting for the cr, a process spins
- Progress condition is not satisfied properly. It may leads to deadlock.

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int **turn**;
  - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process $\mathbf{P_i}$ is ready!

Case 3:Peterson's Algorithm

P0

```
While(1)
{
        flag[0]=T;
        turn=1;
        while(turn==1 && flag[1]==T);
        critical section
        flag[0]=F;
}
```

P1

```
While(1)
{
        flag[1]=T;
        turn=0;
        while(turn==0 && flag[0]==T);
        critical section
        flag[1]=F;
}
```

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

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
            critical section
    release lock
            remainder section
} while (TRUE);
```

TestAndSet Instruction

- Definition:

```
boolean  TestAndSet (boolean *target)
{
    boolean  rv = *target;
    *target = TRUE;
    return  rv;
}
```

Must be executed atomically

Solution using TestAndSet

- Shared Boolean variable lock, initialized to false.
- Solution:

```
do {
        while ( TestAndSet (&lock ))
                ;   // do nothing
              //    critical section
        lock = FALSE;
                //      remainder
section
    } while (TRUE);
```

Swap Instruction

- Definition:

```
void Swap (boolean *a, boolean *b)
 {
        boolean temp = *a;
        *a = *b;
        *b = temp:
 }
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
                    //    critical section
         lock = FALSE;
                    //      remainder section
 } while (TRUE);
```

The Producer-Consumer Problem (aka Bounded Buffer Problem)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

The problem with sleep and wake-up calls

- Empty buffer,count==0
- Consumer gets replaced by producer before it goes to sleep
- Produces something, count++, sends wakeup to consumer
- Consumer not asleep, ignores wakeup, thinks count= = 0, goes to sleep
- Producer fills buffer, goes to sleep
- P and C sleep forever
- So the problem is lost wake-up calls

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
} while (TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }

Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
   wait (mutex);
        // Critical Section
     signal (mutex);
     // remainder section
} while (TRUE);
```

Example Semaphore

P1

process 1 statements

```
signal(synch);
```

done executing

P2

since `synch = 0`,
must stay idle until
signal from P1

```
wait(synch);
```

received signal from P1

process 2 statements

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
- Sleeping Barber Problem

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Semaphores

- Semaphore is an integer variable
- Used to sleeping processes/wakeups
- Two operations, down and up
- Down checks semaphore. If not zero, decrements semaphore. If zero, process goes to sleep
- Up increments semaphore. If more then one process asleep, one is chosen randomly and enters critical region (first does a down)
- ATOMIC IMPLEMENTATION-interrupts disabled

Producer Consumer with Semaphores

- 3 semaphores: full, empty and mutex
- Full counts full slots (initially 0)
- Empty counts empty slots (initially N)
- Mutex protects variable which contains the items produced and consumed

Producer Consumer with semaphores

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

. . .

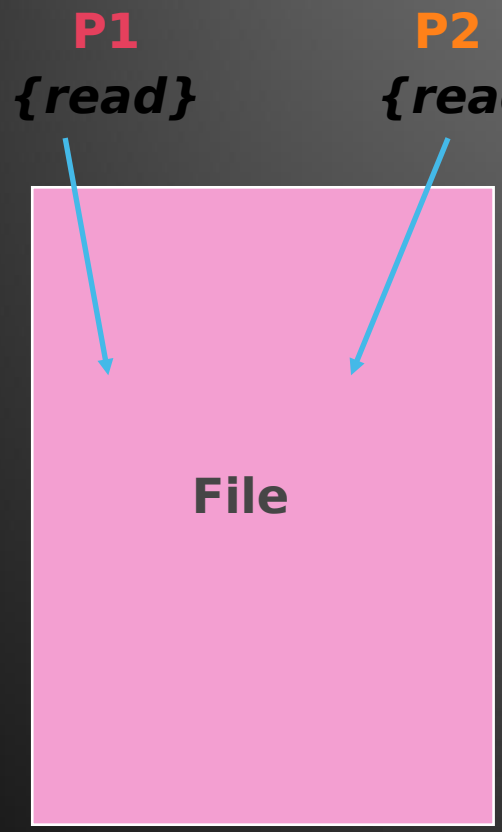
Readers–Writers Problem

- A data set is shared among a number of concurrent processes.
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1 (controls access to readcount)
  - Semaphore wrt initialized to 1 (writer access)
  - Integer readcount initialized to 0 (how many processes are reading object)

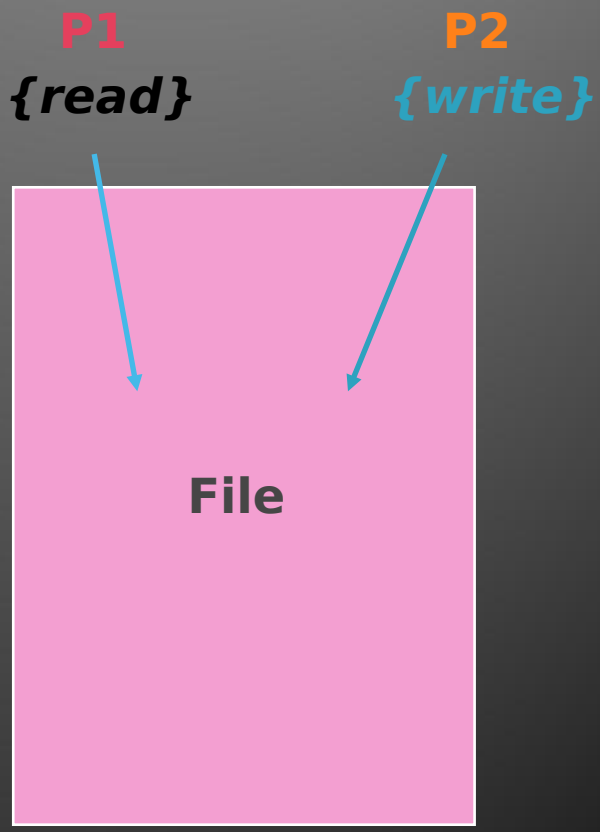
Readers–Writers

- concurrent processes share a file, record, or other resources
- some may read only (readers), some may write (writers)
- two concurrent reads have no adverse effects
- problems if
  - concurrent reads and writes
  - multiple writes

Readers-Writers Diagram



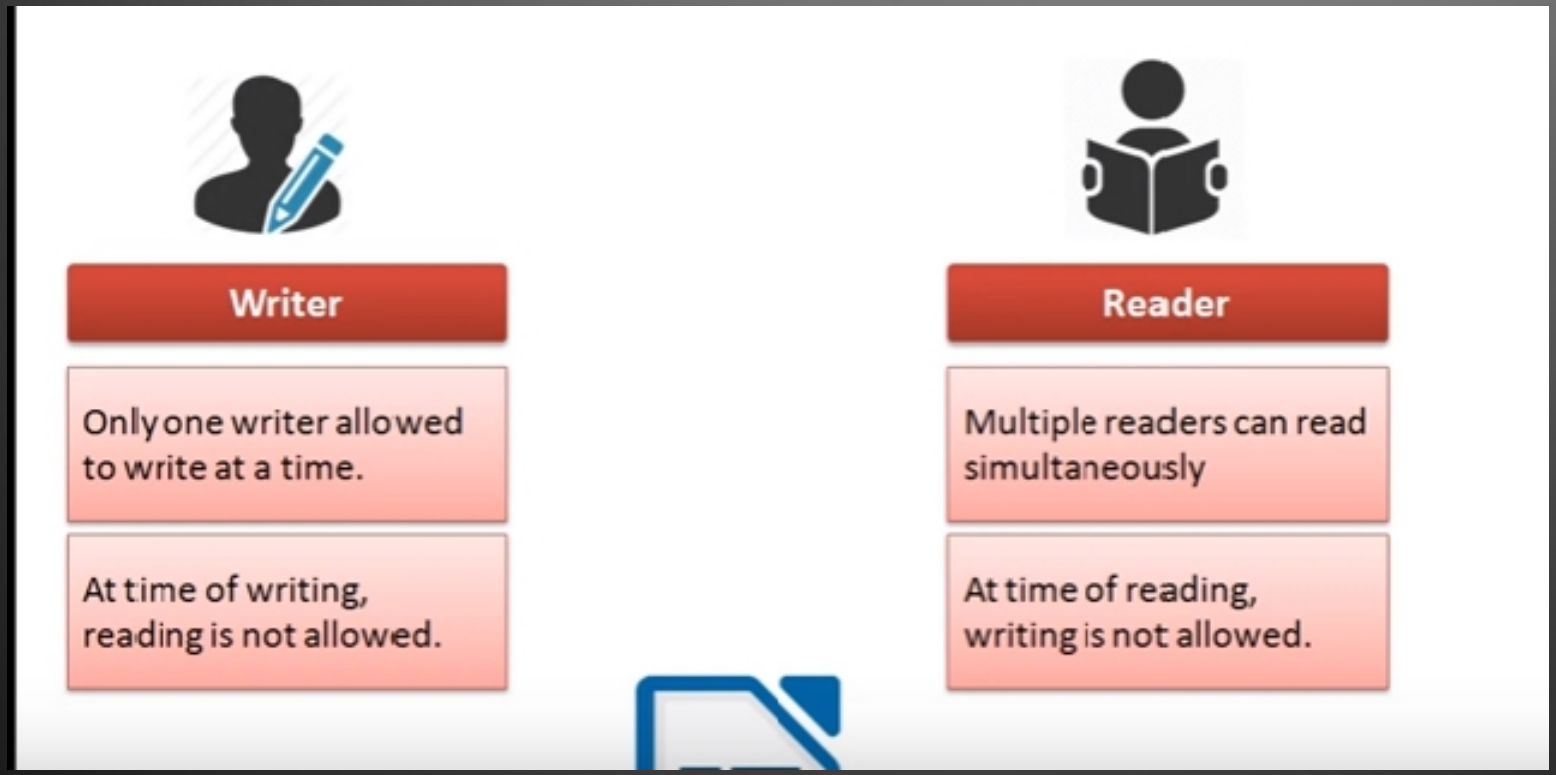
No problem



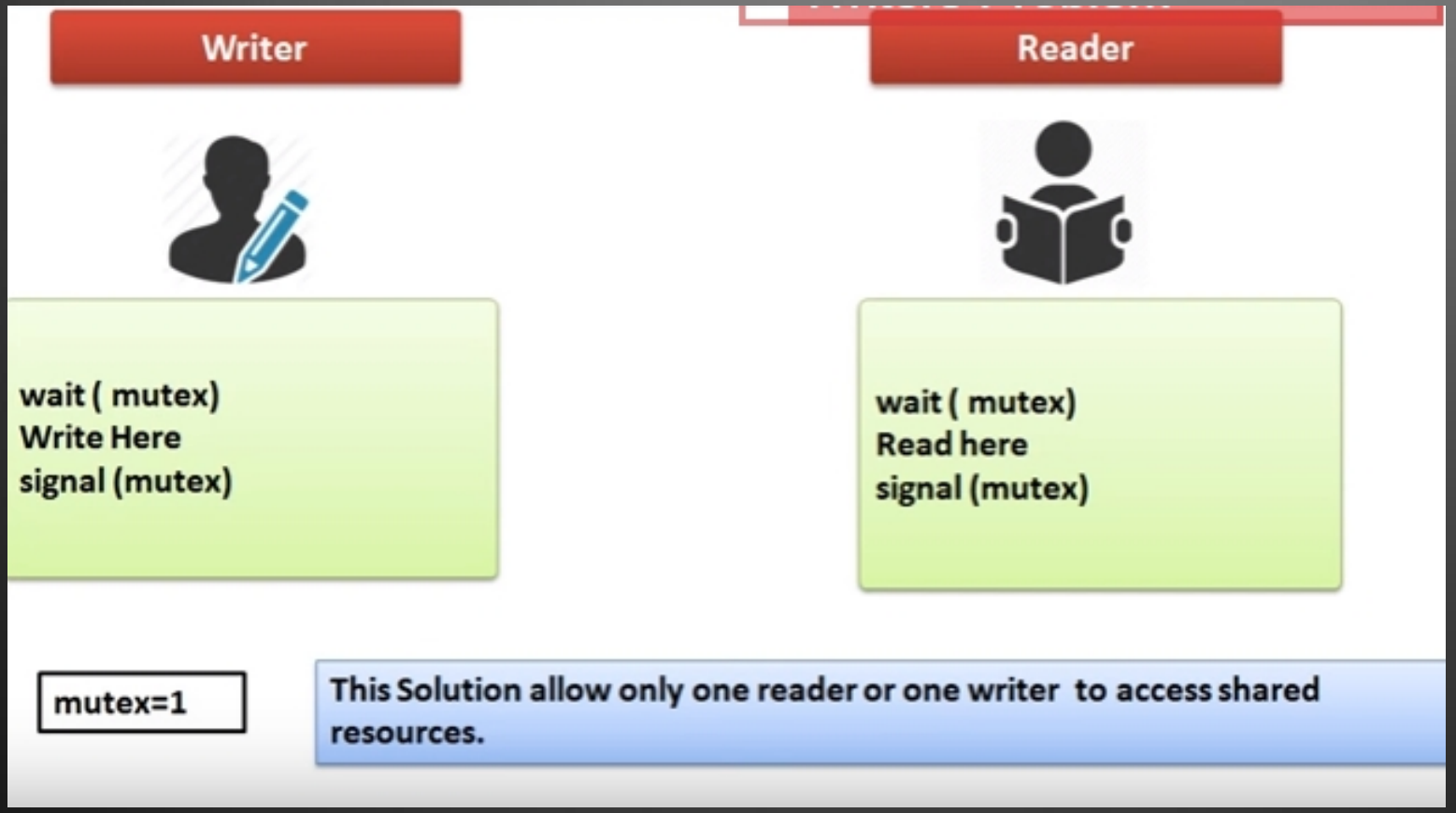
Problem!

[Son 97]

Readers writers problem



Readers writers problem



Readers–Writers Problem

- The structure of a Reader process

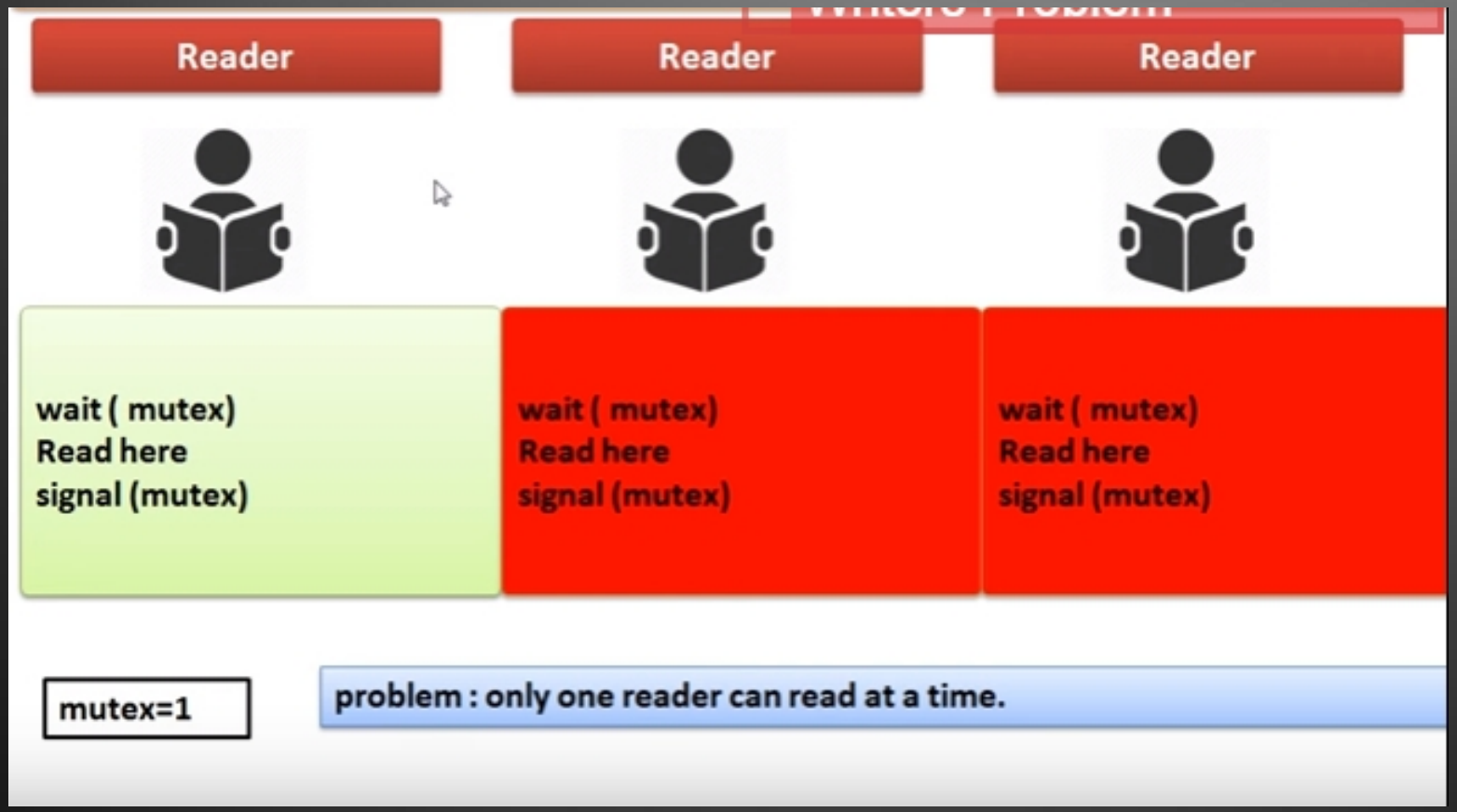
```
do {
        wait (mutex) ;
        //  Reading is performed
        signal (mutex) ;
} while (TRUE);
```

Readers–Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait (mutex) ;
            //    writing is performed
        signal (mutex) ;
} while (TRUE);
```

Readers–writers problem



Readers writers problem

Writer

```
wait ( wrmutex)
Write Here
signal (wrmutex)
```

Reader

```
readers++
if( readers == 1)
    wait ( wrmutex)

Read here

readers--
if(readers==0)
signal (wrmutex)
```

mutex=1

Multiple Readers can execute simultaneously and they might try to access value of readers at same time(race condition).

Readers writers problem

Writer

```
wait ( wrmutex)
Write Here
signal (wrmutex)
```

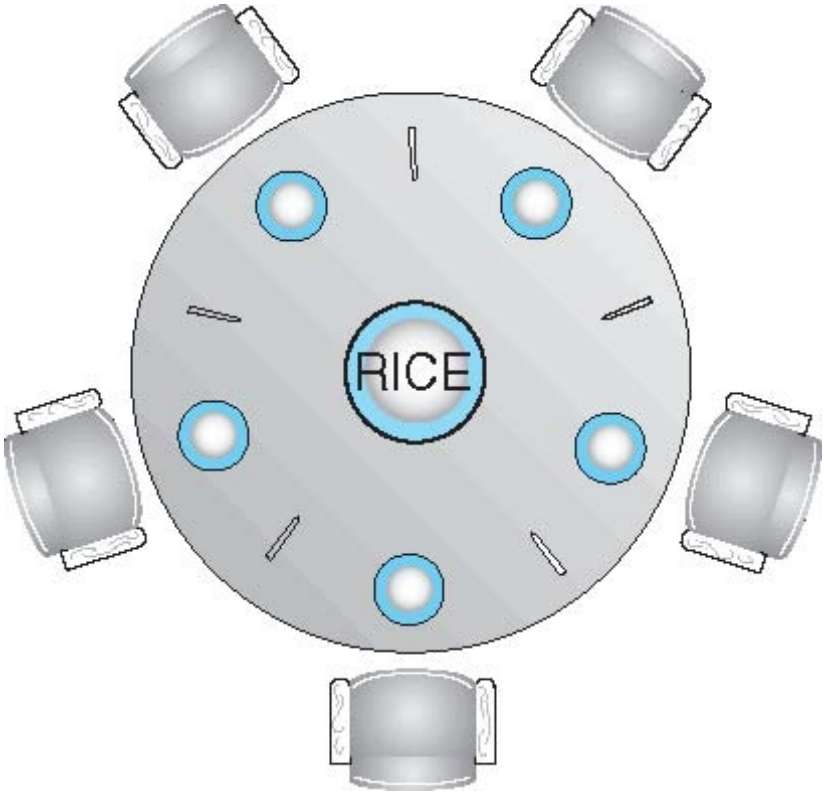
mutex=1

Reader

```
wait(mutex)
readers++
if( readers == 1)
    wait ( wrmutex)
signal(mutex)
Read here
wait(mutex)
readers--
if(readers==0)
signal (wrmutex)
signal(mutex)
```

Multiple Readers can execute simultaneously and they might try to access value of readers at same time(race condition).

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining Philosopher Problem

Solution - 1 Take left fork first and then take second fork.

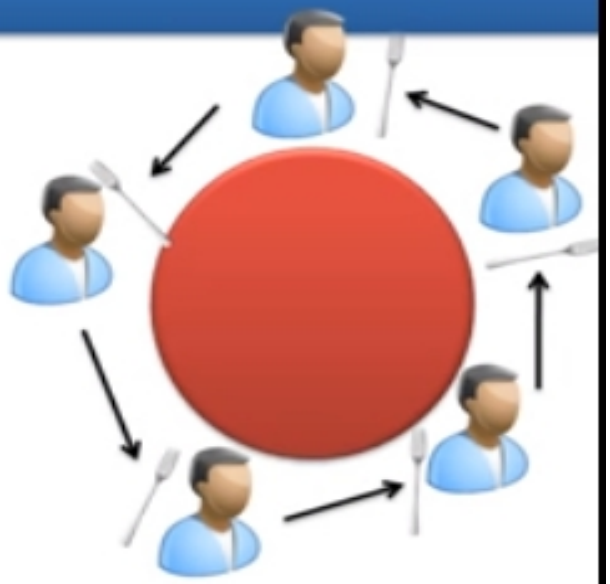
```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```

Dining Philosopher Problem

Problem : Deadlock

Solution - 1

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```



Deadlock State

Two neighbours can't eat at same time.

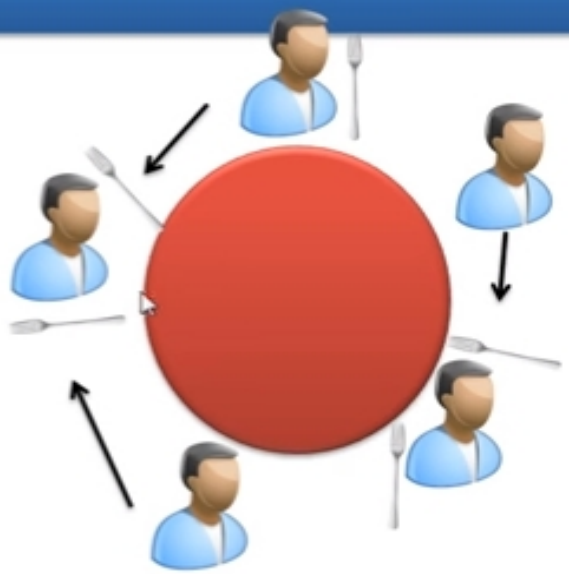
If all five philospher become hungry at same time then they pick their left chopstick first which will make all chopstick value to 0 hence they will wait forever for right chop stick.

Dining Philosopher Problem

Problem : Starvation

Solution - 1

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```



Deadlock State

If two philosophers are faster than others, they are thinking fast and eating fast in this case all the time only these two philosopher will occupy forks.

Dining Philosopher Problem

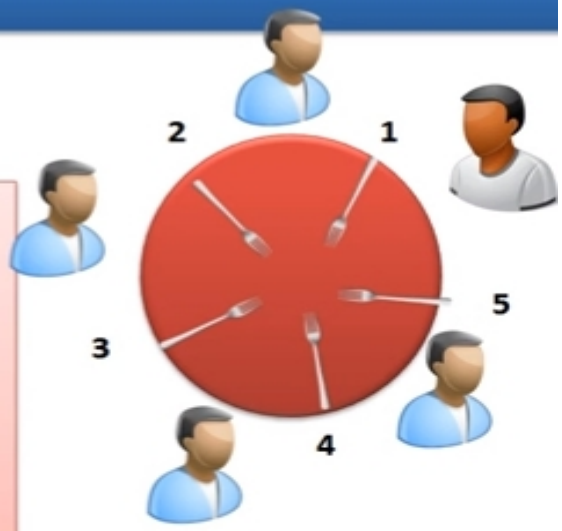
Solution - 2

Lefty

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```

Righty

```
do {
wait(chopstick[(i+1)%5];
wait(chopstick[i]);

// Critical Section
eat

signal(chopstick[(i+1)%5];
signal(chopstick[i]);

think
}while(true);
```



If two philosophers are faster than others, they are thinking fast and eating fast in this case all the time only these two philosopher will occupy forks.

Dining Philosopher Problem

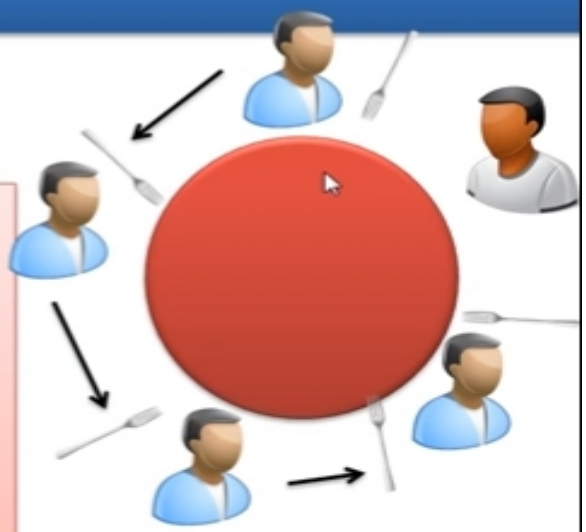
Solution - 2 **Problem : Starvation**

Lefty

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1)%5];

// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];

think
}while(true);
```

Righty

```
do {
wait(chopstick[(i+1)%5];
wait(chopstick[i]);

// Critical Section
eat

signal(chopstick[(i+1)%5];
signal(chopstick[i]);

think
}while(true);
```



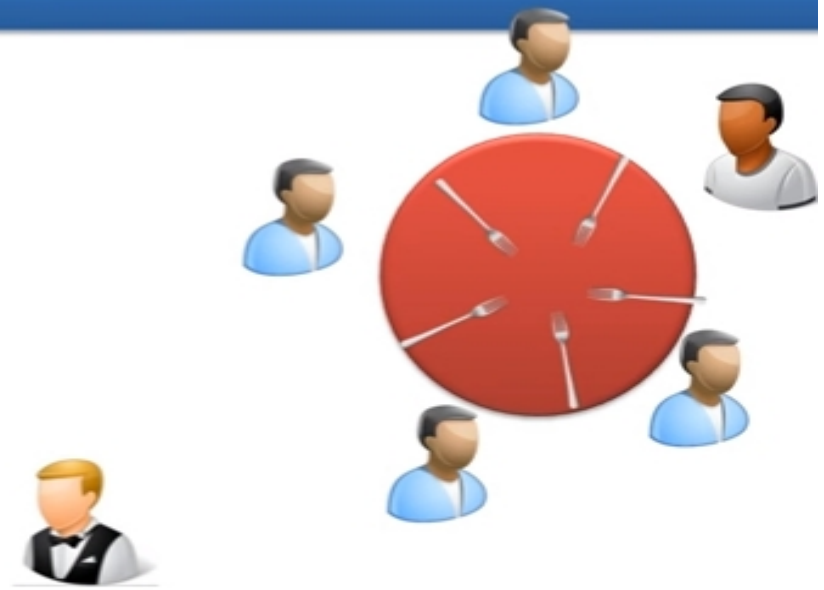
There won't be any deadlock. Circular waiting condition is unsatisfied.

Dining Philosopher Problem

Solution - 3

Use of Arbitrator

```
do {
wait(mutex)

wait(chopstick[i]);
wait(chopstick[(i+1)%5];
signal(mutex)
// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];
think
}while(true);
```



Philosopher has to ask waiter before picking forks. Waiter can give permission to only one philosopher at a time until he has picked both forks.

Dining Philosopher Problem

Solution - 3

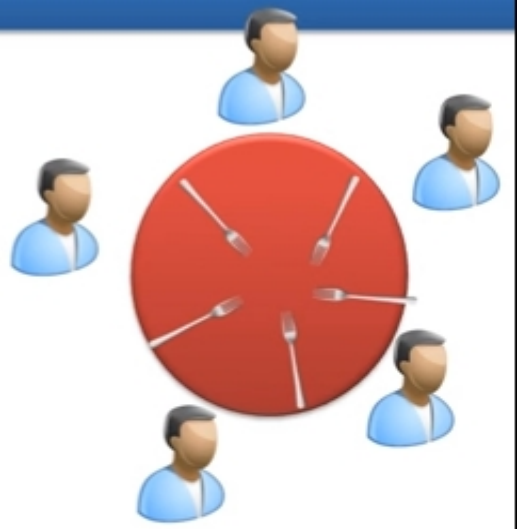
Use of Arbitrator

```
do {
wait(mutex)

wait(chopstick[i]);
wait(chopstick[(i+1)%5];
signal(mutex)
// Critical Section
eat

signal(chopstick[i]);
signal(chopstick[(i+1)%5];
think
}while(true);
```

if a philosopher is eating and one of his neighbors is requesting the forks, all other philosophers must wait until this request has been fulfilled even if forks for them are still available.

Dining Philosopher Problem

Solution - 4 **Tannenbaum's Solution**



HUNGRY
THINKING
EATING
} State of philosopher

S[5] } Semaphore [initially all set 0]

me } Semaphore [initially all set 1]

Dining Philosopher Problem

Solution - 4 Tannenbaum's Solution

```
while TRUE do
{


THINKING;

take_chopsticks(i);

EATING;

drop_chopsticks(i);


}
```

```
procedure take_chopsticks(i)
{
DOWN(me);
/* critical section */
 pflag[i] := HUNGRY;
 test[i];
 UP(me);
/* end critical section */
DOWN(s[i])
/* Eat if enabled */
}
```

```
procedure drop_chopsticks(int i)
{ DOWN(me);
pflag[i]=THINKING
 /* critical section */
 test(i-1);
/* Let phil. on left eat if possible */
test(i+1);
 /* Let phil. on right eat if possible
*/
UP(me);
/* up critical section */
}
```

Dining Philosopher Problem

Tannenbaum's Solution

```
void test(i)     /* Let phil[i] eat, if waiting */
{
if ( pflag[i] == HUNGRY && pflag[i-1] != EAT && pflag[i+1] != EAT)
{
   pflag[i] := EAT;
   UP(s[i])
 }
 }
```

Dining Philosopher Problem


```
while TRUE do
{


THINKING;

DOWN(me);
pflag[i] := HUNGRY;
 test[i];
 UP(me);
DOWN(s[i]) ;

EATING;

DOWN(me);
pflag[i]=THINKING
test(i-1);
test(i+1);
UP(me);
 }
```

Dining Philosopher Problem

ME : 0	
S[0] : 0	pflag[0] : THINK
S[1] : 0	pflag[1] : THINK
S[2] : 0	pflag[2] : THINK
S[3] : 0	pflag[3] : THINK
S[4] : 0	pflag[4] : THINK

```
while TRUE do
 {


THINKING;

DOWN(me);
pflag[i] := HUNGRY;
 test[i];
 UP(me);
DOWN(s[i]) ;

EATING;

DOWN(me);
pflag[i]=THINKING
test(i-1);
test(i+1);
UP(me);
 }
```

Dining Philosopher Problem

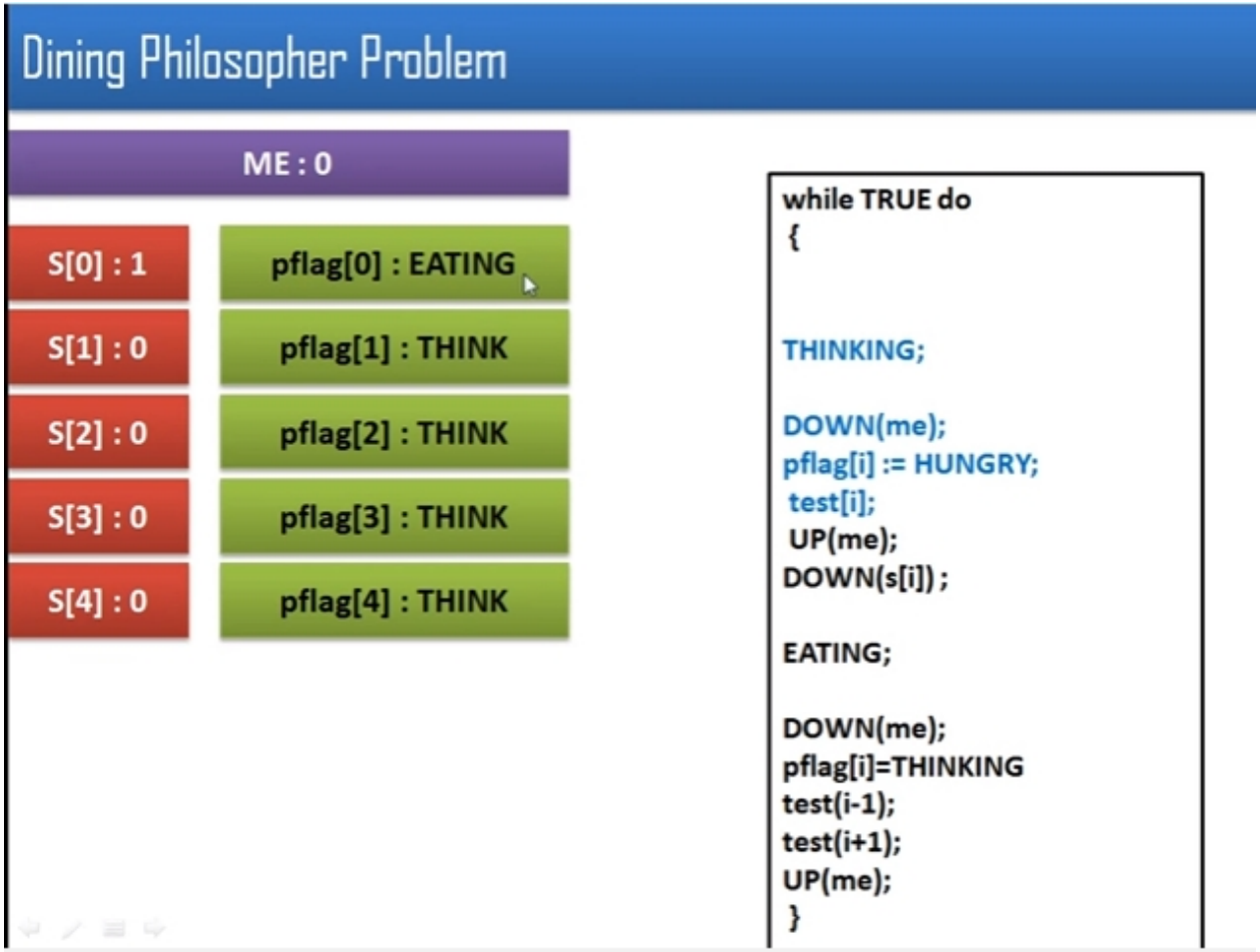
ME : 0

S[0] : 0	pflag[0] : HUNGRY
S[1] : 0	pflag[1] : THINK
S[2] : 0	pflag[2] : THINK
S[3] : 0	pflag[3] : THINK
S[4] : 0	pflag[4] : THINK

If neighbours are not eating then up s[0]

```
while TRUE do
{


THINKING;

DOWN(me);
pflag[i] := HUNGRY;
 test[i];
 UP(me);
DOWN(s[i]) ;

EATING;

DOWN(me);
pflag[i]=THINKING
test(i-1);
test(i+1);
UP(me);
 }
```

Dining Philosopher Problem



Dining Philosopher Problem


```
while TRUE do
{


THINKING;

DOWN(me);
pflag[i] := HUNGRY;
 test[i];
 UP(me);
DOWN(s[i]) ;

EATING;

DOWN(me);
pflag[i]=THINKING
test(i-1);
test(i+1);
UP(me);
 }
```

Dining Philosopher Problem

ME : 0	
S[0] : 0	pflag[0] : THINK
S[1] : 0	pflag[1] : THINK
S[2] : 0	pflag[2] : THINK
S[3] : 0	pflag[3] : THINK
S[4] : 0	pflag[4] : THINK

```
while TRUE do
{


THINKING;

DOWN(me);
pflag[i] := HUNGRY;
 test[i];
 UP(me);
DOWN(s[i]) ;

EATING;

DOWN(me);
pflag[i]=THINKING
test(i-1);
test(i+1);
UP(me);
}
```

Solution to dining philosopher's problem

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N–1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */

typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */


void philosopher(int i)         /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}

void take_forks(int i)          /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}
```

Solution to dining philosopher's problem

```
void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

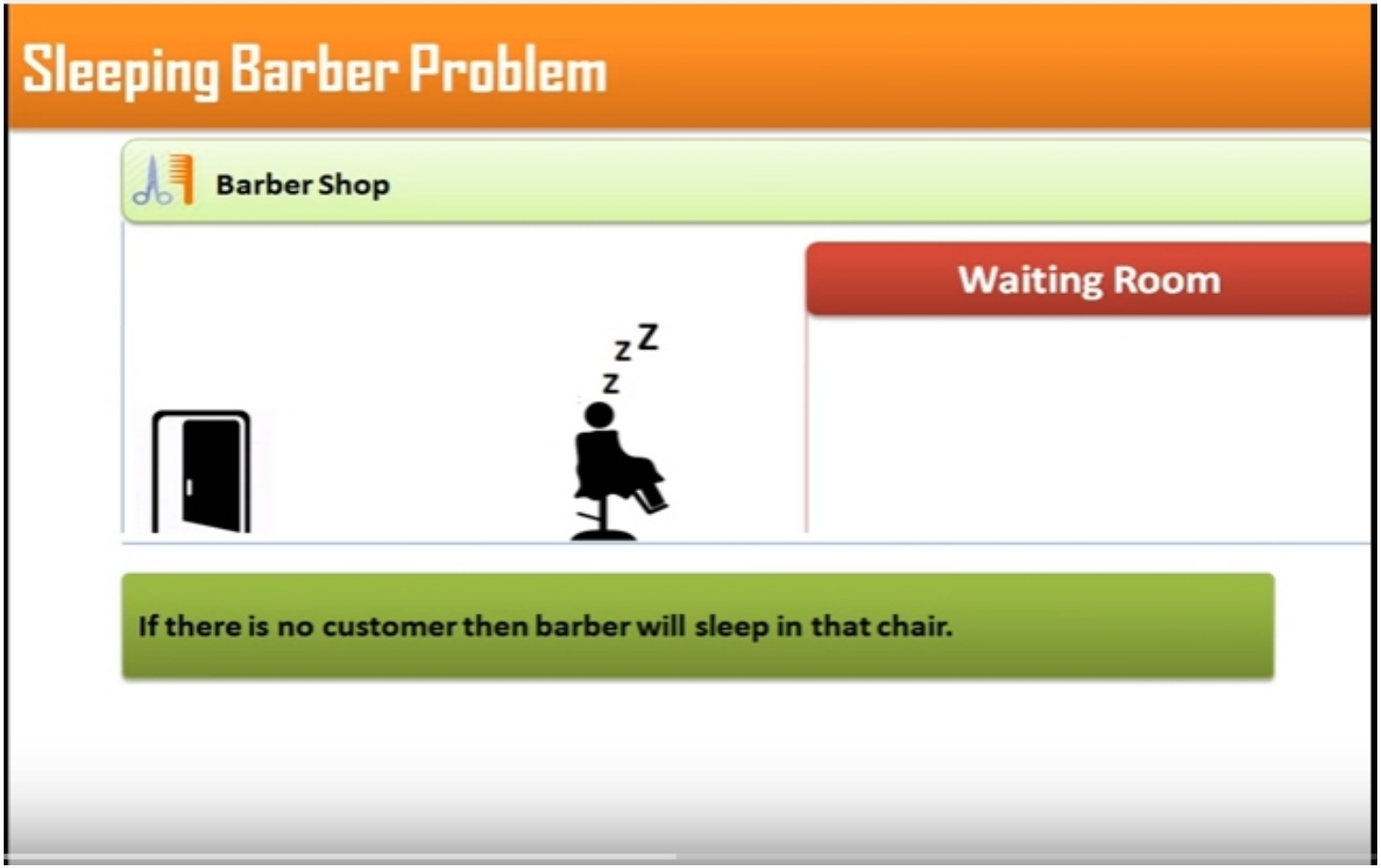
Sleeping Barber's problem

- Imagine a barber shop with one barber, one barber chair, and a number of chairs for waiting customers.
- When there are no customers, the barber sits in his chair and sleeps. As soon as a customer arrives, he either awakens the barber or, if the barber is cutting someone else's hair, sits down in one of the vacant chairs.
- If all of the chairs are occupied, the newly arrived customer simply leaves.

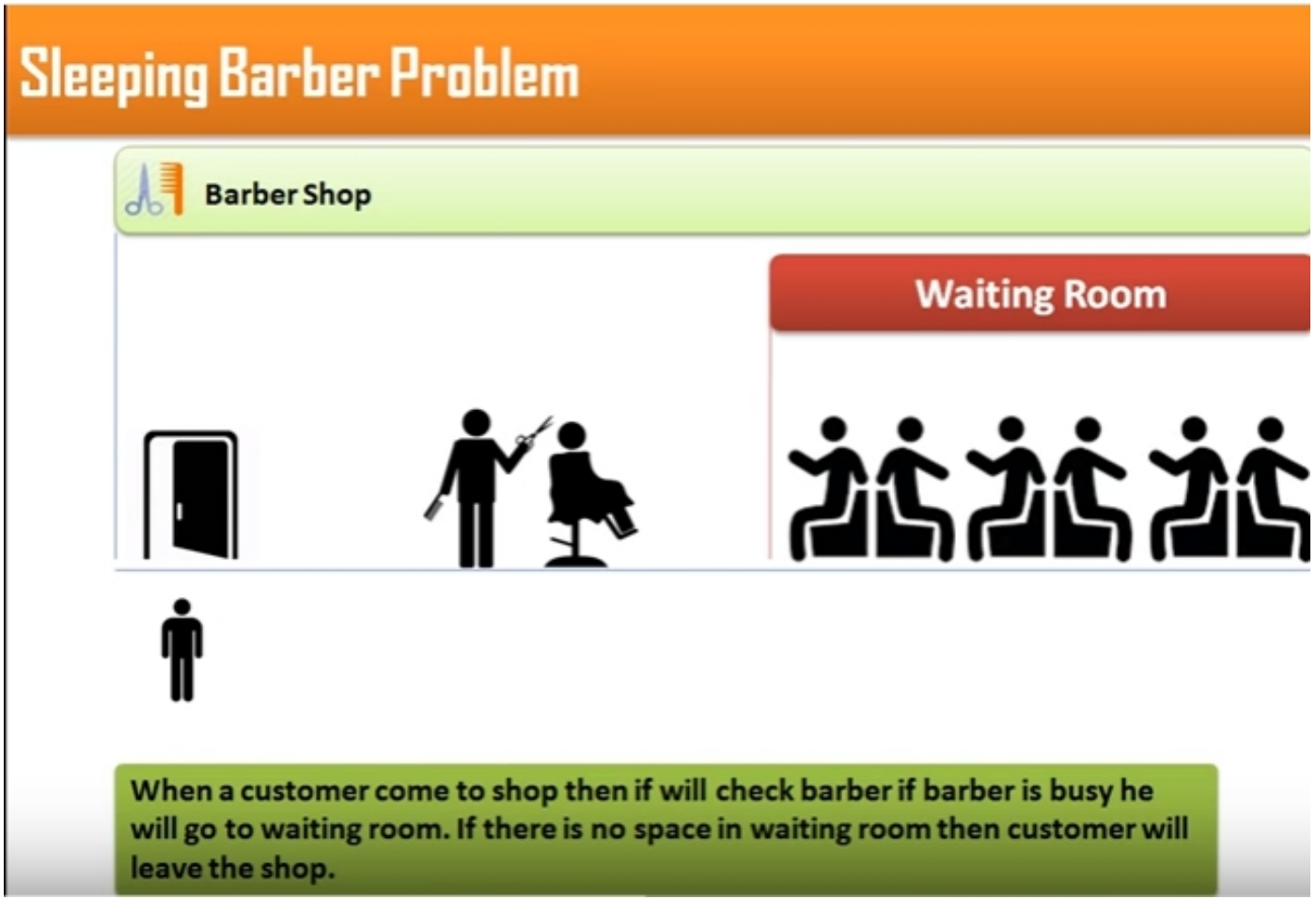
Sleeping Barber Problem

Barber has a chair to cut hair of customer. When barber is done with customer then he will cut hair of next customer from waiting room.

Sleeping Barber Problem



Sleeping Barber Problem



Sleeping Barber Problem - Solution

Sleeping Barber Problem - Solution

BARBER

```
While (true)
{
 wait(custReady);
wait(accessWRSeats);
numberOfSeatWR++;
signal(barberReady);
signal( accessWRSeats);
# Cut Hair here
}
```

CUSTOMER

```
wait(accessWRSeats);
if ( numberOfSeatWR >0)
 {
 numberOfSeatWR--;
 signal(custReady)
 signal(accessWRSeats);
 wait(barberReady);
 }
else
 signal(accessWRSeats);
```

Sleeping Barber Problem - Solution

numberOfSeatsWR = N

barberReady = 0 | accessWRSeats = 1 | custReady = 0

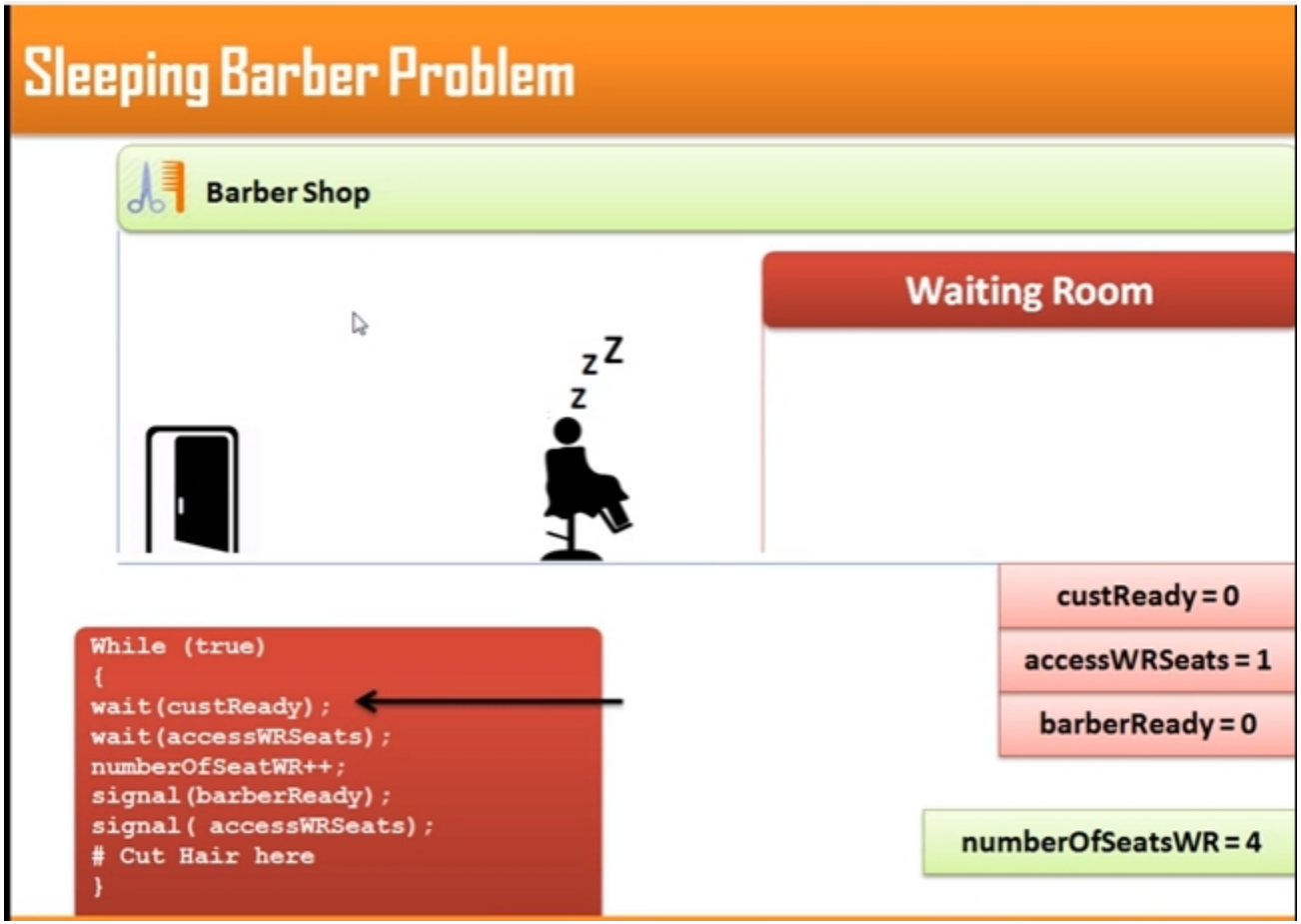
BARBER

```
While (true)
{
 wait(custReady);
wait(accessWRSeats);
numberOfSeatWR++;
signal(barberReady);
signal( accessWRSeats);
# Cut Hair here
}
```

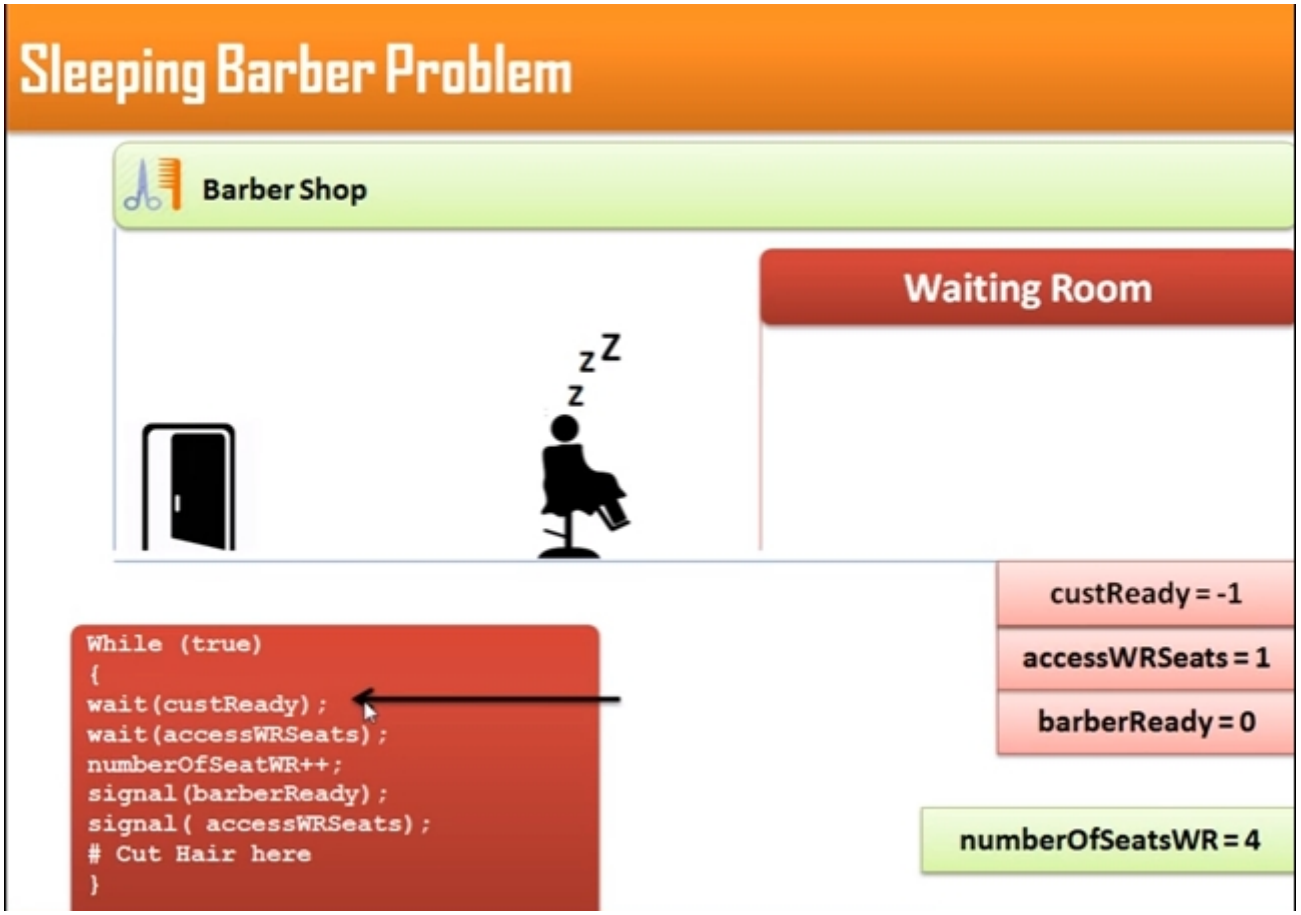
CUSTOMER

```
wait(accessWRSeats);
if ( numberOfSeatWR >0)
 {
 numberOfSeatWR--;
 signal(custReady)
 signal(accessWRSeats);
 wait(barberReady);
 }
else
 signal(accessWRSeats);
```

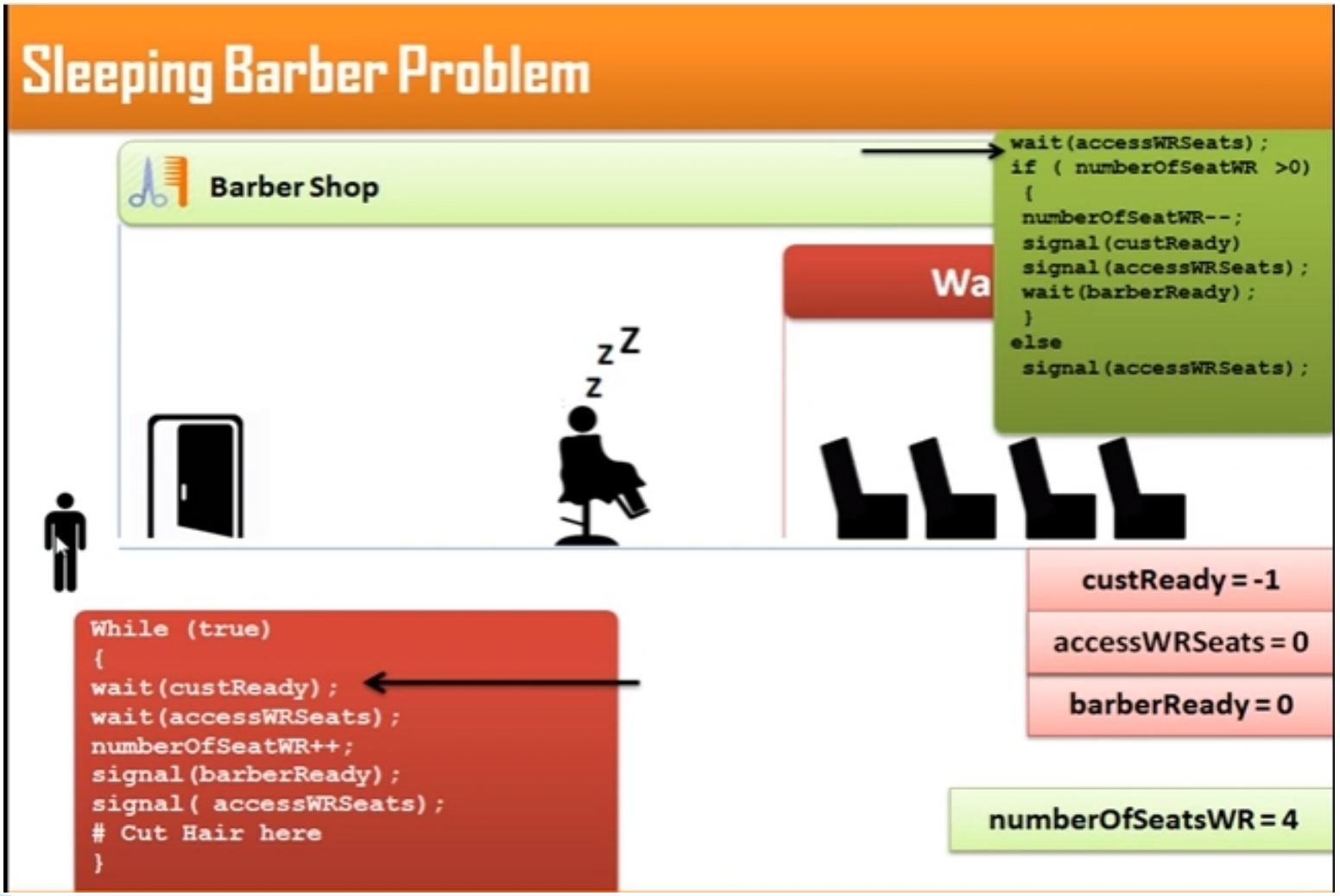
Sleeping Barber Problem



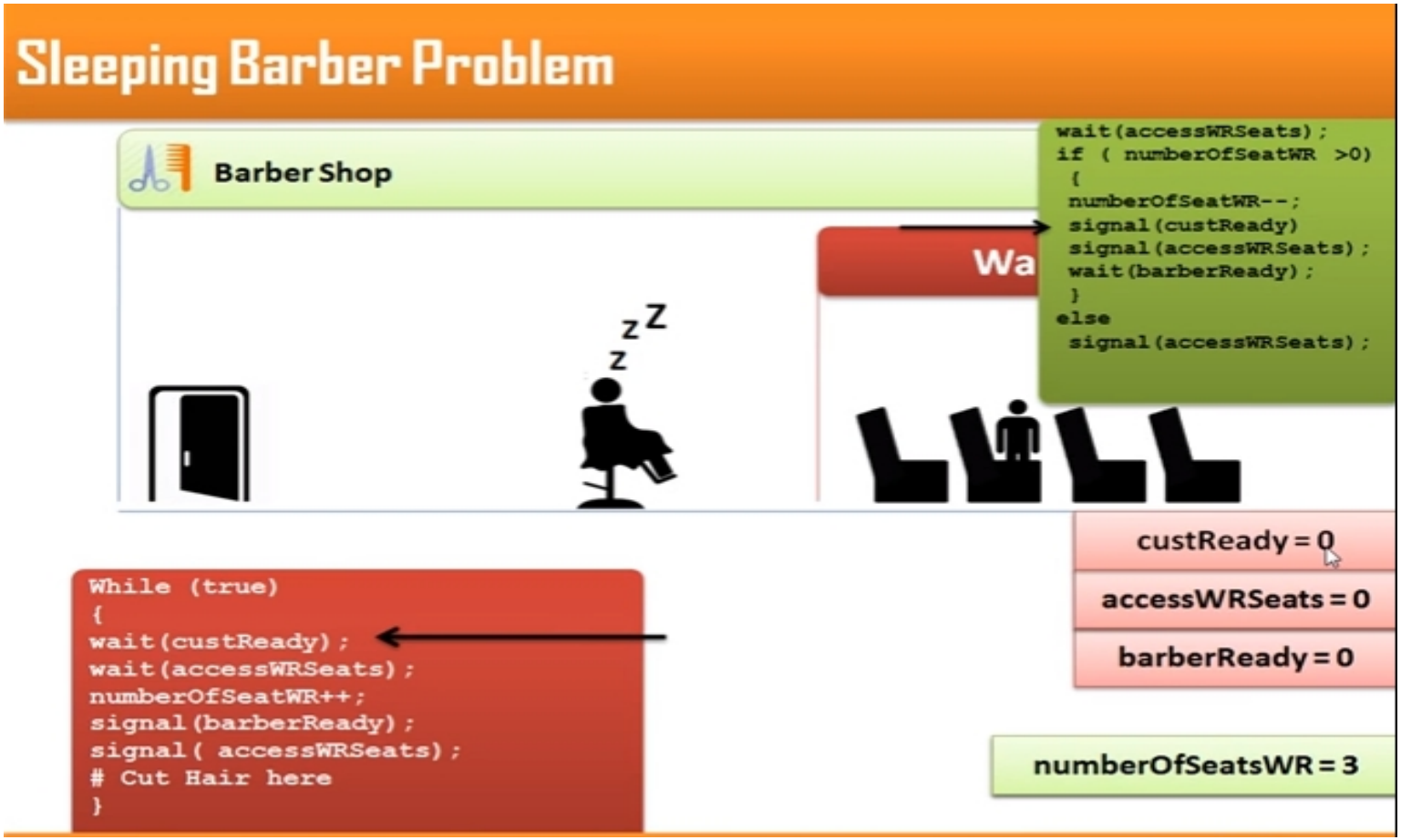
Sleeping Barber Problem



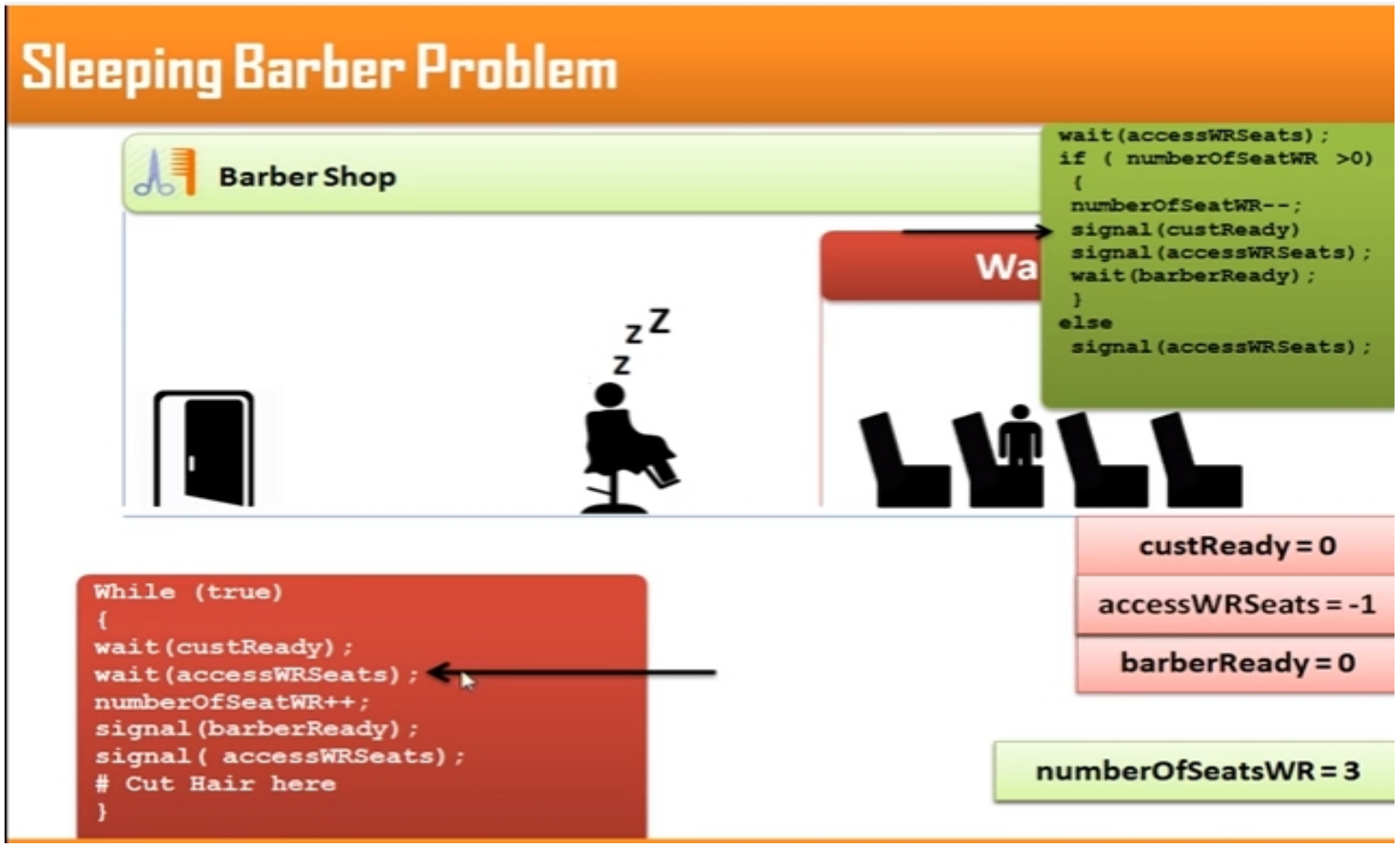
Sleeping Barber Problem

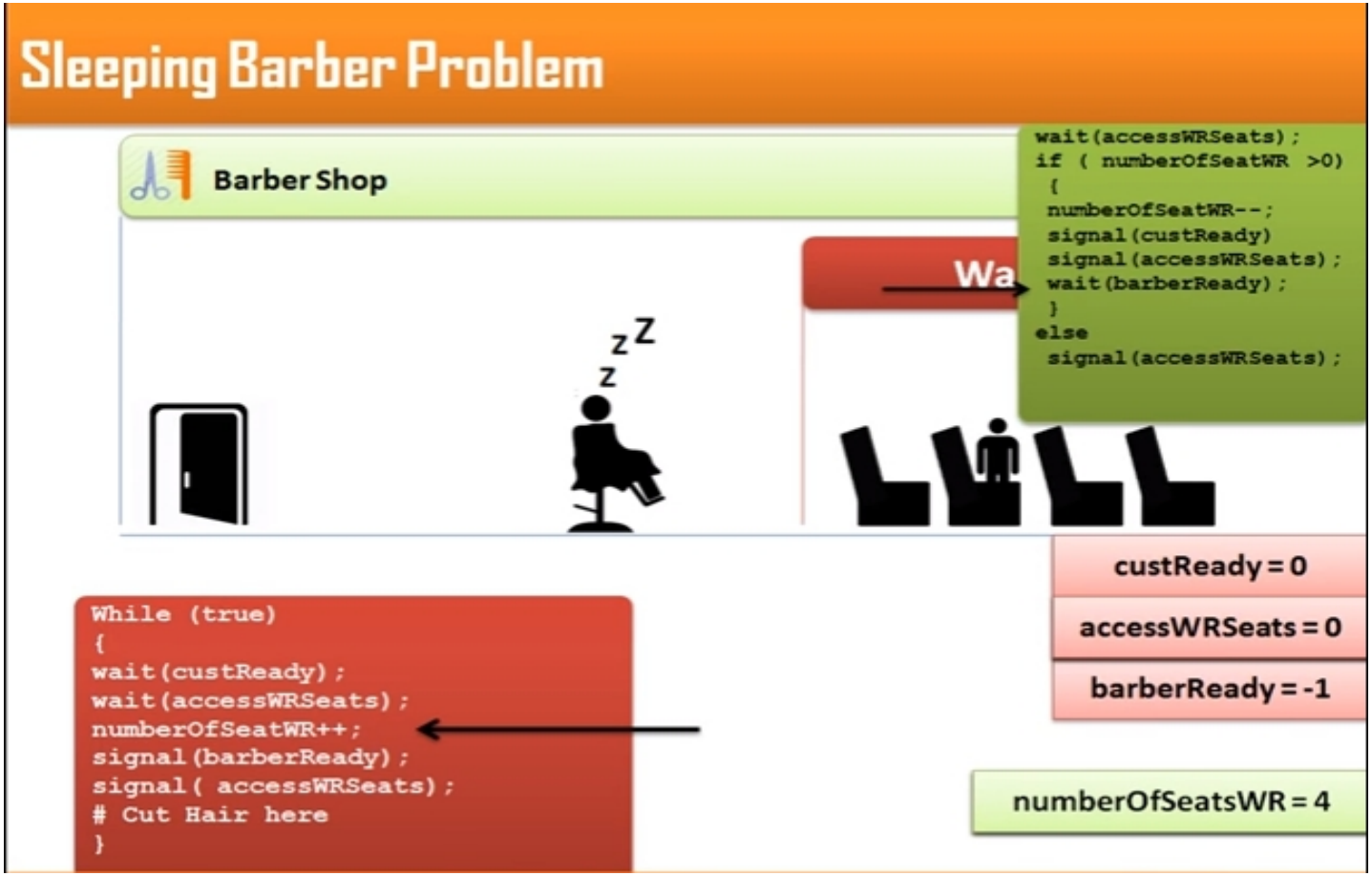


Sleeping Barber Problem

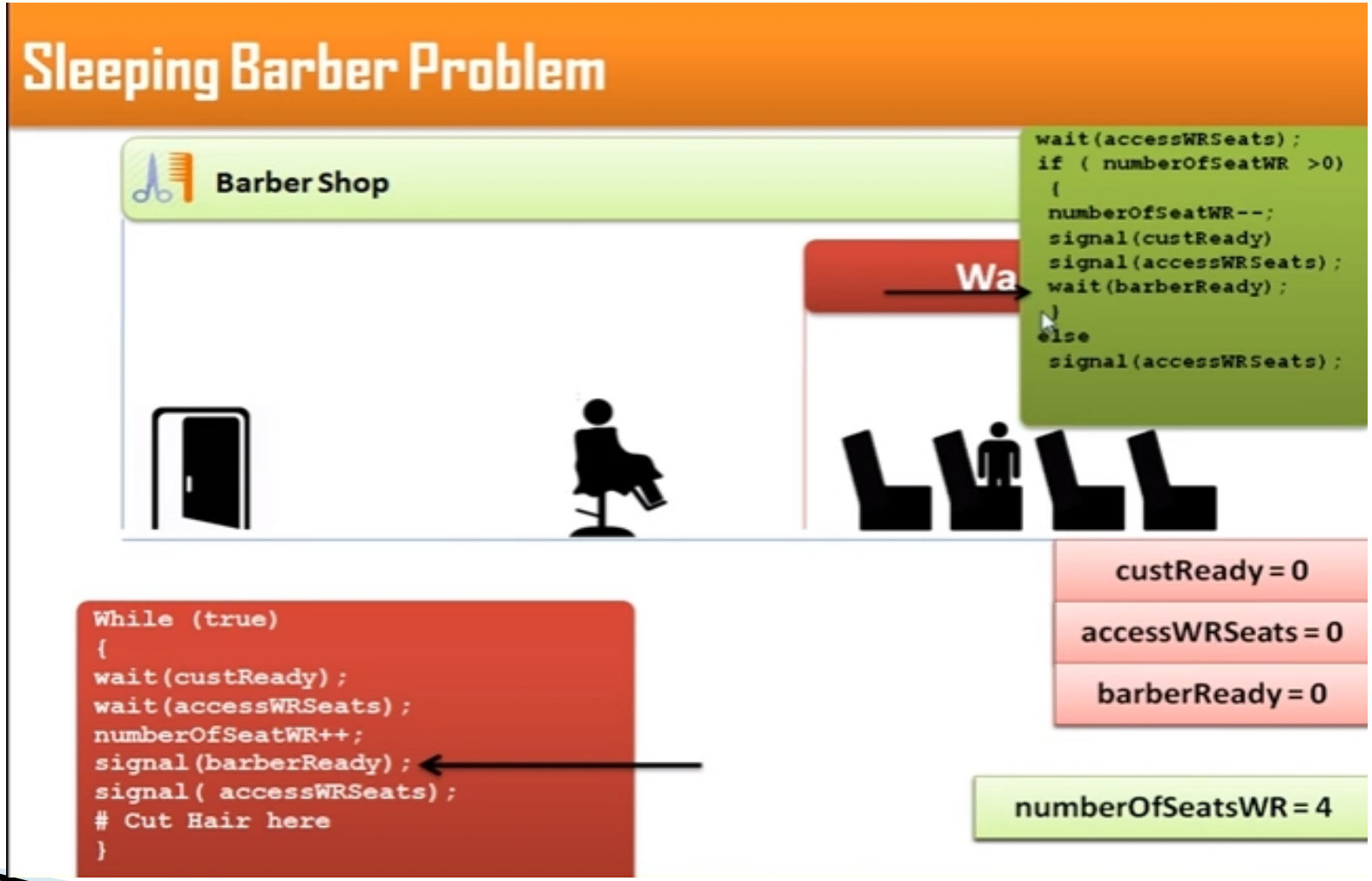


Sleeping Barber Problem

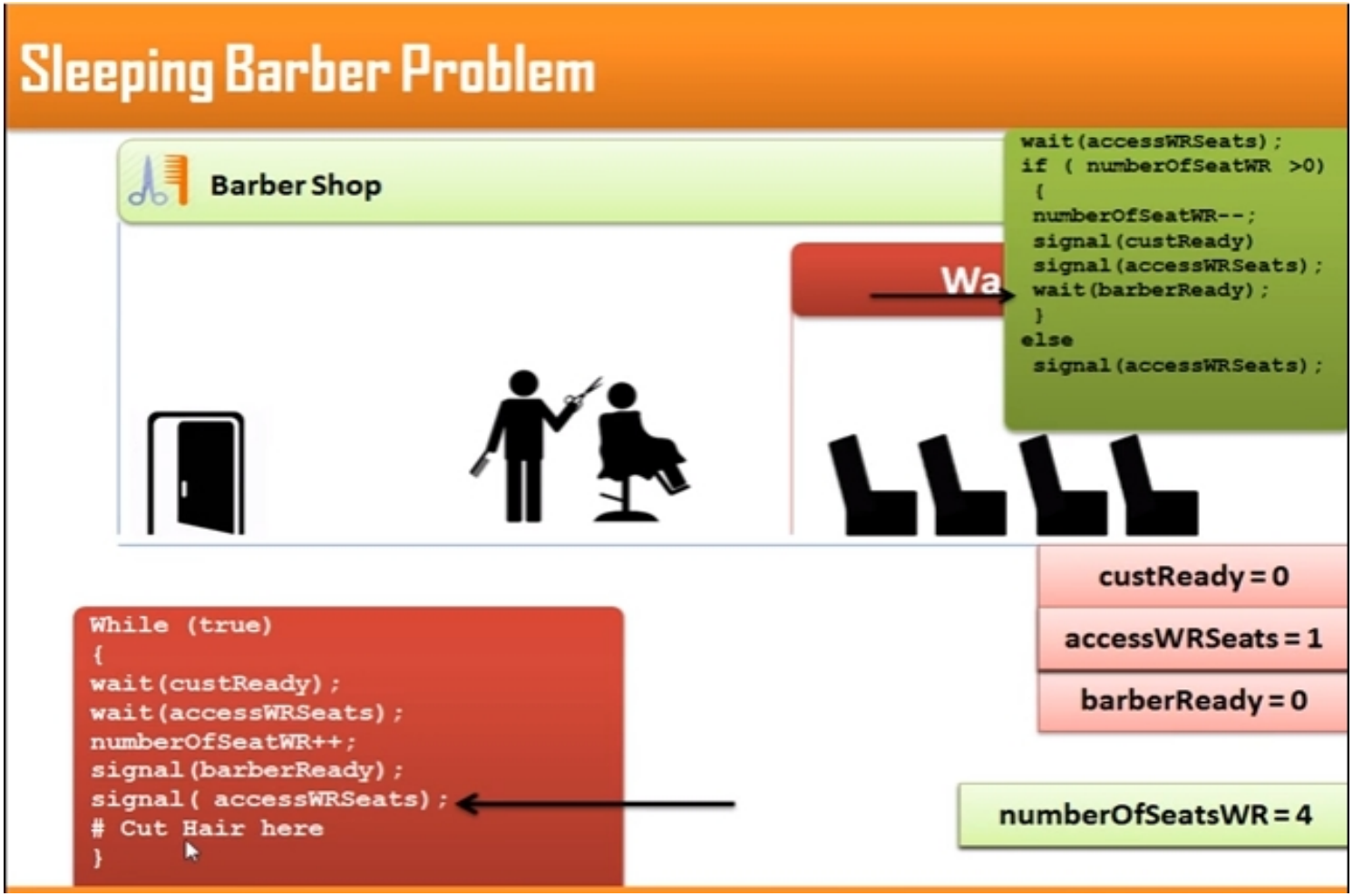



Sleeping Barber Problem
Barber Shop
Wa
z Z z
wait(accessWRSeats);
if (numberOfSeatWR >0)
{
numberOfSeatWR--;
signal(custReady)
signal(accessWRSeats);
wait(barberReady);
}
else
signal(accessWRSeats);
While (true)
{
wait(custReady);
wait(accessWRSeats);
numberOfSeatWR++;
signal(barberReady);
signal(accessWRSeats);
# Cut Hair here
}
custReady = 0
accessWRSeats = 0
barberReady = -1
numberOfSeatsWR = 4

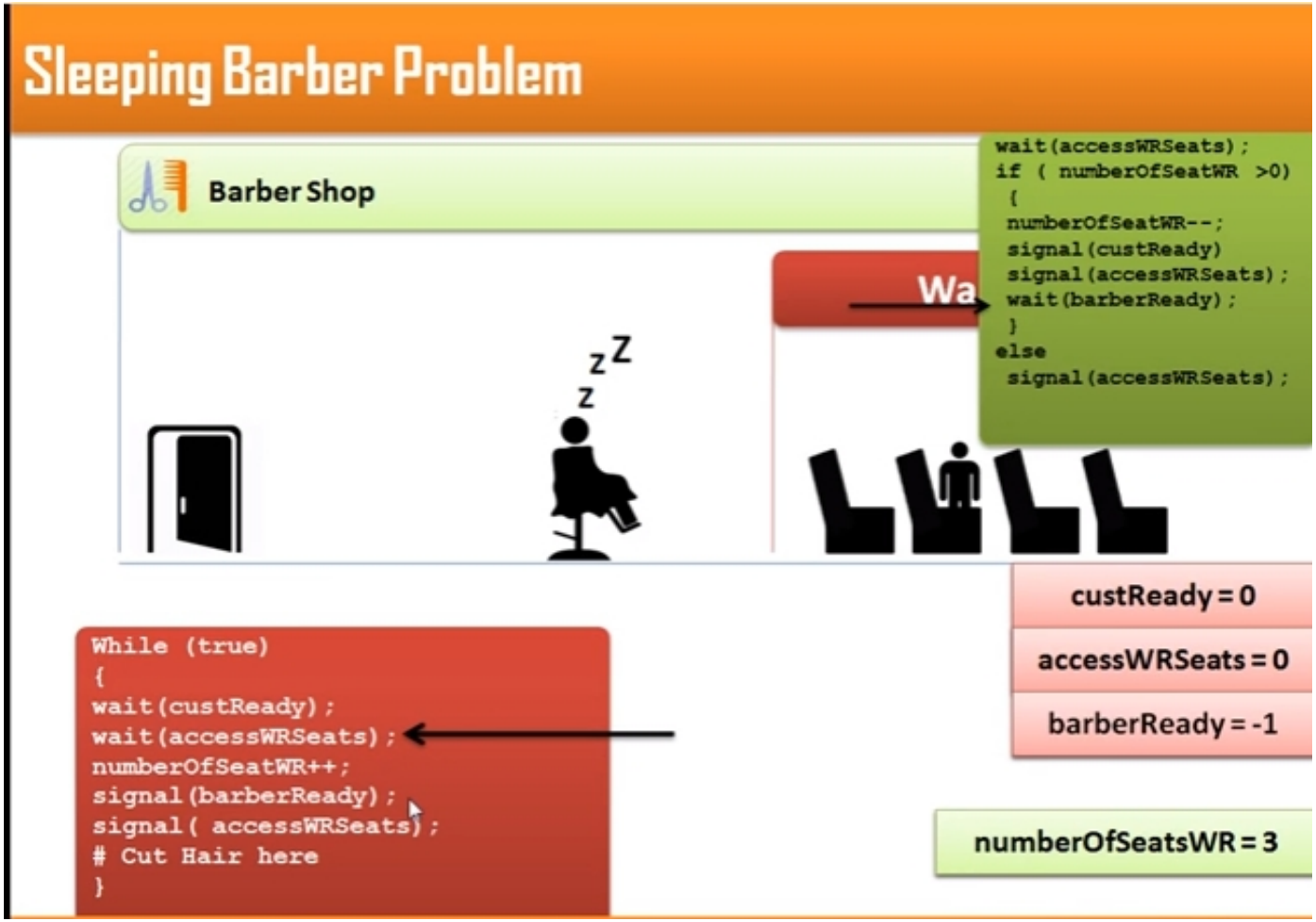
Sleeping Barber Problem



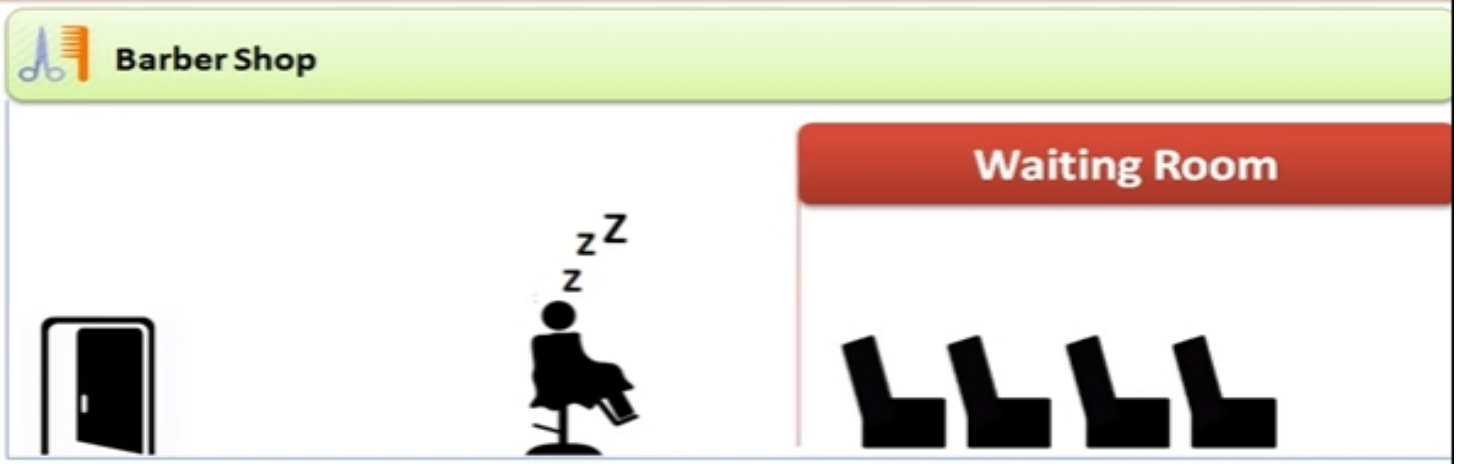
Sleeping Barber Problem



Sleeping Barber Problem



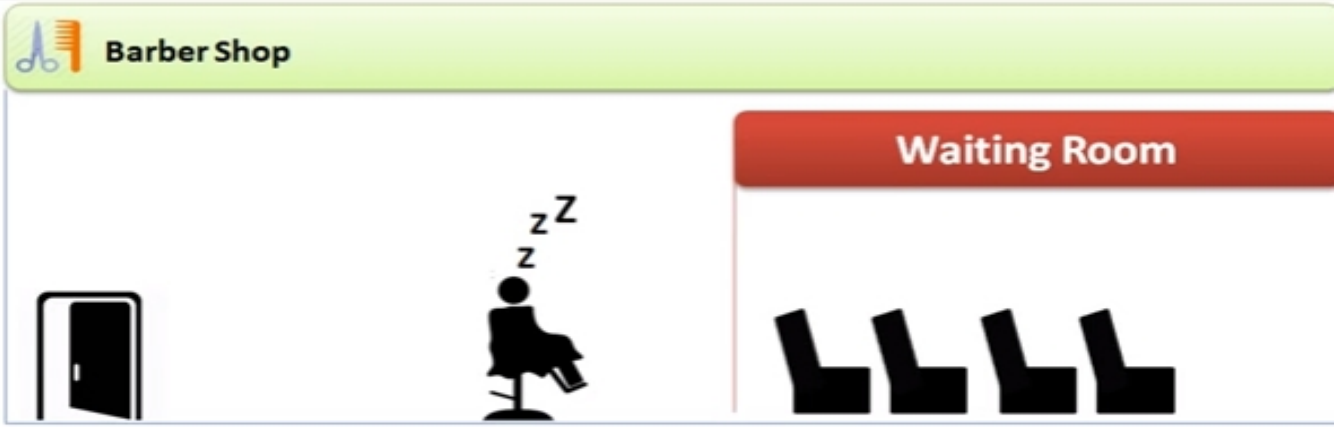
Sleeping Barber Problem



DeadLock Free

Starvation Problem

Sleeping Barber Problem



Starvation Problem

A queue can be used to add customers, FIFO order.

Sleeping Barber's problem

```
Semaphore Customers = 0
Semaphore Barber = 0
Semaphore accessSeats = 1
 int NumberOfFreeSeats = n        //total number of chairs

The Barber Process

while(true) {
P(Customers)                      //get a customer - if none is available - sleep
P(accessSeats)                    //got a customer, free one chair
NumberOfFreeSeats++               //one chair gets free
V(accessSeats)                    //no need to lock the chairs anymore
V(Barber)                         //the barber is ready to cut hair
                                  //cutting hair
}
```

Sleeping Barber's problem

A Customer Process

```
P(accessSeats)                  //the customer tries to get a chair
if ( NumberOfFreeSeats > 0 )
{                               //if there are any free chairs – sit down
NumberOfFreeSeats--
V(accessSeats)                  //no need to lock the chairs anymore
V(Customers)                    //notify the barber
P(Barber)                       //"our turn", … but wait if the barber is busy
                                //here the customer is having his/her hair cut
}
else
{                               //there are no free chairs - tough luck …
V(accessSeats)                  //but don't forget to release the lock on the chairs
                                //customer leaves without a haircut  :o(
}
```

More Problems with Semaphores

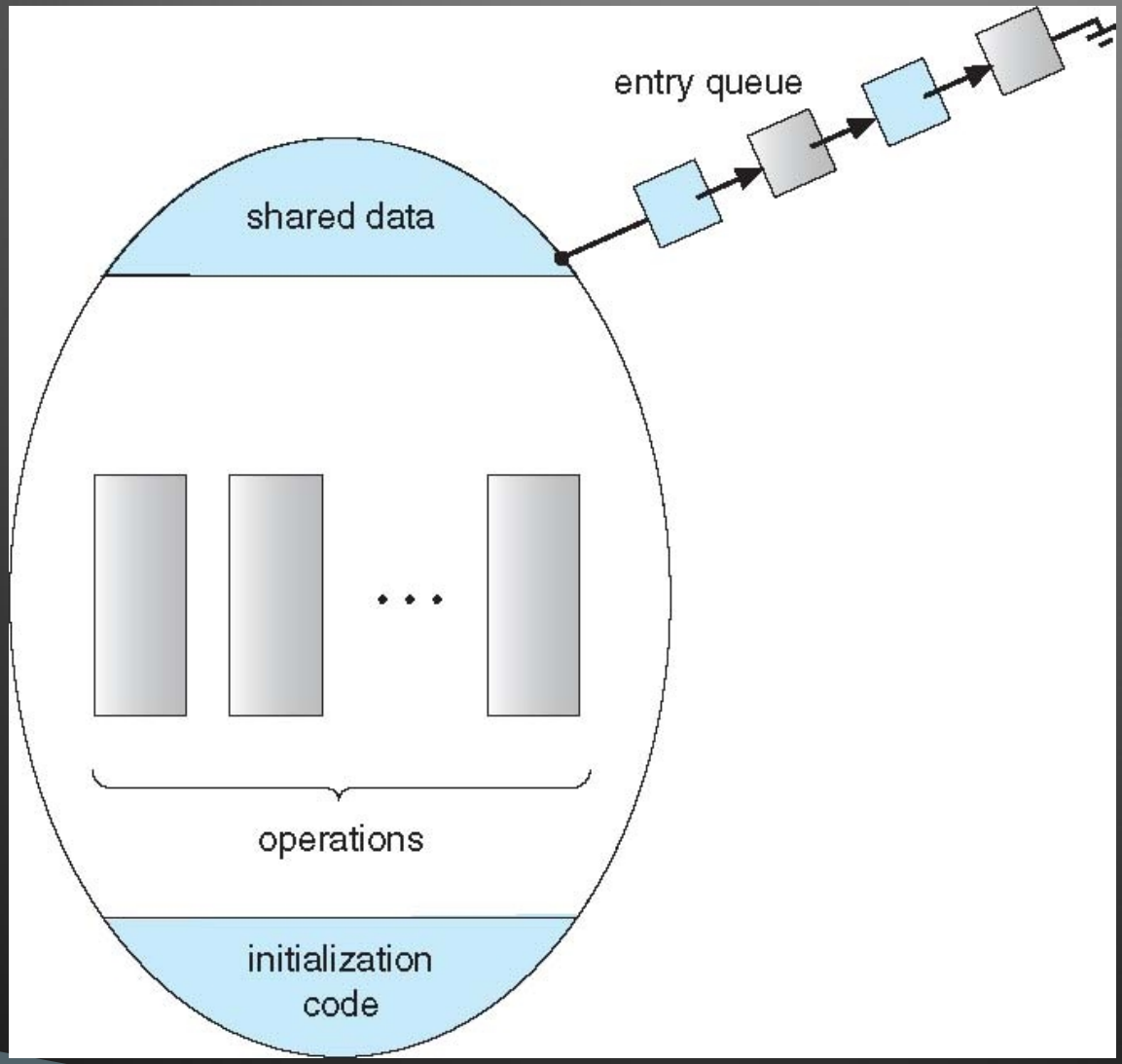
- Relies too much on programmers not making mistakes (accidental or deliberate)
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
   …
  procedure Pn (…) {……}
    Initialization code ( ….) { … }
   …
  }
}
```

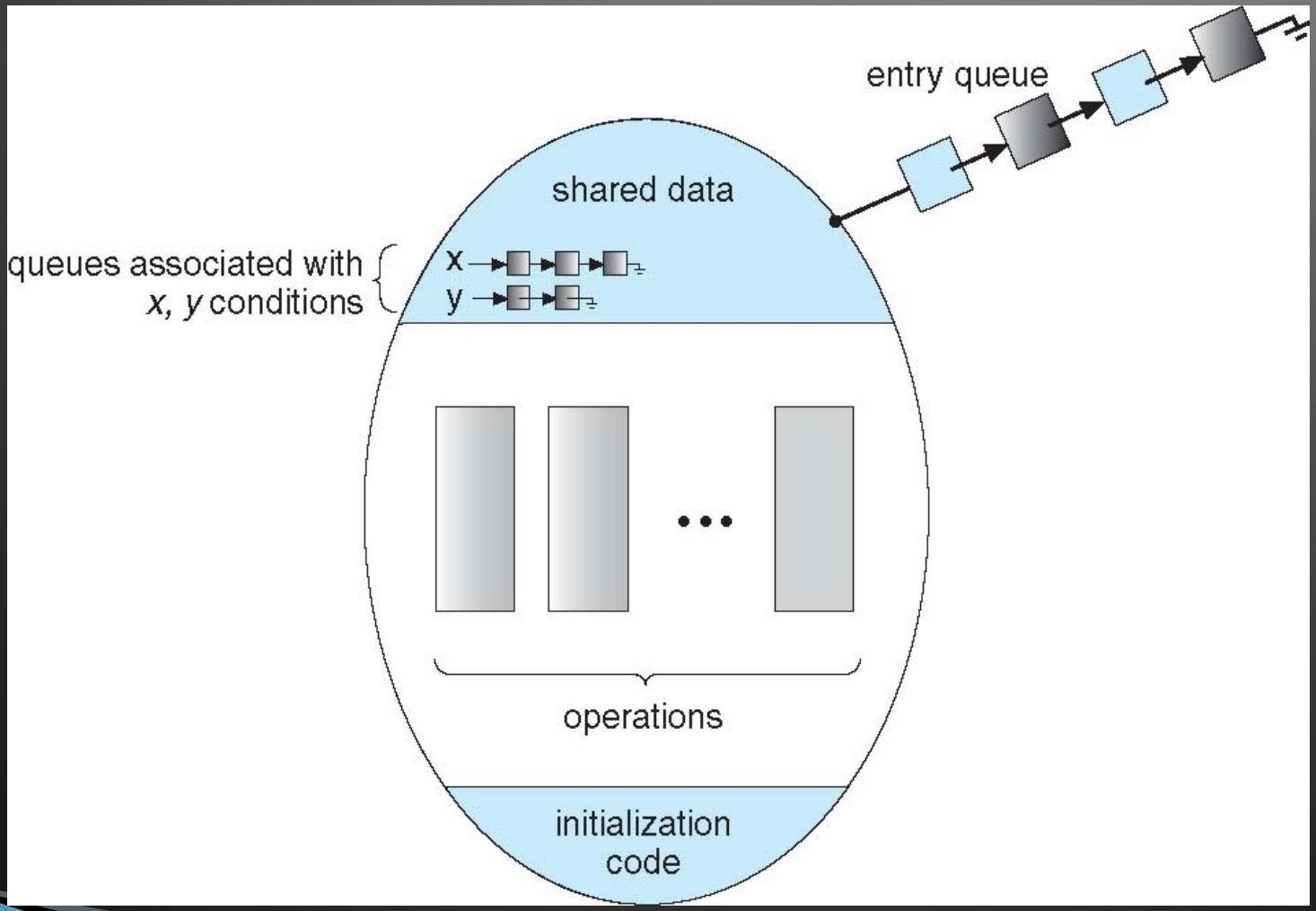
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

```
monitor DP
   {
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self [i].wait;
   }

      void putdown (int i) {
         state[i] = THINKING;
                  // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
       }
```

Solution to Dining Philosophers (Cont.)

```
   void test (int i) {
          if ( (state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
              state[i] = EATING ;
              self[i].signal () ;
           }
    }

      initialization_code() {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
    }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher *I* invokes the operations pickup()
  and putdown() in the following sequence:

```
DiningPhilosophters.pickup (i);

    EAT

DiningPhilosophers.putdown (i);
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    ...
            body of F;

    ...
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially = 0)
int x-count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

Monitor Implementation

- The operation x.signal can be implemented as:

```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
                if (busy)
                     x.wait(time);
                busy = TRUE;
    }
    void release() {
                busy = FALSE;
                x.signal();
    }
initialization code() {
     busy = FALSE;
    }
}
```

End of Process Synchronization