OPERATING SYSTEMS (THEORY)

LECTURE - 7

K.ARIVUSELVAN

Assistant Professor (Senior) – (SITE)

VIT University

PROCESS SYNCHRONIZATION

Introduction to Cooperating Processes

- Processes within a system may be independent or cooperating

- Independent process cannot affect or be affected by the execution of another process

- Cooperating process can affect or be affected by other processes, including sharing data

Race Condition:

- Several processes access and manipulate the shared data concurrently

- The outcome of the execution depends on the particular order in which the access takes place

time	Person A	Person B
8:00	Look in fridge. *Out of milk*	
8:05	Leave for store.	
8:10	Arrive at store.	Look in fridge. *Out of milk*
8:15	Buy milk.	Leave for store.
8:20	Leave the store.	Arrive at store.
8:25	Arrive home, put milk away.	Buy milk.
8;30		Leave the store.
8:35		Arrive home, *OH! OH!*

Someone gets milk, but NOT everyone (*too much milk!*)

Race Condition: Solution

- ❑ To prevent race conditions, concurrent processes must be synchronized

- ❑ Ensure that only one process at a time can be manipulating the shared Data

The Critical-Section

- A section of code Common to n cooperating processes, in which the processes may be accessing common variables

A critical section environment contains:

1. **Entry Section:** Code requesting entry into the critical section

2. **Critical Section:** Code in which only one process can execute at any one time

3. **Exit Section:** The end of the critical section, releasing or allowing others in

4. **Remainder Section:** Rest of the code AFTER the critical section

General Structure of a Typical Process

```
do {
      entry section

            critical section

      exit session

            remainder section

} while (TRUE);
```

Solution to Critical-Section Problem

The critical section must ENFORCE all the 3 rules:

(1) Mutual Exclusion

(2) Progress

(3) Bounded Waiting

Solution to Critical-Section Problem

1. Mutual Exclusion – If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

(i.e. no two processes will simultaneously be inside the same CS)

2. Progress – the selection of the process 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.

+

- **processes will remain inside its CS for a short time only, without blocking**

Initial Attempts to Solve Problem

- Only 2 processes, *P*0 and *P*1

General structure of process *Pi* (other process *Pj*)

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- **Shared variables:**

int turn;
initially **turn = 0 (or) 1**

- **Structure of Process *Pi* :**

turn = j (*Pj* can enter its critical section)

repeat
    while (turn != i) do nothing; /*busy wait*/
    critical section
    turn = i;
    remainder section
Until false**;**

Algorithm 1

- This solution guarantees **mutual exclusion**

Drawback 1: processes must **strictly alternate**

Drawback 2: if one processes fails other process is **permanently blocked**

Algorithm 2

Shared variables:

boolean flag[2];

initially **flag [0] = flag [1] =** **false**

flag [i] = true ⇒ *Pi* ready to enter its critical section

Structure of Process *Pi* :

repeat

flag[i] := true;

while (flag[j]) do nothing;

critical section

flag [j] = false;

remainder section

Until false**;**

Algorithm 2

- This solution guarantees mutual exclusion

Drawback 1: This approach may lead to dead lock

- What is wrong with this implementation ?

- Dead lock occurs because each process can insist on its right to enter critical section

Algorithm 3 (PETERSON ALGORITHM)

- **Combined shared variables of algorithms 1 and 2**

Process *Pi*

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn = j) do nothing;
        critical section
        flag [i] = false;
        remainder section
} while (1);
```

- **Meets all three requirements; solves the critical-section problem for two processes**

Bakery Algorithm

Critical section for n processes:

- Before entering its critical section, process receives a number.

- Holder of the smallest number enters the critical section.

- If processes *Pi* and *Pj* receive the same number, if *i* < *j*, then *Pi* is served first; else *Pj* is served first.

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,4,5...

Notation => lexicographical order (**ticket #**, **process id #**)

❑ Shared data:

```
boolean choosing[n];

int number[n];
```

- **Data structures are initialized to false and 0 respectively**

- **choosing array =>indicate that a process wants to enter it's critical section**

- **number array => contains the numbers associated with each process.**

Bakery Algorithm

```
Repeat {

    choosing[i] = true;
    number[i] = max(number[0], number[1], …, number [n – 1])+1;
    choosing[i] = false;

    for (j = 0 to n-1) do begin
    {
    while (choosing[j]) do no-op ;
    while ((number[j] != 0) && ((number[ j ], j ) < (number[ i ], i ) ))
    do no-op ;
    }

    critical section

    number[i] = 0;

    remainder section

} until false;
```

Semaphores

- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
- Can only be accessed via two indivisible (atomic) operations

wait (*S*):
S --;

signal (*S*):
S ++;

- **Wait** operation **decrements** the semaphore value
- **Signal** operation **increments** semaphore value

Semaphores

- If a process is waiting for a signal, it is suspended until that signal is sent
- Queue is used to hold processes waiting on the semaphore

Critical Section of *N* Processes

Shared data:

semaphore mutex; //initially ***mutex* = 1**

Process *Pi:*

```
do {
        wait(mutex);

        critical section

        signal(mutex);

        remainder section

    } while (1);
```

Implementation

- Semaphore operations are now defined as:

```
wait(mutex):

        mutex.value--;
        if (mutex.value < 0)
        {
            add this process to S.L;
            block;
        }
```

Implementation

```
signal(mutex):

        mutex.value++;
        if (mutex.value <= 0)
        {
            remove a process P from S.L;
            wakeup(P);
        }
```

- ❑ Value of semaphore can be **negative** and represents the **number of processes waiting** on it

Semaphore As a General Synchronization Tool

- Execute **B** in **Pj** only after **A** executed in **Pi**
- Use semaphore *flag* initialized to **0**

- **Code:**

```
Pi                  Pj
 :                   :
A                   wait(flag)
signal(flag)        B
```

Semaphores can be used to control the order in which concurrent processes run. In this example the following conditions must be met:-

*Process A must run first
*Process D can not run until after Process C is finished
*Process E must not run until all other processes are finished

Note that semA is a general semaphore that is used to control the start of both Processes B and C.

LEGEND


a semaphore which will block a process


a semaphore which will allow a process to continue

Start

Two Types of Semaphores

- ❑ **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**

Classical Problems of Synchronization

- ❑ Bounded-Buffer Problem

- ❑ Dining-Philosophers Problem

- ❑ Readers-Writers Problem

(1) Bounded-buffer Problem

Shared data:

semaphore full, empty, mutex ;

Initially:

full = 0, empty = n, mutex = 1

- **Buffer size is n**
- **Mutex provides exclusive access to the buffer**
- **Consumers wait on full**
- **Producers wait on empty**

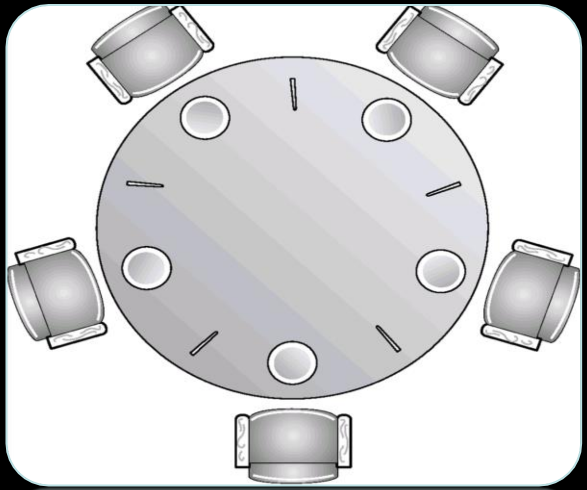
Bounded-buffer Problem: Producer Process

```
do {
    …
    produce an item in nextp
     …
    wait(empty);
    wait(mutex);
     …
    add nextp to buffer
    …
    signal(mutex);
    signal(full);
} while (1);
```

Bounded-buffer Problem: Consumer Process

```
do {
        wait(full)
        wait(mutex);

                    ...
        remove an item from buffer to nextc
                    ...

        signal(mutex);
        signal(empty);

                    ...
        consume the item in nextc
                    ...

} while (1);
```

(2) Dining-philosophers Problem



- ❑ To start eating, a philosopher needs two chopsticks

- ❑ After eating, the philosopher releases both the chopsticks

Shared data:

```
semaphore chopstick[5];
(Initially all values are 1)
```

Philosopher i :

```
do {
    wait(chopstick[i])
    wait(chopstick[(i+1) % 5])
    …
    eat
    …
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    …
    think
    …
} while (1);
```

- The solution is **not deadlock free!!**

  => All philosophers pick up left chopsticks!!

Solution:

- **Allow at most 4 philosophers to be sitting on the table**

- **Allow a philosopher to pick chopsticks only if both are available**

- **An *odd* philosopher picks up first the left and then the right chopstick while a *even* philosopher does the reverse**

(3) Readers-Writers Problem

- Two readers can access the shared data item simultaneously

- A writer requires exclusive access

- No reader should wait unless a writer is already in critical section

Shared data:

var *mutex*, *wrt* : *semaphore* (=1);
readcount : *integer* (=0);

Readers-Writers Problem

- *wrt* is common to both readers and writers.
- It functions as mutual exclusion semaphore for writers.
- It is also used by first and last reader that enters or exits the CS.

- *readcount* keeps track of how many readers are currently accessing the object.

- *mutex* provides mutual exclusion for updating *readcount*.

Readers-Writers Problem

Writer process

```
wait(wrt);
        …
        writing is performed
        …
signal(wrt);
```

Readers-Writers Problem

Reader process

```
        wait(mutex);
            readcount := readcount +1;
            if readcount = 1 then wait(wrt);
        signal(mutex);
                    …
            reading is performed
                    …
        wait(mutex);
            readcount := readcount – 1;
        if readcount = 0 then signal(wrt);
        signal(mutex):
```

Arivuselvan.K

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
type monitor-name = monitor
        variable declarations
        procedure entry P1 :(…);
                begin … end;
        procedure entry P2(…);
                begin … end;
                    ⋮
        procedure entry Pn (…);
                begin…end;
        begin
                initialization code
        end
```

Monitors

A monitor is similar to a C++ class that ties the data, operations, and in particular, the synchronization operations all together,

particular, the synchronization operations all together,

Unlike classes,

monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor method at a time.

monitor method at a time.

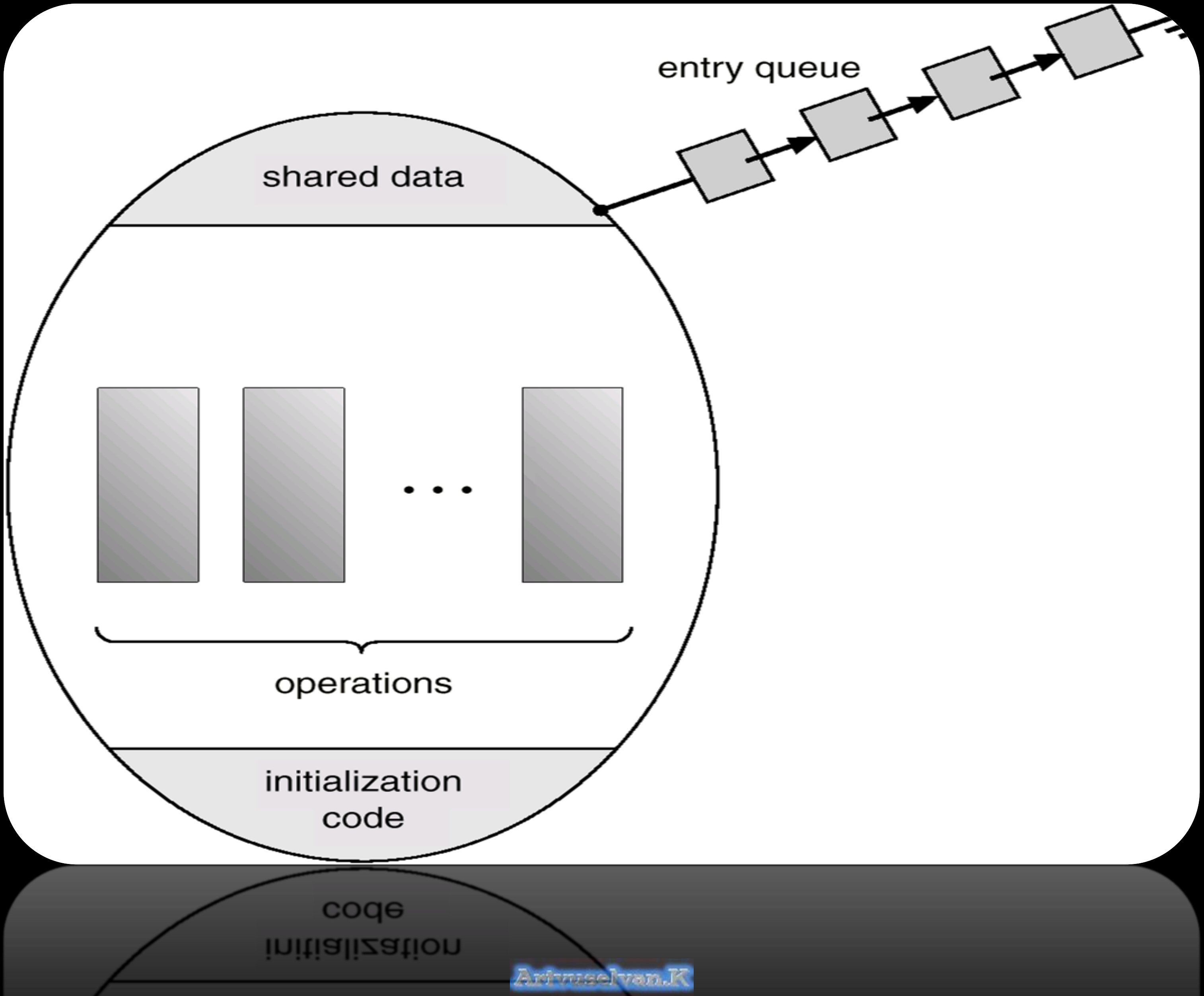
Monitors

A Monitor defines a *lock* and zero or more *condition variables* for managing concurrent access to shared data.

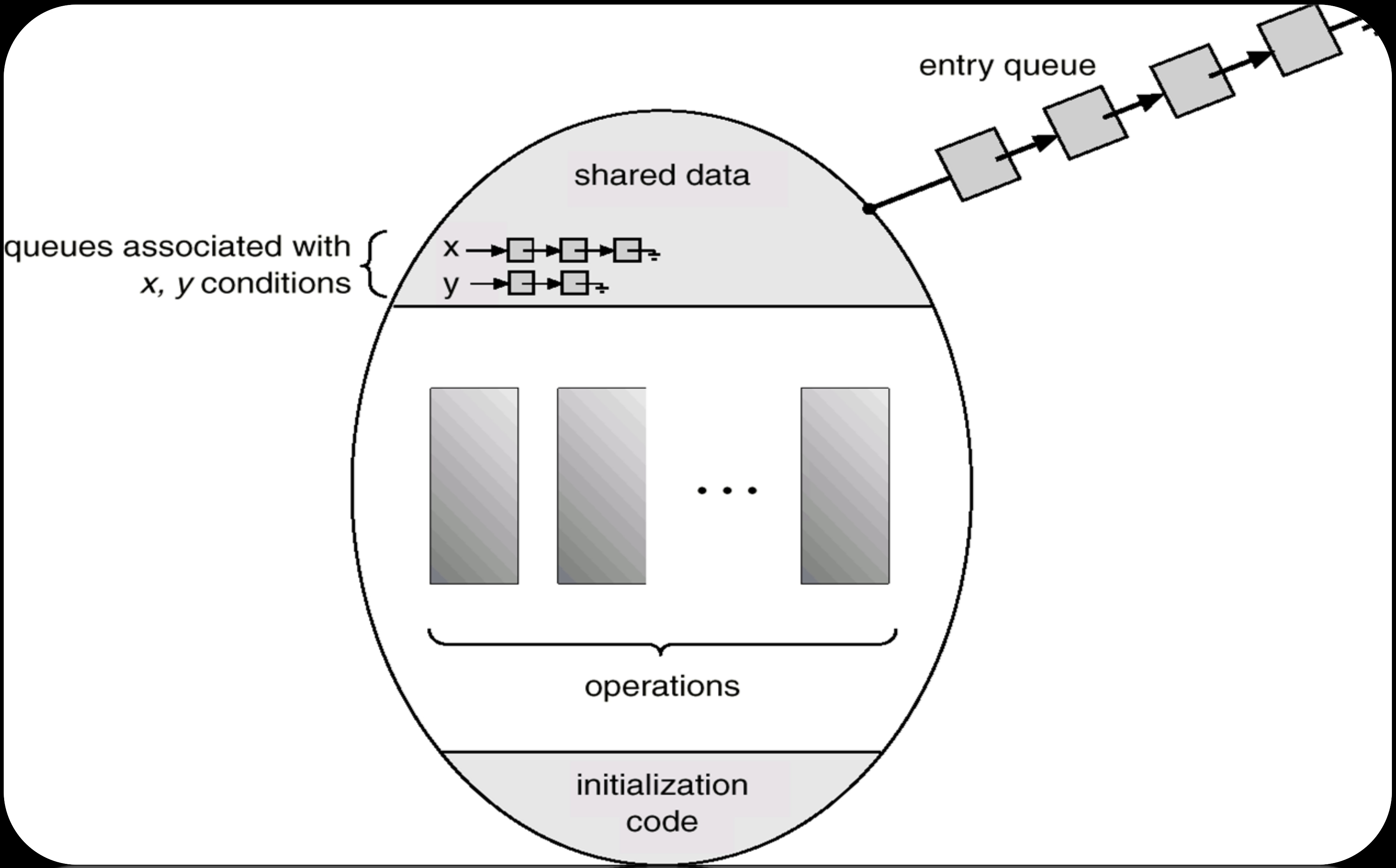
The monitor uses the *lock* to insure that only a single thread is active in the monitor at any instance.

Condition variables enable threads to go to sleep inside of critical sections, by releasing their lock at the same time it puts the thread to sleep.

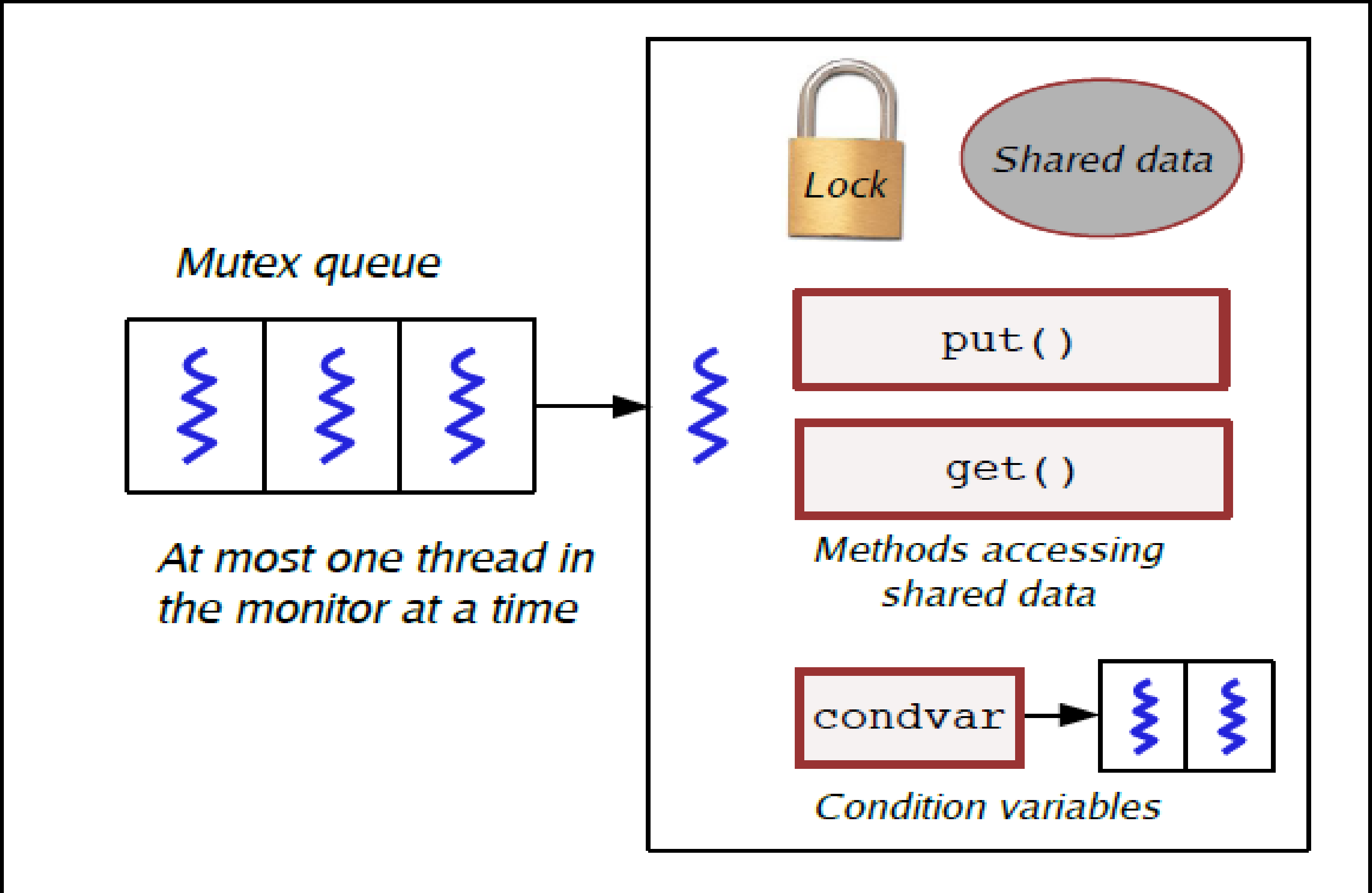
Schematic view of a monitor



Monitor with condition variables



Monitor Example



Operations on Condition Variables

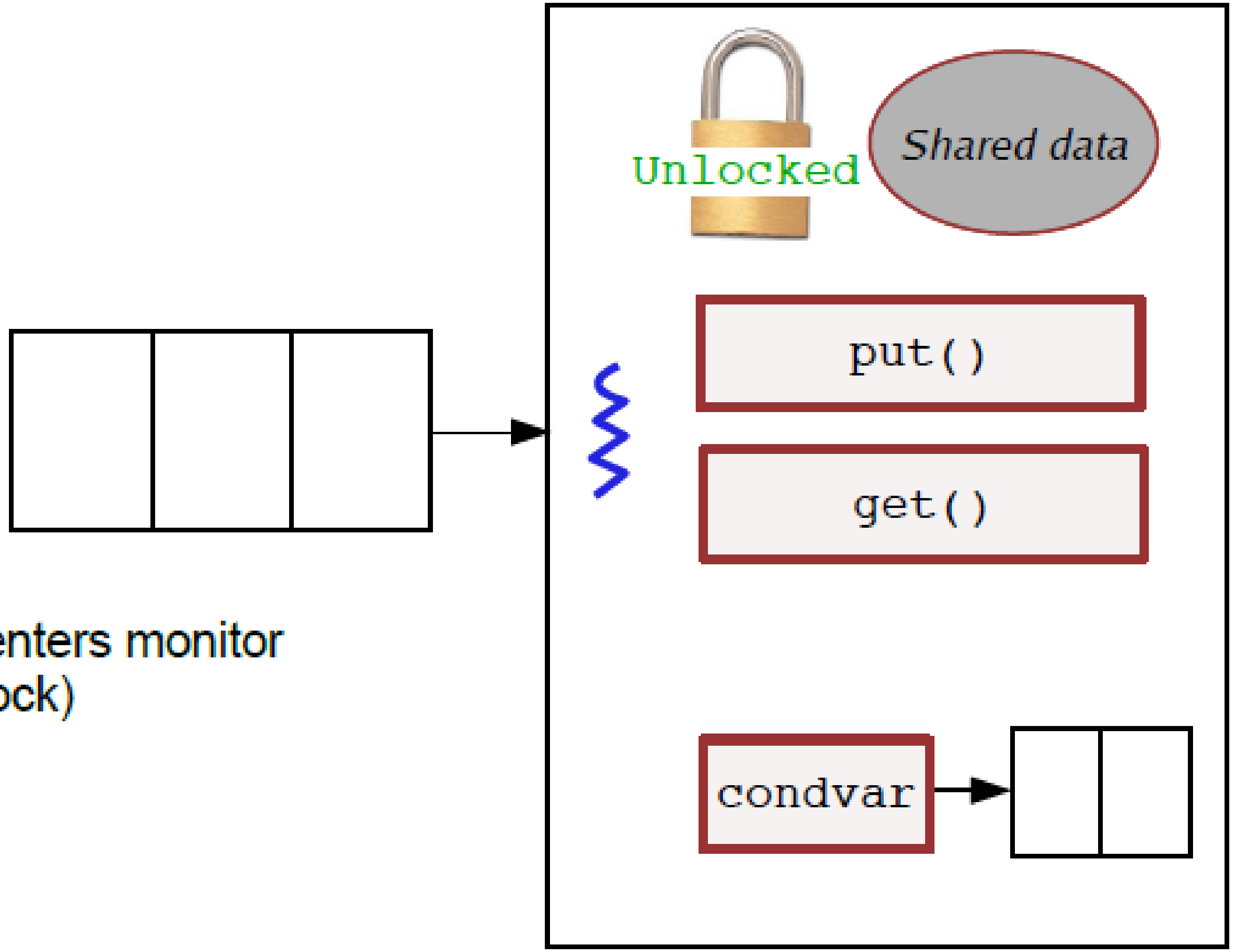
Condition variable: is a queue of threads waiting for something inside a critical section.

Condition variables support three operations:

Wait(Lock lock): atomic (release lock, go to sleep), when the process wakes up it re-acquires lock.
Signal(): wake up waiting thread, if one exists. Otherwise, it does nothing.
Broadcast(): wake up all waiting threads

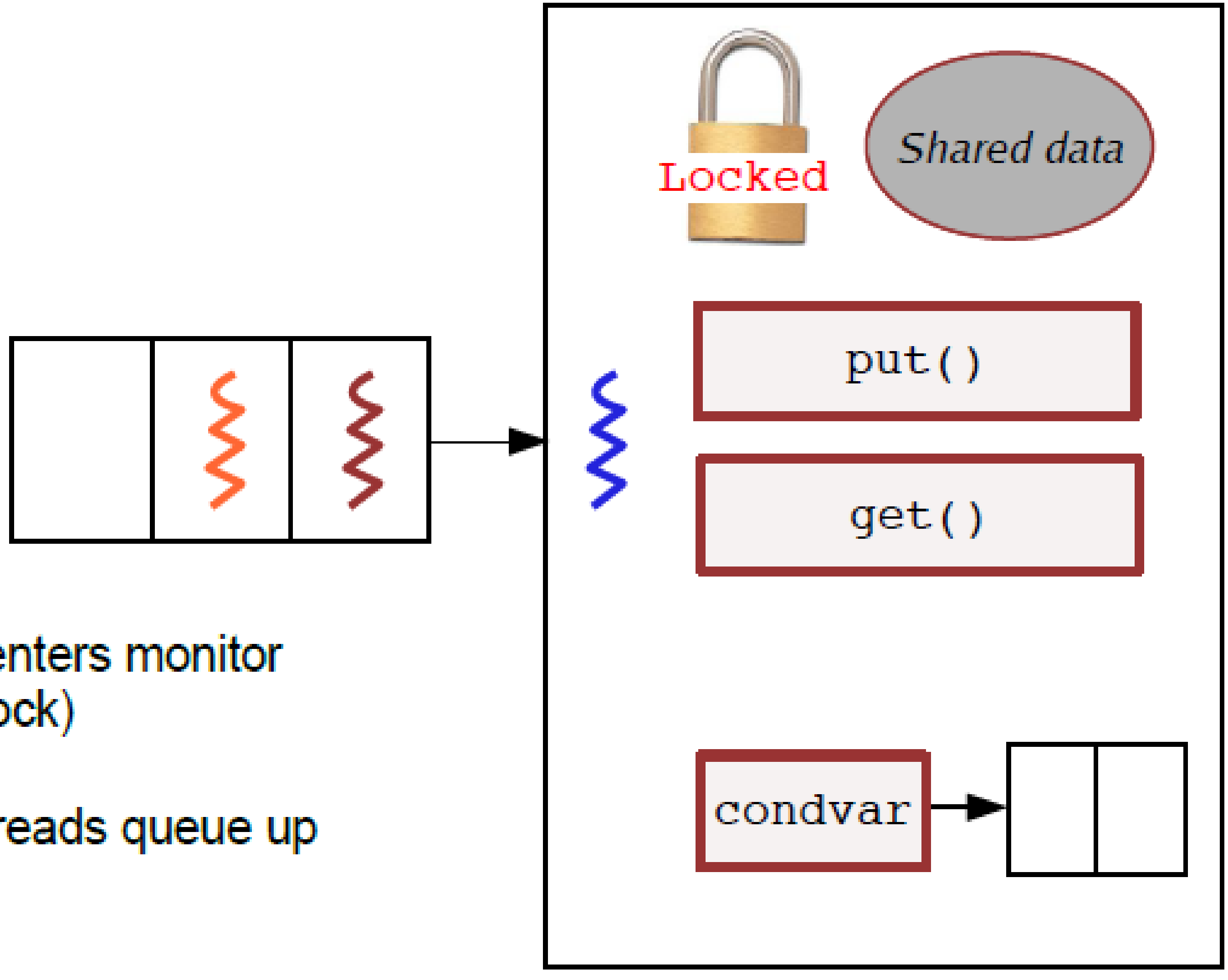
Broadcast(): wake up all waiting threads

Monitors



1) Thread enters monitor
(grabs lock)

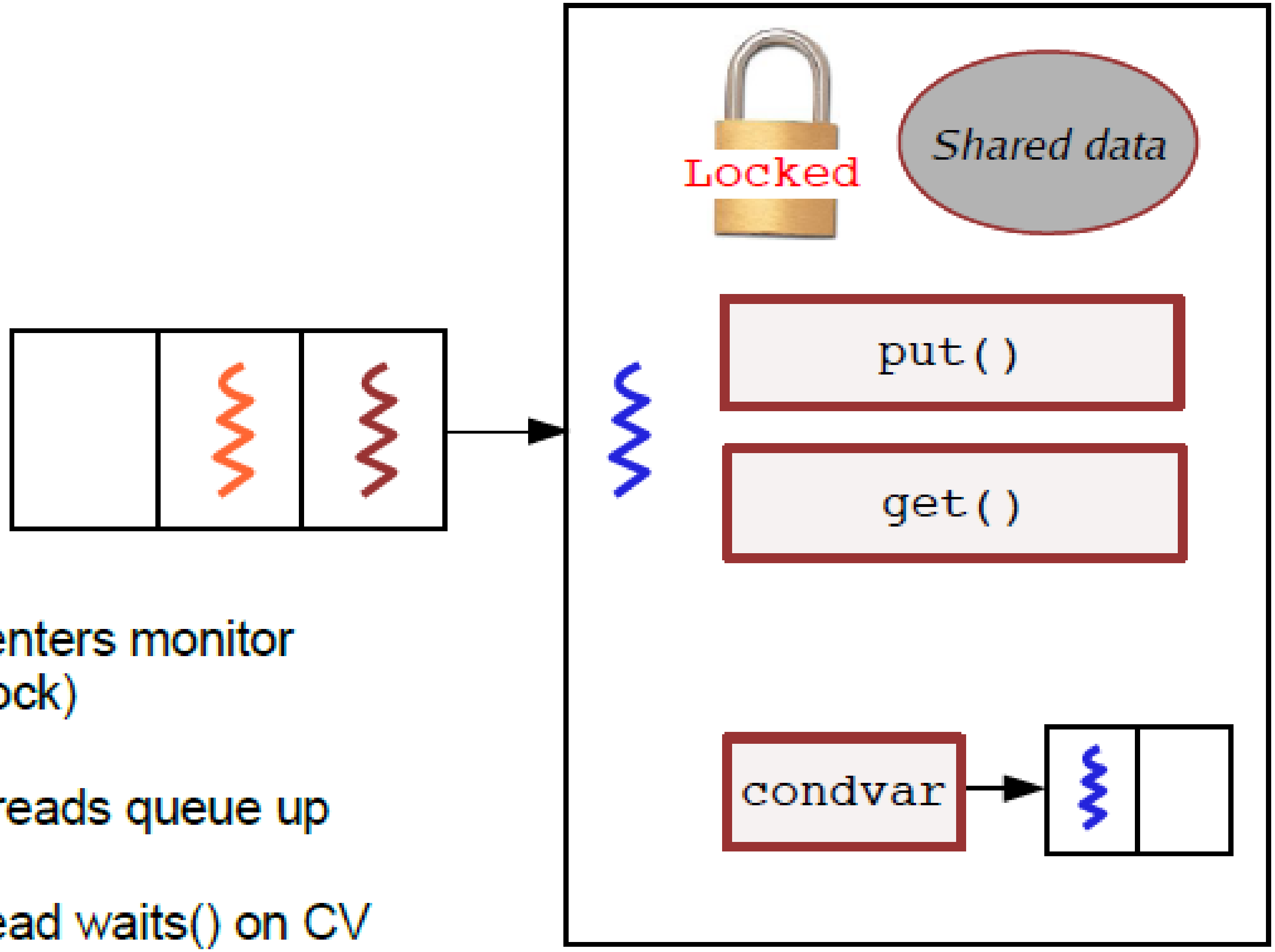
Monitors



1) Thread enters monitor
(grabs lock)

2) Other threads queue up

Monitors

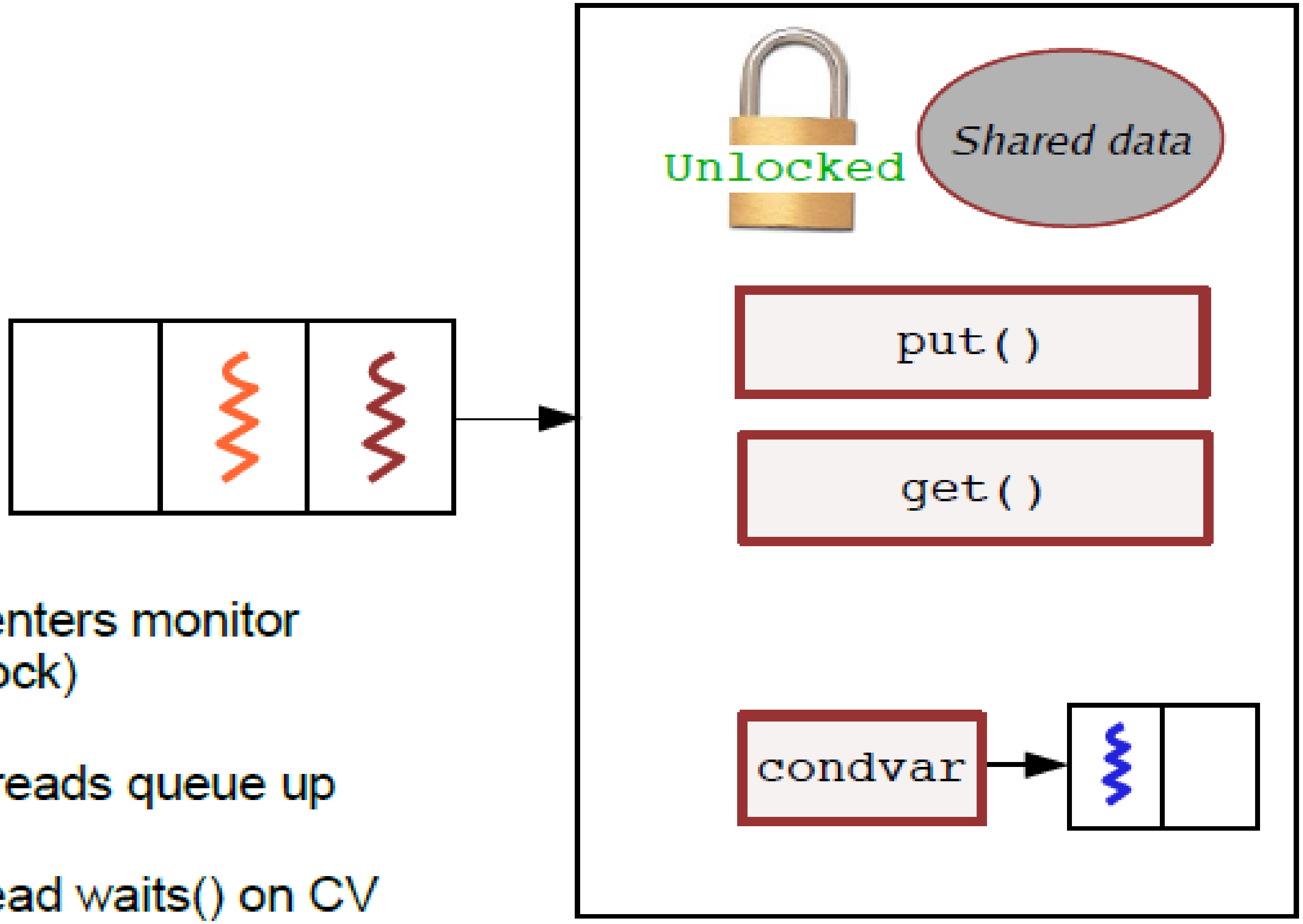


1) Thread enters monitor
(grabs lock)

2) Other threads queue up

3) Blue thread waits() on CV

Monitors



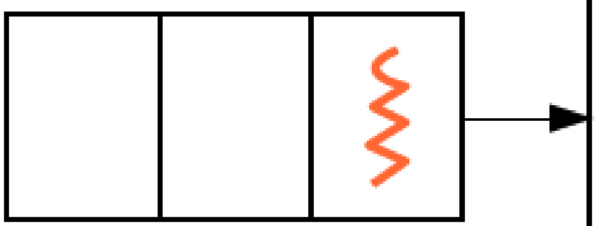
1) Thread enters monitor (grabs lock)

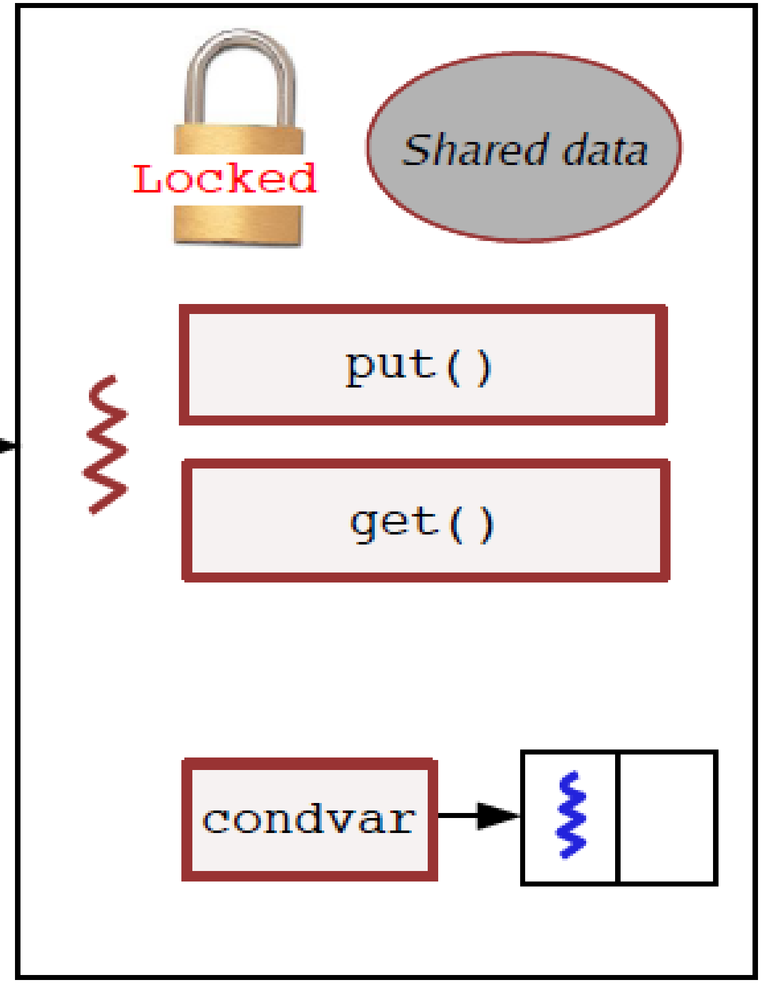
2) Other threads queue up

3) Blue thread waits() on CV

4) Next thread enters monitor

Monitors





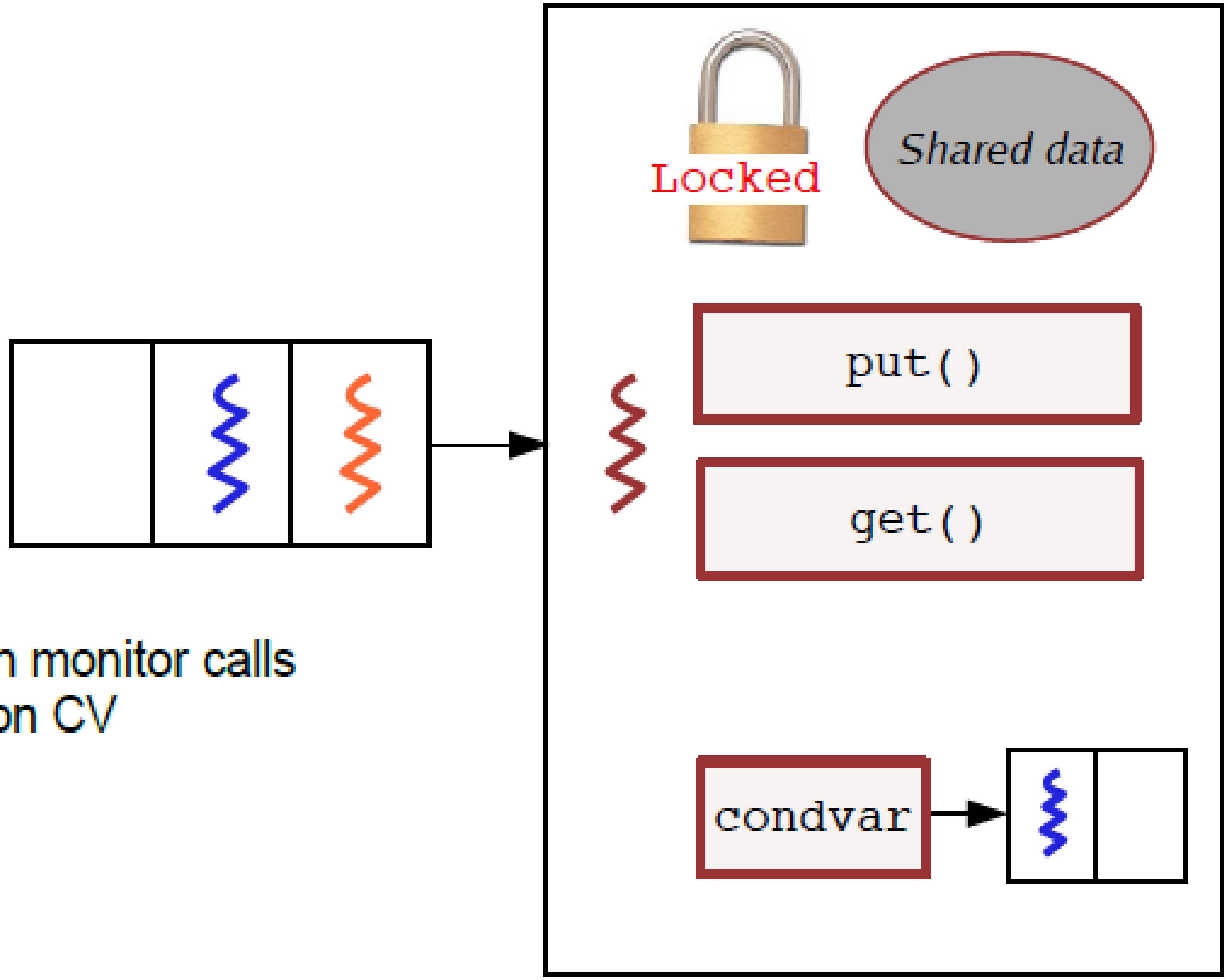
1) Thread enters monitor
(grabs lock)

2) Other threads queue up

3) Blue thread waits() on CV

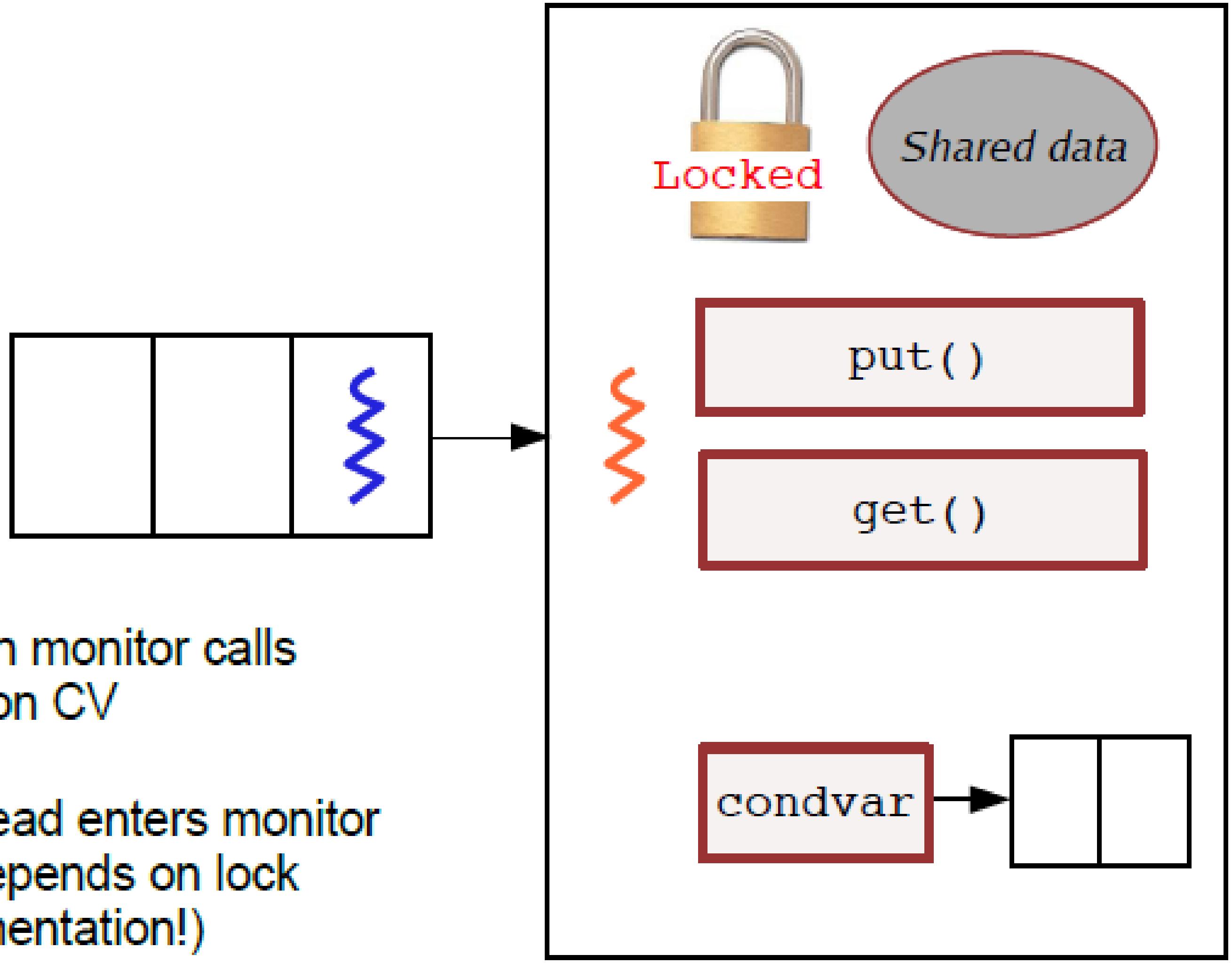
4) Next thread enters monitor

Monitors



5) Thread in monitor calls notify() on CV

Monitors



5) Thread in monitor calls notify() on CV

6) Next thread enters monitor (order depends on lock implementation!)

Dining-philosophers Problem with Monitor

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
```

```
void pickup(int i)
void putdown(int i)
void test(int i)
```

```
void init() {
    for (int i = 0; i < 5; i++)
            state[i] = thinking;
        }
```

Dining-philosophers Problem with Monitor

```
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) mod 5);
test((i+1) mod 5);
 }
```

Dining-philosophers Problem with Monitor

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

Producer-Consumer Problem with Monitor

```
Monitor Producer-consumer
{
    condition full, empty;
    int count;
    void insert(int item); //the following slide
    int remove(); //the following slide
    void init() {
                count = 0;
      }
}
```

Producer-Consumer Problem with Monitor

```
void insert(int item)
  {
  if (count == N)  full.wait();
     insert_item(item);  // Add the new item
  count ++;
  if (count == 1)  empty.signal();
    }
```

```
int remove()
  { int m;
  if (count == 0)  empty.wait();
  m = remove_item(); // Retrieve one item
  count --;
  if (count == N -1)  full.signal();
     return m;
    }
```

Producer-Consumer Problem with Monitor

```
void producer()        //Producer process
      {
      while (1) {
            item = Produce_Item();
            Producer-consumer.insert(item);
                  }
    }
```

```
void consumer()  //Consumer process
  {
     while (1) {
           item = Producer-consumer.remove(item);
           consume_item(item);
                 }
    }
```

Monitor VS Semaphores

Semaphores

- ✧wait()/signal() implement blocking mutual exclusion
- ✧Also used as atomic counters (counting semaphores)
- ✧Can be inconvenient to change and debug

Monitors

Synchronizes execution within procedures that manipulate encapsulated data shared among procedures

- ✧Only one thread can execute within a monitor at a time
- ✧Relies upon high-level language support

Monitor VS Semaphores

Semaphores (Disadvantages):

Low-level:

- Easy to forget a Set or a Reset of the Semaphore

Scattered:

- Semaphore calls are scattered in the code
- Difficult and error-prone to debug (aliasing!).