

# Chapter 5. Process Synchronization

Abraham Silberschatz, Peter Baer Galvin, Greg Gagne.  
(2018). *Operating System Concepts* (10th ed.). Wiley.



# Contents

- 1. Background
- 2. Critical-Section Problem
- 3. Peterson's Solution
- 4. Hardware Support for Synchronization
- 5. Mutex Locks
- 6. Semaphores
- 7. Monitors



# Contents

- 8. Classic Problems of Synchronization
- 9. Synchronization in Java



# 1. Background

- Processes can execute concurrently or in parallel.
- One process may only partially complete execution before another process is scheduled.
  - A process may **be interrupted at any point** in its instruction stream, and the processing core may be assigned to execute instructions of another process.
- Concurrent access to shared data may result in data inconsistency.
- A situation, where several processes **access and manipulate the same data concurrently** and the outcome of the execution **depends on the particular order** in which the access takes place, is called a **race condition**.



# 1. Background

## Race Condition

- Example:

### Producer Process

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

### Consumer Process

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



# 1. Background

## Race Condition

- Let: *counter* = 5
- If I run two operations *counter ++* and *counter --* in any order:

*counter ++;*

*counter --;*

or

*counter -- ;*

*counter++ ;*

What should be the value of *counter*?



# 1. Background

## Race Condition

- Let: *counter* = 5
- If I run two operations *counter ++* and *counter--* in any order:

*counter ++;*

*counter--;*

or

*counter -- ;*

*counter++ ;*

What should be the value of *counter*?

### Producer

***counter ++***

- $\text{register}_1 = \text{counter}$
- $\text{register}_1 = \text{register}_1 + 1$
- $\text{counter} = \text{register}_1$

### Consumer

***counter--***

- $\text{register}_2 = \text{counter}$
- $\text{register}_2 = \text{register}_2 - 1$
- $\text{counter} = \text{register}_2$

# 1. Background

## Race Condition

### Producer

counter ++

- $\text{register}_1 = \text{counter}$
- $\text{register}_1 = \text{register}_1 + 1$
- $\text{counter} = \text{register}_1$

### Consumer

counter--

- $\text{register}_2 = \text{counter}$
- $\text{register}_2 = \text{register}_2 - 1$
- $\text{counter} = \text{register}_2$

Time	Execute	Register 1	Register 2	Counter
T=0	$\text{register}_1 = \text{counter}$	5		5
T=1	$\text{register}_1 = \text{register}_1 + 1$ $\text{register}_2 = \text{counter}$	6	5	5
T=2	$\text{counter} = \text{register}_1$ $\text{register}_2 = \text{register}_2 - 1$	6	4	6
T=3	$\text{counter} = \text{register}_2$		4	4





## 2. Critical-Section Problem

- Consider system of  $n$  processes  $\{P_0, P_1, \dots, P_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- *Critical section problem* is to design protocol to solve this.
- Each process must ask permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.



## 2. Critical-Section Problem

### Critical Section

- General structure of process  $P_i$ :

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```

## 2. Critical-Section Problem

### Critical Section

#### Producer Process

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

#### Consumer Process

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

Critical Section

## 2. Critical-Section Problem

### Solution for Critical-Section Problem

- Requirements for solution to critical-section problem
  - Mutual exclusion:** If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.
  - Progress:** If a process is not executing its critical section, then it should not stop any other process from entering their critical section. In other words, any process can execute its critical section, if any other process do not execute their critical section.
  - Bounded waiting:** Each process must have a limited waiting time. It should not wait forever to enter in the critical section.
  - No assumptions:** may be made about speeds or the number of CPUs.

P<sub>1</sub>

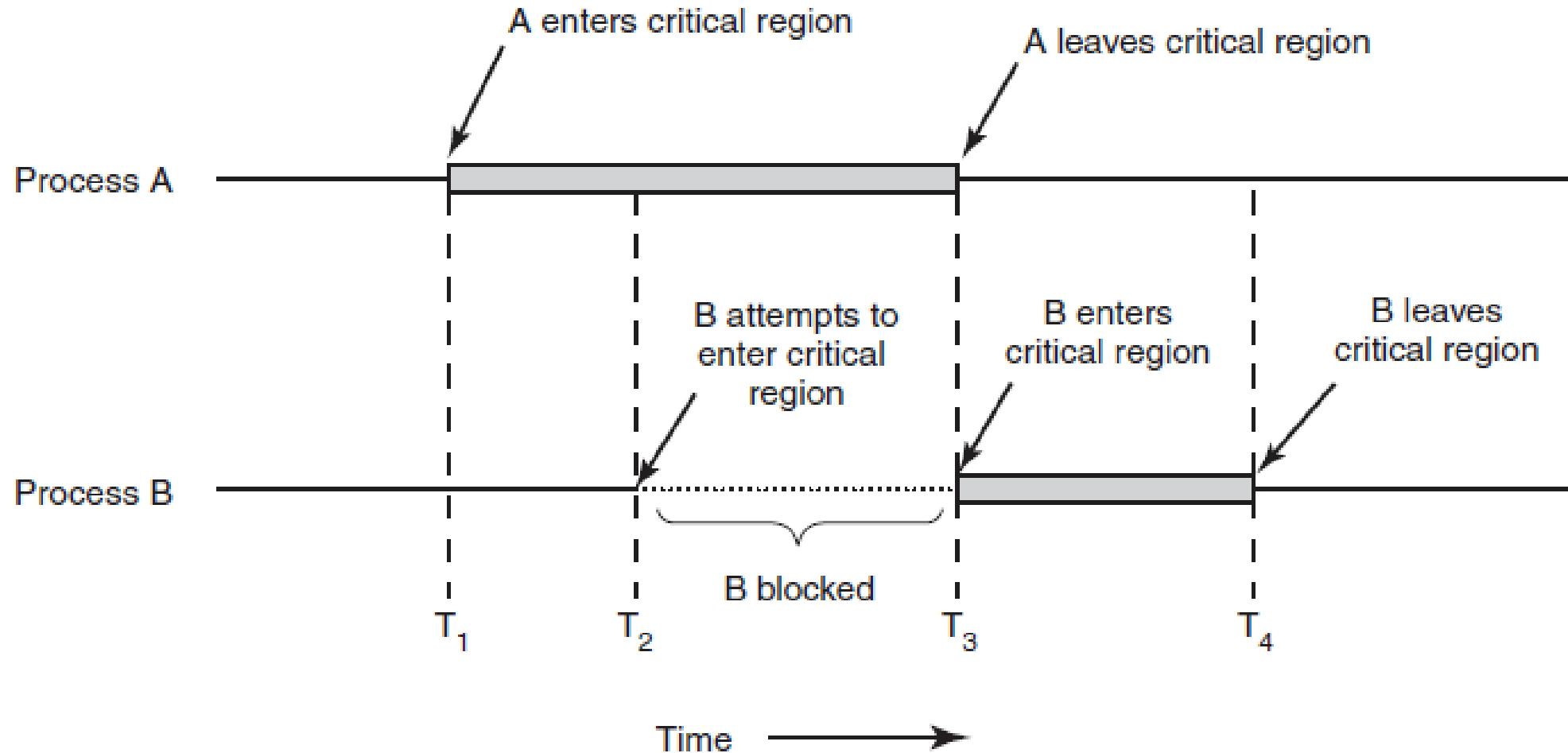
```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```

P<sub>2</sub>

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```

## 2. Critical-Section Problem

### Solution for Critical-Section Problem



### 3. Peterson's Solution

## A Software Solution

- Two-process solution
- The two processes share one variable:

```
int turn;
```

- The variable `turn` indicates whose turn it is to enter the critical section.
- Initially, the value of `turn` is set to 0.
- **Mutual exclusion is preserved.**

$P_i$  enters critical section only if:

```
turn = i
```

and `turn` cannot be both 0 and 1 at the same time

- **What about the Progress requirement? NO**


```
int turn = 0;
```

#### Algorithm for process $P_0$

```
while(true) {  
    while(turn == 1);  
    /* critical section */  
    turn = 1;  
    /* remainder section */  
}
```

#### Algorithm for process $P_1$

```
while(true) {  
    while(turn == 0);  
    /* critical section */  
    turn = 0;  
    /* remainder section */  
}
```



### 3. Peterson's Solution

- Two-process solution
- The two processes share one variable:
  - `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  $P_i$  is ready!
- **Mutual exclusion is preserved.**
- **Progress requirement is satisfied**
- **Bounded-waiting requirement is met**

```
int turn = 0;
boolean flag[2];
flag[0] = flag[1] = false;
```

#### Algorithm for process $P_0$

```
while(true) {
    flag[0] = true;
    turn = 1;
    while(flag[1] && turn == 1);
    /* critical section */
    flag[0] = false;
    /* remainder section */
}
```

#### Algorithm for process $P_1$


```
while(true) {
    flag[1] = true;
    turn = 0;
    while(flag[0] && turn == 0);
    /* critical section */
    flag[1] = false;
    /* remainder section */
}
```



### 3. Peterson's Solution

- Although useful for demonstrating an algorithm, Peterson's solution **is not guaranteed to work on modern architectures.**
  - Because to improve performance, processors and/or compilers on modern architecture may **reorder operations that have no dependencies.**





### 3. Peterson's Solution

- Modern architecture may **reorder** operations that have no dependencies

Two threads share the data:

```
boolean flag = false;  
int x = 0;
```

Thread 1 performs

```
while (!flag);  
print x
```

Thread 2 performs

```
x = 100;  
flag = true
```

However, since the variables `flag` and `x` are independent of each other, the instructions may be reordered.



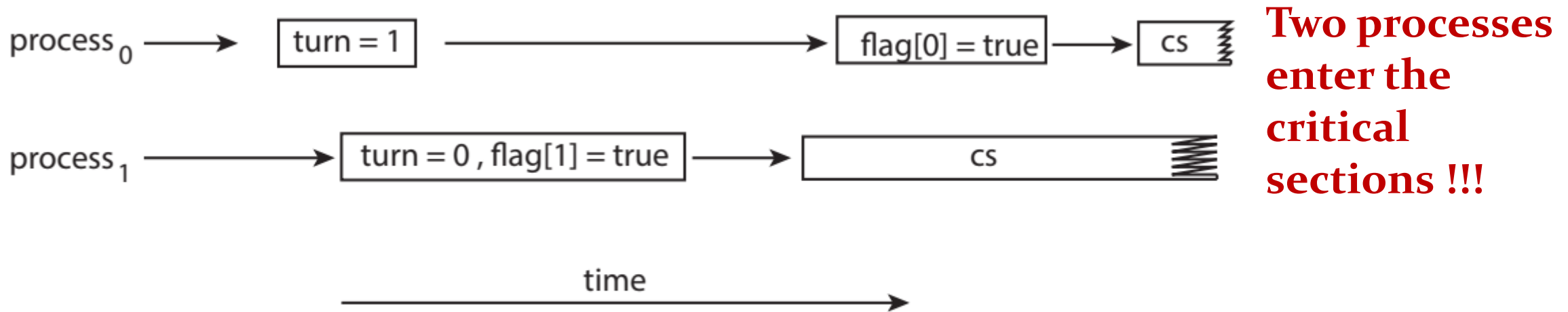
```
flag = true;  
x = 100;
```

What is the expected output?

100

If this occurs, the output may be 0!

### 3. Peterson's Solution



**Figure 6.4** The effects of instruction reordering in Peterson's solution.

Software-based solutions are not guaranteed to work on modern computer architectures.



## 4. Hardware Support for Synchronization

### Memory Barriers

- In general, a memory model falls into one of two categories:
  - 1. **Strongly ordered**, where a memory modification on one processor is immediately visible to all other processors.
  - 2. **Weakly ordered**, where modifications to memory on one processor may not be immediately visible to other processors.
- Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor.
  - To address this issue, computer architectures provide instructions (**memory barriers**) that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors.



## 4. Hardware Support for Synchronization

### Memory Barriers

- If we add a memory barrier operation to Thread 1

```
while (!flag)
    memory_barrier();
print x;
```

we guarantee that the value of `flag` is loaded before the value of `x`.

- Similarly, if we place a memory barrier between the assignments performed by Thread 2

```
x = 100;
memory_barrier();
flag = true;
```

we ensure that the assignment to `x` occurs before the assignment to `flag`.



## 4. Hardware Support for Synchronization

### Memory Barriers

- We could place a memory barrier between the first two assignment statements in the entry section to avoid the reordering of operations.
- Note that memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

```
int turn = 0;
boolean flag[2];
flag[0] = flag[1] = false;
```

#### Algorithm for process $P_0$

```
while(true) {
    flag[0] = true;
    memory_barrier();
    turn = 1;
    while(flag[1] && turn == 1);
    /* critical section */
    flag[0] = false;
    /* remainder section */
}
```

#### Algorithm for process $P_1$

```
while(true) {
    flag[1] = true;
    memory_barrier();
    turn = 0;
    while(flag[0] && turn == 0);
    /* critical section */
    flag[1] = false;
    /* remainder section */
}
```



## 4. Hardware Support for Synchronization

### Hardware Instructions

- Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically** - that is, as one **uninterruptible** unit.
  - `test_and_set()`
  - `compare_and_swap()`



## 4. Hardware Support for Synchronization

### `test_and_set()`

---

```
boolean test_and_set(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
  
    return rv;  
}
```

---

**Figure 6.5** The definition of the atomic `test_and_set()` instruction.

- Properties
  - Executed **atomically**
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to `true`



## 4. Hardware Support for Synchronization

### `test_and_set()`

- Solution using `test_and_set()`
  - Shared boolean variable `lock`, initialized to `false`
  - Solution:

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
    /* remainder section */  
} while (true);
```

- Does it solve the critical-section problem?





## 4. Hardware Support for Synchronization

### `compare_and_swap()`

---

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

---

**Figure 6.7** The definition of the atomic `compare_and_swap()` instruction.

- Properties
  - Executed **atomically**
  - Returns the original value of passed parameter value
  - Set the variable value the value of the passed parameter **new\_value** but only **if \*value == expected** is true. That is, the swap takes place only under this condition.



## 4. Hardware Support for Synchronization

### `compare_and_swap()`

- Solution using `compare_and_swap()`
  - Shared integer variable `lock`, initialized to 0
  - Solution:

```
while (true){  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
}
```

- Does it solve the critical-section problem?



## 4. Hardware Support for Synchronization

### Atomic Variables

- Typically, instructions such as `compare_and_swap` are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides atomic (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let `sequence` be an atomic variable
  - Let `increment()` be operation on the atomic variable `sequence`
- The command:
  - `increment(&sequence);`ensures `sequence` is incremented without interruption.



## 4. Hardware Support for Synchronization

### Atomic Variables

- The `increment()` function can be implemented using CAS as follows:

```
void increment(atomic_int *v) {  
    int temp;  
    do {  
        temp = *v;  
    } while (temp != (compare_and_swap(v, temp, temp+1)) );  
}
```



## 5. Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers. OS designers build software tools to solve critical section problem.
- Simplest is **mutex lock**.
  - A Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to `acquire()` and `release()` must be **atomic**.
- **Implemented via hardware atomic instructions such as `compare_and_swap`.**
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

```
while (true) {  
    acquire lock  
        critical section  
    release lock  
        remainder section  
}
```



## 5. Mutex Locks

- The definition of `acquire()` is as follows:

```
acquire() {  
    while (available == false)  
        ; /* busy wait */  
    available = false;  
}
```

**Mutex locks can be implemented using the CAS operation.**

- The definition of `release()` is as follows:

```
release() {  
    available = true;  
}
```



## 6. Semaphores

- A **semaphore**  $S$  is an integer variable that, apart from initialization, is accessed only through two standard **atomic** operations: `wait()` (or `down()`) and `signal()` (or `up()`).
  - Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the `wait()` operation was originally termed  $P$  (from the Dutch *proberen*, “to test”); `signal()` was originally called  $V$  (from *verhogen*, “to increment”).



## 6. Semaphores

- Definition

```
wait(S) {  
    while (S <= 0)  
        ; // busy wait  
    S--;  
}
```

```
signal(S) {  
    S++;  
}
```

- All modifications to the integer value of the semaphore in the `wait()` and `signal()` operations must be executed **atomically**. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.





## 6. Semaphores

- There are two types of semaphores:
  - Binary semaphore
  - Counting Semaphore
- A binary semaphore can have only two integer values: 0 or 1.
  - Binary semaphores behave similarly to mutex locks.
- A counting semaphore is an integer value, which can range over an unrestricted domain
  - is used to resolve synchronization problems like resource allocation.



## 6. Semaphores

- Binary semaphore

```
semaphore S = 1;  
while(true) {  
    wait(S);  
    /* critical section */  
    signal(S);  
    /* remainder section */  
}
```

### Mutex Lock

```
while (true) {  
    acquire lock  
        critical section  
    release lock  
        remainder section  
}
```



## 6. Semaphores

- Solution to the critical section problem

```
semaphore S = 1;  
while(true) {  
    wait(S);  
    /* critical section */  
    signal(S);  
    /* remainder section */  
}
```



## 6. Semaphores

- Consider two processes  $P_1$  and  $P_2$  that with two statements  $S_1$  and  $S_2$  and **the requirement that  $S_1$  to happen before  $S_2$** .
- Solution: Create a semaphore “synch” initialized to 0

$P_1$ :

$S_1$ ;

signal(synch);

$P_2$ :

wait(synch);

$S_2$ ;



## 6. Semaphores

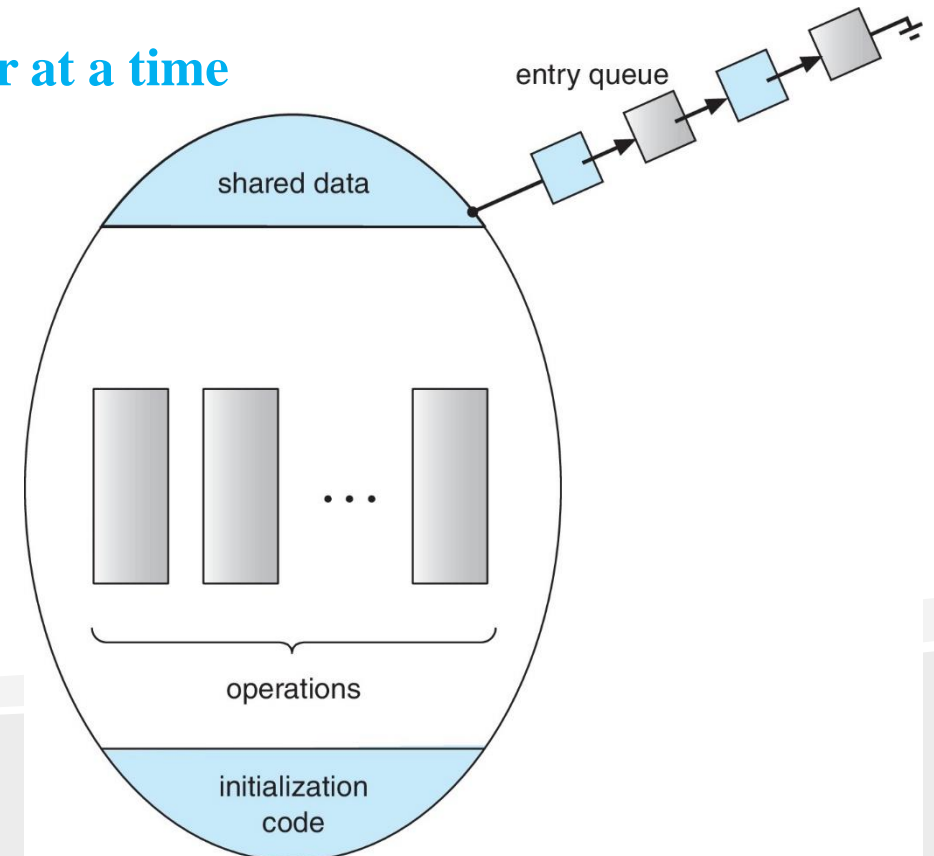
### Semaphore Implementation with No Busy Waiting

- The implementation of mutex locks suffers from **busy waiting**. The definitions of the `wait()` and `signal()` semaphore operations just described present the same problem.
- To overcome this problem, we can modify the definition of the `wait()` and `signal()` operations as follows:
  - When a process executes the `wait()` operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, **the process can suspend itself**. The suspend operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the **waiting state**. Then control is transferred to the CPU scheduler, which selects another process to execute.

## 7. Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- **Only one process may be active within the monitor at a time**
- Pseudocode syntax of a monitor:

```
monitor monitor-name {  
    // shared variable declarations  
  
    procedure P1 (...) { ... }  
  
    procedure P2 (...) { ... }  
  
    procedure Pn (...) {.....}  
  
    initialization code (...) { ... }  
}
```





## 7. Monitors

### Monitor Implementation Using Semaphores

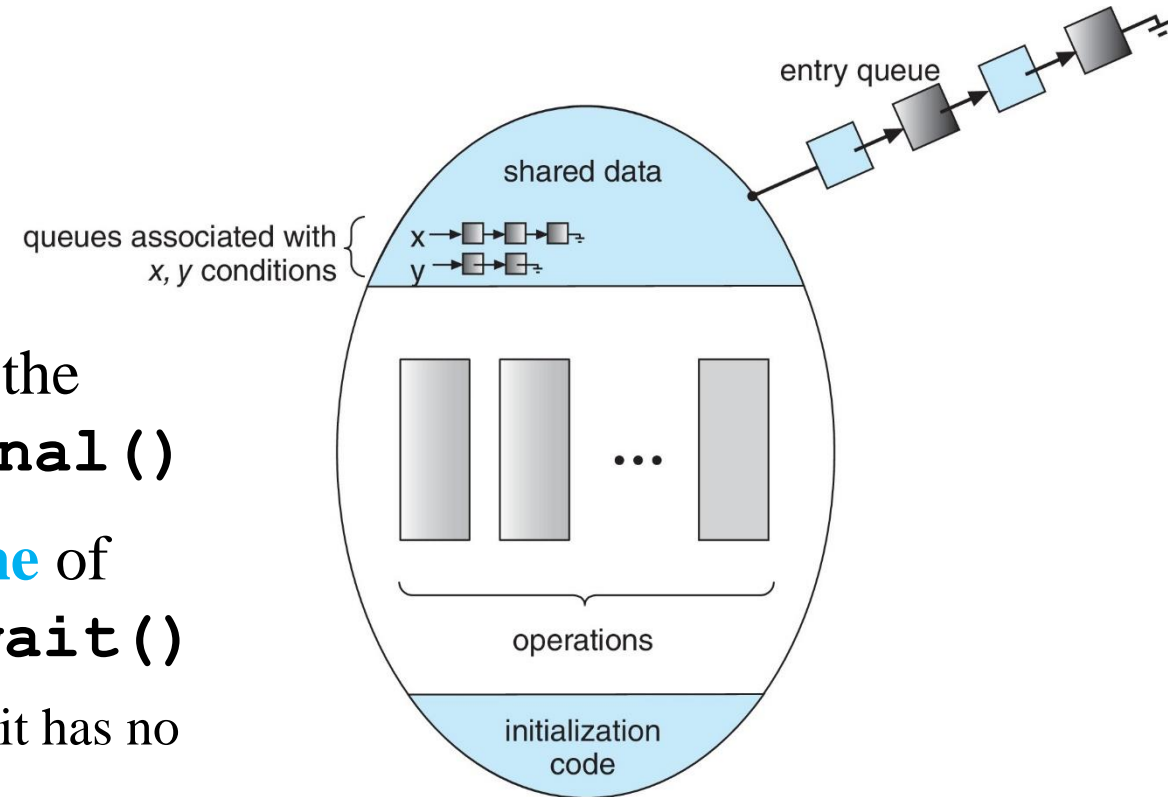
- Variables
  - `semaphore mutex = 1`
- Each procedure  $P$  is replaced by

```
wait(mutex) ;  
...  
body of P;  
...  
signal(mutex) ;
```
- Mutual exclusion within a monitor is ensured.

# 7. Monitors

## Condition Variables

- **condition  $x, y$ ;**
- Two operations are allowed on a condition variable:
  - **$x.\text{wait}()$** : A process that invokes the operation is suspended until  **$x.\text{signal}()$**
  - **$x.\text{signal}()$** : Resumes **exactly one** of processes (if any) that invoked  **$x.\text{wait}()$** 
    - If no  **$x.\text{wait}()$**  on the variable, then it has no effect on the variable.







## 8. Classic Problems of Synchronization

- The bounded-buffer problem
- The readers-writers problem
- The dining-philosophers problem

## 8. Classic Problems of Synchronization

### The Bounded-Buffer Problem

```
#define BUFFER_SIZE 10  
typedef struct {  
    . . .  
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

```
int counter = 0;
```

**Mutual Exclusion: NO  
Race condition occurs !!!**

#### Producer Process

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

#### Consumer Process

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

## 8. Classic Problems of Synchronization

### The Bounded-Buffer Problem

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;

int counter = 0;

semaphore mutex = 1;
```

**Mutual Exclusion: YES**  
**Busy Waiting !!!**

#### Producer Process

```
while(true) {
    /* produce an item in next_produced */
    while(count == BUFFER_SIZE)
        ; /* do nothing */
    wait(mutex);
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
    signal(mutex);
}
```

#### Consumer Process

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

## 8. Classic Problems of Synchronization

### The Bounded-Buffer Problem

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
semaphore mutex = 1;
semaphore empty = BUFFER_SIZE;
semaphore full = 0;
```

**Mutual Exclusion: YES**  
**Busy Waiting: NO**

#### Producer Process

```
while(true) {
    /* produce an item in next_produced */
    wait(empty);
    wait(mutex);
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
    signal(mutex);
    signal(full);
}
```

#### Consumer Process

```
while(true) {
    wait(full);
    wait(mutex);
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    signal(mutex);
    signal(empty);
    /* consume the item in next_consumed */
}
```



## 8. Classic Problems of Synchronization

### The Readers-Writers Problem

- Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database (**readers**), whereas others may want to update (write) the database (**writers**).
  - If two readers access the shared data simultaneously, no adverse affects will result.
  - However, if a writer and some other thread (either a reader or a writer) accesses the database simultaneously, **chaos** may ensue.
  - To ensure that these difficulties do not arise, we require that **the writers have exclusive access to the shared database while writing to the database**. This synchronization problem is referred to as the **readers–writers problem**.



## 8. Classic Problems of Synchronization

### The Readers-Writers Problem

- The readers–writers problem has several variations, all involving priorities.
- The first readers–writers problem (the simplest one) requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
  - **Writers may starve.**
- The second readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
  - **Readers may starve.**
- A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.



## 8. Classic Problems of Synchronization

### The First Readers-Writers Problem

```
semaphore rw mutex = 1;  
semaphore mutex = 1;  
int read_count = 0;
```

#### Writer Process

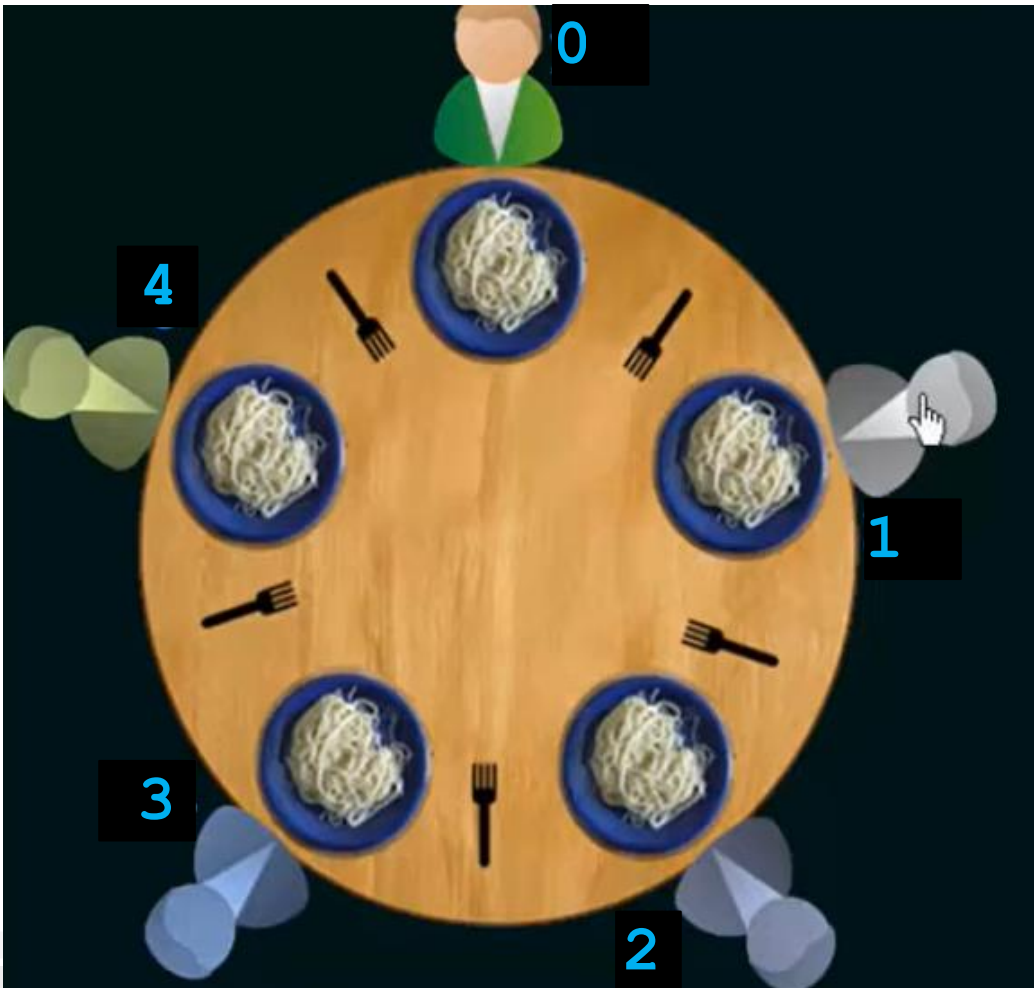
```
while(true) {  
    wait(rw_mutex);  
  
    /* writing is performed */  
  
    signal(rw_mutex);  
}
```

#### Reader Process

```
while(true) {  
    wait(mutex);  
    read_count++;  
    if(read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
  
    /* reading is performed */  
  
    wait(mutex);  
    read_count--;  
    if(read_count == 0)  
        signal(rw_mutex);  
    signal(mutex);  
}
```

## 8. Classic Problems of Synchronization

### The Dining-Philosophers Problem



- Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork.
- The life of a philosopher consists of alternating periods of eating and thinking. When a philosopher gets sufficiently hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think.
- The key question is: Can we write a program for each philosopher that does what it is supposed to do and never gets stuck?



## 8. Classic Problems of Synchronization

### The Dining-Philosophers Problem

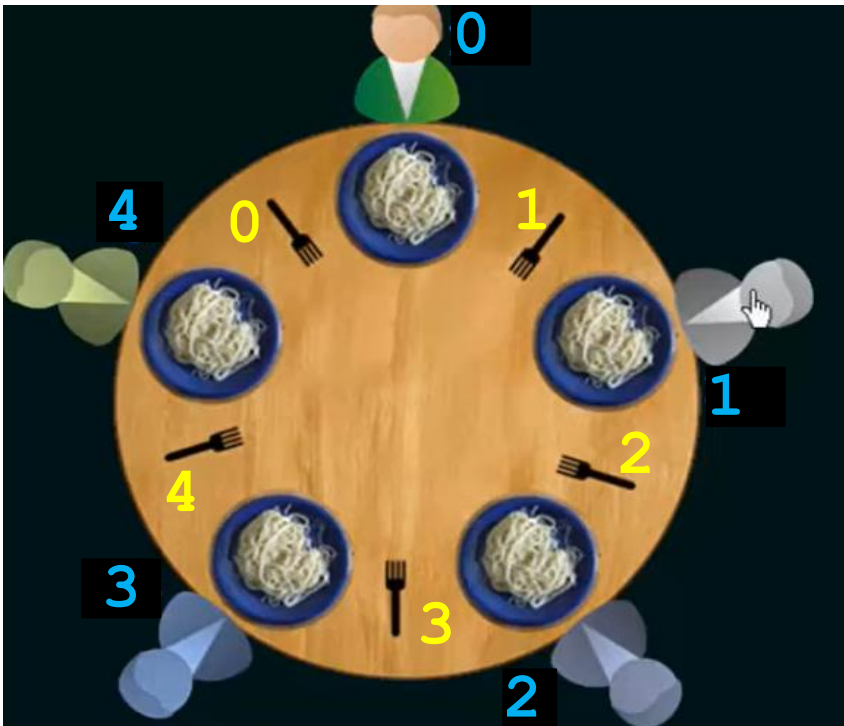
- A nonsolution to the problem

```
semaphore fork[5];
```

```
// All the elements of fork are initialized to 1.
```

```
Philosopher i (i = 0..4) Process
```

```
while(true) {  
    wait(fork[i]); // get right fork  
    wait(fork[i + 1] % 5); // get left fork  
  
    /* eat for a while */  
  
    signal(fork[i]); // release right fork  
    signal(fork[i + 1] % 5); // release left fork  
  
    /* think for a while */  
}
```



Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a **deadlock**. Suppose that all five philosophers become hungry at the same time and each grabs her right fork. All the elements of fork will now be equal too. When each philosopher tries to grab her left fork, she will be delayed forever.

## 8. Classic Problems of Synchronization

### The Dining-Philosophers Problem

- This solution imposes the restriction that a philosopher may pick up her fork only if both of them are available.

```
monitor DiningPhilosophers
{
    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((i + 4) % 5);
    test((i + 1) % 5);
}

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;
}
}
```



## 8. Classic Problems of Synchronization

### The Dining-Philosophers Problem

- Thus, philosopher  $i$  must invoke the operations `pickup()` and `putdown()` in the following sequence:

```
while(true) {  
    DiningPhilosophers.pickup(i) ;  
    ...  
    /* eat for a while */  
    ...  
    DiningPhilosophers.putdown(i) ;  
    ...  
    /* think for a while */  
}
```

- It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. **However, it is possible for a philosopher to starve to death.**



## 9. Synchronization in Java

- Java provides a rich set of synchronization features:
  - Java monitors
  - Reentrance locks
  - Semaphores
  - Condition variables



## 9. Synchronization in Java

### Java Monitors

- Every Java object has associated with it a **single lock**.
- If a method is declared as **synchronized**, a calling thread **must require owning the lock for the object**.
- If the lock is owned by another thread, the calling thread blocks and **waits for the lock until it is released**.
- Locks are released when the owning thread exits the **synchronized** method.

```
public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

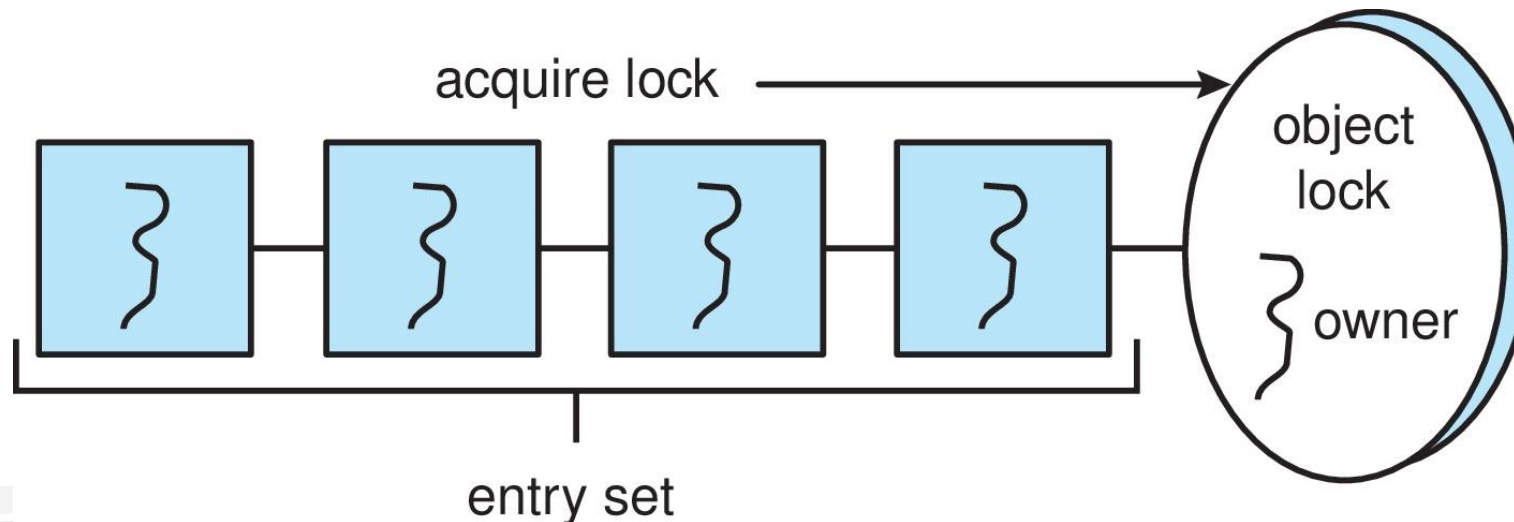
    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}
```

# 9. Synchronization in Java

## Java Monitors

- A thread that tries to acquire an **unavailable lock of the object**:
  1. The state of the thread is set to **blocked**.
  2. The thread is placed in the **object's entry set**. The entry set represents the set of threads waiting for the lock to become available.

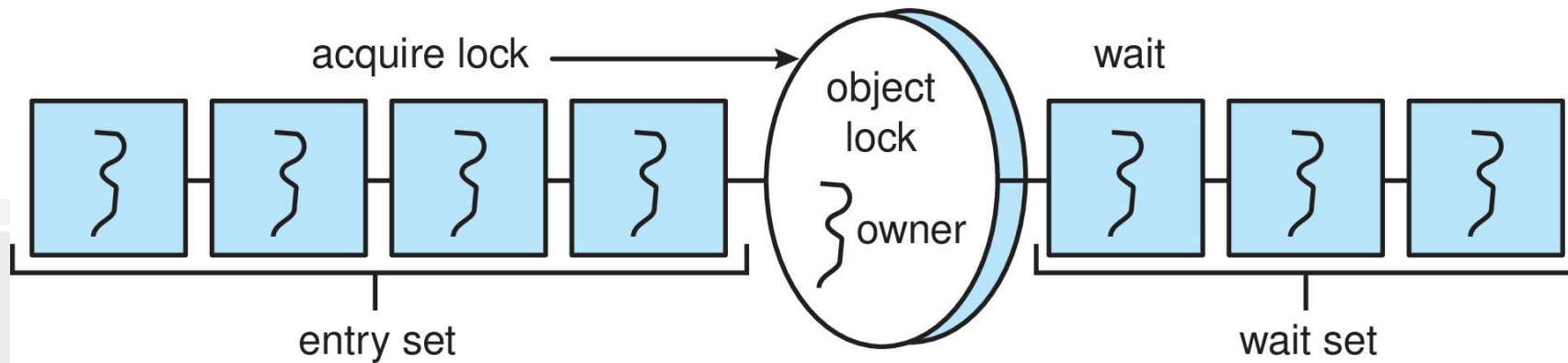


# 9. Synchronization in Java

## Java Monitors

- Similarly, each object also has a **wait set**.
- When a thread call **wait()** of the object (**the thread must be now the owner of the lock**):
  1. It **releases** the lock for the object.
  2. The state of the thread is set to **blocked**.
  3. The thread is placed in the **wait set** for the object.

**A thread typically calls `wait()` when it is waiting for a condition to become true.**





## 9. Synchronization in Java

### Java Monitors

- A thread typically calls `wait()` when it is waiting for a condition to become true.
- How does a thread get notified?
- When a thread calls `notify()`:
  1. An arbitrary thread `T` is selected from the **wait set**.
  2. `T` is moved from the **wait set** to the **entry set**.
  3. `T` can now compete for the object's lock. If the object's lock is available, `T` becomes the owner of the object's lock and can enter the method.



# 9. Synchronization in Java

## Java Monitors

```
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

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

    notify();
}
```

```
/* Consumers call this method */
public synchronized E remove() {
    E item;

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

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

    notify();

    return item;
}
```

# 9. Synchronization in Java

## Java Monitors

### Block Synchronization

- The amount of time between when a lock is acquired and when it is released is defined as the **scope** of the lock.
- A **synchronized** method that has only a small percentage of its code manipulating shared data may yield a scope that is too large.
- In such an instance, it may be better to synchronize only the block of code that manipulates shared data than to synchronize the entire method. Such a design results in a smaller lock scope.
- In addition to declaring **synchronized** methods, Java also allows block synchronization. Only the access to the critical-section code requires ownership of the object lock for the this object.

```
public void someMethod() {  
    /* non-critical section */  
  
    synchronized(this) {  
        /* critical section */  
    }  
  
    /* remainder section */  
}
```



## 9. Synchronization in Java

### Reentrant Locks

- Similar to mutex locks
- The `finally` clause ensures the lock will be released in case an exception occurs in the try block.

```
Lock key = new ReentrantLock();

key.lock();
try {
    /* critical section */
}
finally {
    key.unlock();
}
```



## 9. Synchronization in Java

### Semaphores

- Constructor

```
Semaphore (int value) ;
```

- Usage

```
Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    /* critical section */
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}
```



## 9. Synchronization in Java

### Condition Variables

- Condition variables are associated with an `ReentrantLock`.
- Creating a condition variable using `newCondition()` method of `ReentrantLock`:

```
Lock key = new ReentrantLock();  
Condition condVar = key.newCondition();
```

- Once the condition variable has been obtained, we can invoke its `await()` and `signal()` methods of the condition.
  - A thread **that is owning the lock** waits by calling the `await()` method, and signals other thread (which are waiting on this lock) by calling the `signal()` method.



## 9. Synchronization in Java

### Condition Variables

- Example:
  - Five threads numbered 0 .. 4
  - Shared variable `turn` indicating which thread's turn it is.
  - Thread calls `doWork()` when it wishes to do some work. (**But it may only do work if it is their turn.**)
  - If not their turn, wait
  - If their turn, do some work for awhile .....
  - When completed, notify the thread whose turn is next.
- Necessary data structures:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition();
```



## 9. Synchronization in Java

### Condition Variables



```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
         */

        /**
         * Now signal to the next thread.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}
```

# 9. Synchronization in Java

## Condition Variables

synchronized methods

```
BoundedBuffer.java ×
1
2 public class BoundedBuffer<E> {
3     private static final int BUFFER_SIZE = 5;
4     private int count, in, out;
5
6     private E[] buffer; //FIFO buffer
7
8     public BoundedBuffer() {
9         count = in = out = 0;
10        buffer = (E[])new Object[BUFFER_SIZE];
11    }
```

ReentrantLock

```
BoundedBuffer2.java ×
1 import java.util.concurrent.locks.Condition;
2
3
4 public class BoundedBuffer2<E> {
5     private static final int BUFFER_SIZE = 5;
6     private int count, in, out;
7
8     private E[] buffer; //FIFO buffer
9
10    private ReentrantLock lock;
11    private Condition cond;
12
13    public BoundedBuffer2() {
14        count = in = out = 0;
15        buffer = (E[])new Object[BUFFER_SIZE];
16
17        this.lock = new ReentrantLock();
18        this.cond = this.lock.newCondition();
19    }
```



## 9. Synchronization in Java

### Condition Variables

#### synchronized methods

```
13 //Producers call this method
14 ⊖ public synchronized void insert(E item) {
15     while (count == BUFFER_SIZE) {
16         try {
17             wait();
18         }
19         catch (InterruptedException ie) { }
20     }
21     buffer[in] = item;
22     in = (in + 1) % BUFFER_SIZE;
23     count++;
24     notify();
25 }
```

#### ReentrantLock

```
21 //Producers call this method
22 ⊖ public void insert(E item) {
23     this.lock.lock();
24     try {
25         while (count == BUFFER_SIZE) {
26             try {
27                 this.cond.await();
28             } catch (InterruptedException e) { }
29         }
30         buffer[in] = item;
31         in = (in + 1) % BUFFER_SIZE;
32         count++;
33         this.cond.signal();
34     }
35     finally {
36         this.lock.unlock();
37     }
38 }
```

# 9. Synchronization in Java

## Condition Variables

### synchronized methods

```
27 //Consumers call this method
28 public synchronized E remove() {
29     E item;
30     while (count == 0) {
31         try {
32             wait();
33         }
34         catch (InterruptedException ie) { }
35     }
36     item = buffer[out];
37     out = (out + 1) % BUFFER_SIZE;
38     count--;
39     notify();
40     return item;
41 }
42 }
```

### ReentrantLock

```
40 //Consumers call this method
41 public E remove() {
42     E item = null;
43
44     this.lock.lock();
45     try {
46         while (count == 0) {
47             try {
48                 this.cond.await();
49             }
50             catch (InterruptedException ie) { }
51         }
52         item = buffer[out];
53         out = (out + 1) % BUFFER_SIZE;
54         count--;
55         this.cond.signal();
56     }
57     finally {
58         this.lock.unlock();
59     }
60     return item;
61 }
62 }
```



# 9. Synchronization in Java

## Notes

- A *semaphore* is a very general synchronization primitive, and most other synchronization operations can be reduced to semaphore operations. Semaphores are in most cases too basic an abstraction to be used effectively. There are a few simple problems that are best solved with semaphores, but in general locks and condition variables are a much better abstraction.
- A *lock* is an object that can be held by at most one thread at a time. Only the thread that last acquired a lock is allowed to release that lock. Locks are useful for guarding critical sections.
- A *condition variable* is an object used in combination with its associated lock to allow a thread to wait for some condition while it is **inside a critical section**.
  - Only a thread holding the associated lock is allowed to use a condition variable associated with that lock.
  - A condition variable has only one associated lock, but multiple condition variables may be associated with a single lock.



## 9. Synchronization in Java

### Notes

- A *monitor* is a critical section guarded by a lock, plus all the condition variables associated with that lock.
  - A straightforward example of a monitor is the set of `synchronized` methods in a Java class.
  - Each Java Object that has `synchronized` methods also has a `lock` and a single conditional variable.
  - When you call a `synchronized` method, Java automatically does a `lock.acquire()`, and when you return from the method (explicitly or through a `throw` statement), Java automatically releases the lock.



# 9. Synchronization in Java

## Notes

- Mesa versus Hoare Monitors
  - What should happen when `signal()` is called?
    - No waiting threads  $\Rightarrow$  the signaler continues and the signal is effectively lost (unlike what happens with semaphores).
    - If there is a waiting thread, one of the threads starts executing, others must wait
  - **Mesa-style:** (Java, and most real operating systems)
    - The thread that signals keeps the lock (and thus the processor).
    - The waiting thread waits for the lock.
  - **Hoare-style:** (most textbooks)
    - The thread that signals gives up the lock and the waiting thread gets the lock.
    - When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signaling thread.

# 9. Synchronization in Java

## Notes

- Note on calling `await()` method of the `Condition` object:

```
void await()  
    throws InterruptedException
```

Causes the current thread to wait until it is signalled or interrupted.

The lock associated with this `Condition` is atomically released and the current thread becomes disabled for thread scheduling purposes and lies dormant until *one* of four things happens:

- Some other thread invokes the `signal()` method for this `Condition` and the current thread happens to be chosen as the thread to be awakened; or
- Some other thread invokes the `signalAll()` method for this `Condition`; or
- Some other thread interrupts the current thread, and interruption of thread suspension is supported; or
- A "spurious wakeup" occurs.

- In computing, a **spurious wakeup** occurs when a thread wakes up from waiting on a condition variable without the variable being satisfied. It is referred to as spurious because the thread has seemingly been awakened **for no reason**.



## 9. Synchronization in Java

### Notes

- How to avoid a spurious wakeup?
  - A thread can also wake up without being notified, interrupted, or timing out, a so-called spurious wakeup. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops, like this one:

```
while (condition does not hold)  
    condition_obj.await();
```