

# MONITORS

---

Dr. Lavika Goel

# Monitors

- Semaphore is a low level primitive because its is unstructured
- In case of large systems being built, using semaphores alone, the responsibility for the correct use of semaphores would be diffused among all the implementations of the system
- Monitors provide a structured concurrent programming primitive that concentrates the responsibility for correctness into modules

# Monitors

- Monitors are generalizations of ***kernel*** or ***supervisor*** found in operating systems, where critical sections such as the allocation of memory are centralized in a privileged program
- Monitors are decentralized versions of the monolithic kernels
- A separate monitor is defined for each object or related group of objects that requires synchronization

# Monitors

- If operations of the same monitor are called by more than one process, the implementation ensures that these are executed under mutual exclusion
- If operations of different monitors are called, their executions can be interleaved
- Monitors have become an extremely important synchronization mechanism because they are a natural generalization of the **object** of OOP, which encapsulates data and operation declarations within a **class**

# Monitors

- Objects of a class can be allocated at runtime and the operations of the class invoked on the fields of the object
- Monitor adds the requirement that only one process can execute an operation on an object at any one time

# Monitors

- While the fields of the object may be declared either ***public*** (directly accessible outside the class) or ***private*** (accessible only by operations declared within the class), the fields of a monitor are all ***private***.
- This ensures that the fields of a monitor are accessed consistently
- Implementations of monitors in programming languages are quite different from one another

# Monitors

## Algorithm 7.1: Atomicity of monitor operations

monitor CS

integer  $n \leftarrow 0$

operation increment

integer temp

temp  $\leftarrow n$

$n \leftarrow \text{temp} + 1$

**p**

p1: CS.increment

**q**

q1: CS.increment

# Monitors

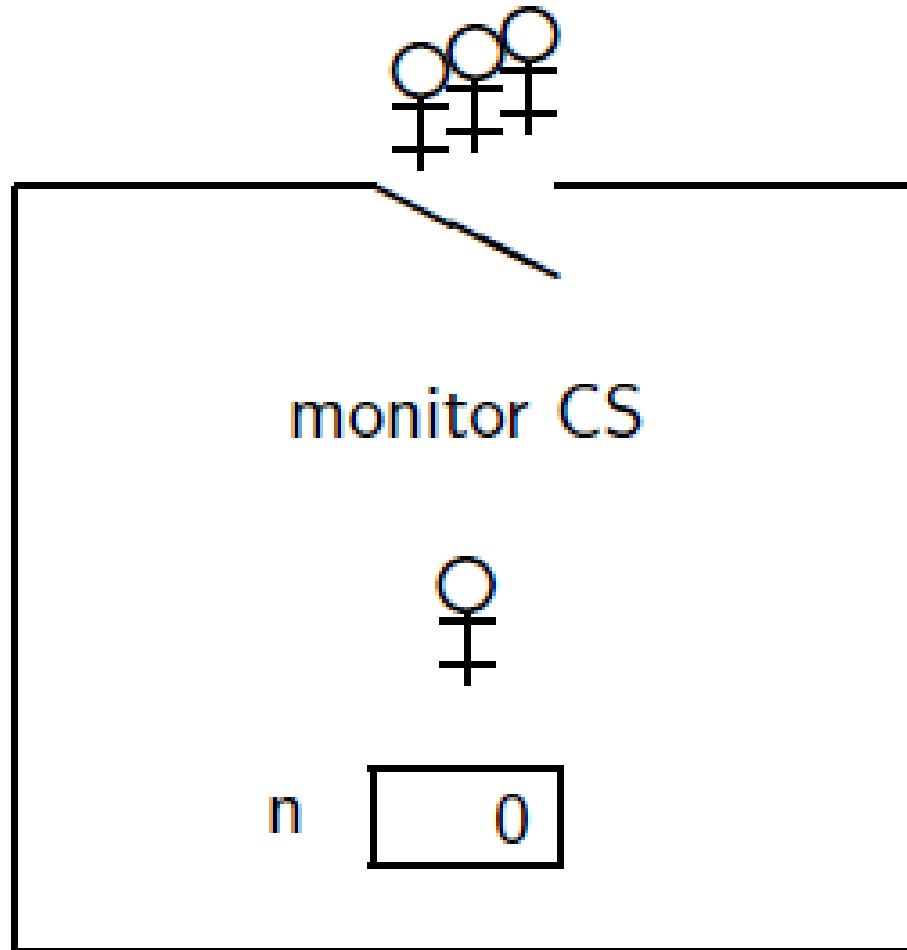
- Monitor **CS** contains one variable **n** and one operation **increment**
- Two statements are contained within this operation, together with the declaration of a local variable
- The variable **n** is not accessible outside the monitor
- Two processes **p** and **q**, each call the monitor operation **CS.increment**



# Monitors

- Only one process can execute the monitor operation and as mutual exclusion is ensured in access to the variable
- As one process is executing the statements of a monitor operation, the other statements are waiting outside
- This also solves the critical section problem as a critical section statements can be placed within the operation of a monitor

# Monitors



# Monitors

- The statements of critical section are encapsulated in the monitor rather than replicated in each process
- The synchronization is implicit and does not require the wait and signal statements
- Monitor is a static entry and not a dynamic process ... It is a set of operations that just sit there (keep waiting) waiting for a process to invoke one of them

# Monitors

- There is an implicit lock on the door to the monitor, ensuring only one process is inside the monitor at any time
- As with semaphores, if there are several processes attempting to enter a monitor, only one of them will succeed
- There is no explicit queue associated with the monitor entry, so starvation is possible

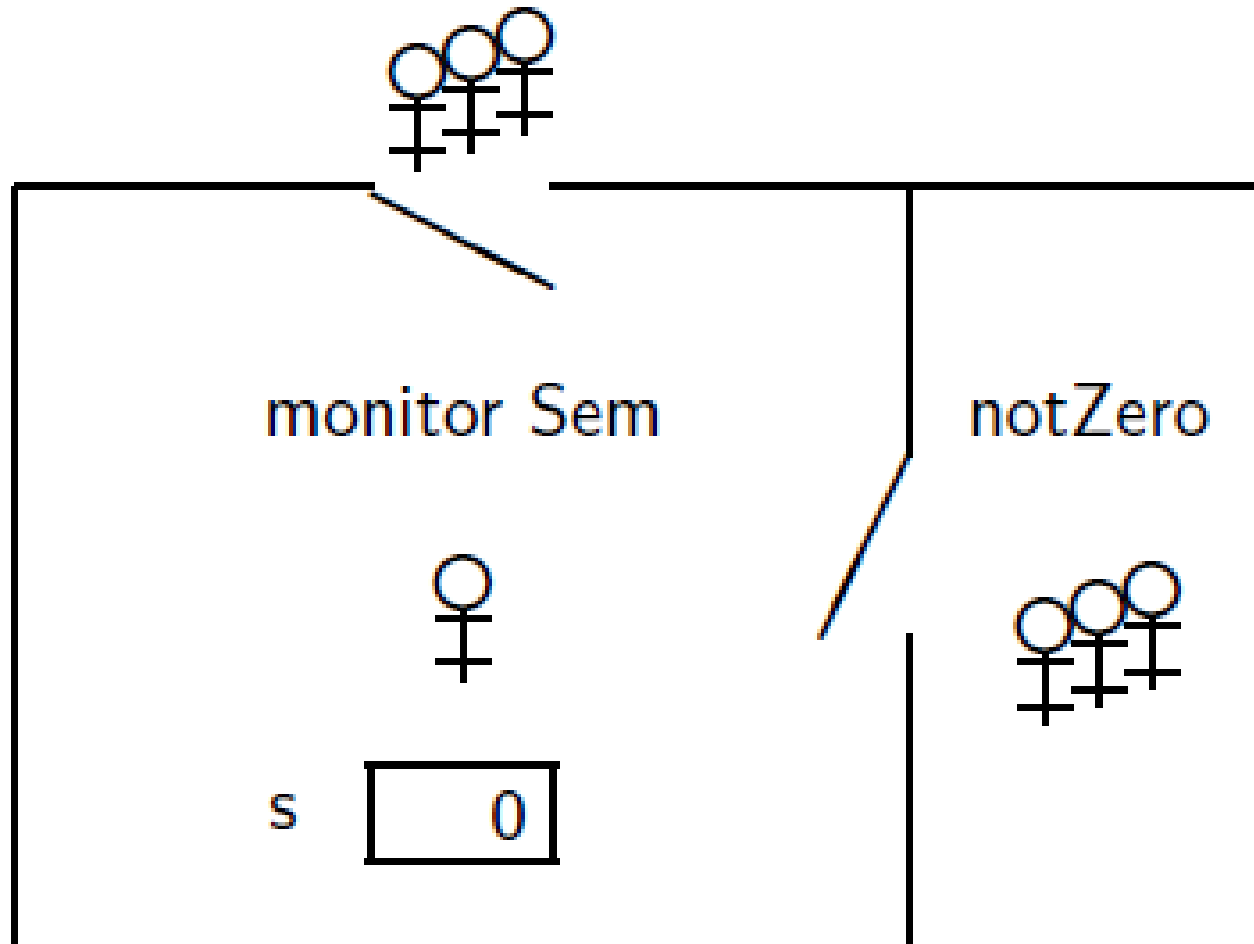
# Condition Variables

- Monitor implicitly enforces mutual exclusion to its variables
- Problems in concurrent programming have explicit synchronization requirements
- Two approaches in providing synchronization:
  - Required condition is named by an explicit *condition variable* (called an *event*). Ordinary boolean expressions are used to test the condition
  - Block directly on the expression and let the implementation implicitly unblock a process when the expression is true

# Condition Variables

- Condition variables provide a mechanism to wait for events (a “rendezvous point”)
  - Resource available, no more writers, etc.
- Condition variables support three operations:
  - Wait – release monitor lock, wait for C/V to be signaled
    - So condition variables have wait queues, too
  - Signal – wakeup one waiting thread
  - Broadcast – wakeup all waiting threads

# Simulating Semaphores



### Algorithm 7.2: Semaphore simulated with a monitor

monitor Sem

integer  $s \leftarrow k$

condition notZero

operation wait

if  $s = 0$

waitC(notZero)

$s \leftarrow s - 1$

operation signal

$s \leftarrow s + 1$

signalC(notZero)

**p**

loop forever

non-critical section

p1: Sem.wait

critical section

p2: Sem.signal

**q**

loop forever

non-critical section

q1: Sem.wait

critical section

q2: Sem.signal



# Simulating Semaphores

- Integer component of semaphore is stored in variable `s`
- Condition variable `cond` implements queue of blocked processes
- Condition variables are named with the condition you want to be true
- `waitC(notZero)` is read as “wait for `notZero` to be true” and `signalC(notZero)` is read as “signal the `notZero` is true”
- If the value of `s` is zero, the process executing `Sem.wait` executes the monitor statement `waitC(notZero)`. The process is said to be blocked on the condition

# Simulating Semaphores

- Process executing waitC blocks unconditionally as we assume that the condition has been tested for in the preceding if statement
- As monitor operation is atomic the value of the condition cannot change between testing its value and executing waitC
- When a process executes a Sem.signal operation, it unblocks the first process blocked on that condition
- With each condition variable is associated a FIFO queue of blocked processes

# Operations on Condition Variables

## waitC(cond)

append p to cond  
p.state  $\leftarrow$  blocked  
monitor.lock  $\leftarrow$  release

## empty(cond)

return cond = empty

## signalC(cond)

if cond  $\neq$  empty  
    remove head of  
    cond and assign to q  
    q.state  $\leftarrow$  ready

# Semaphore vs Monitor

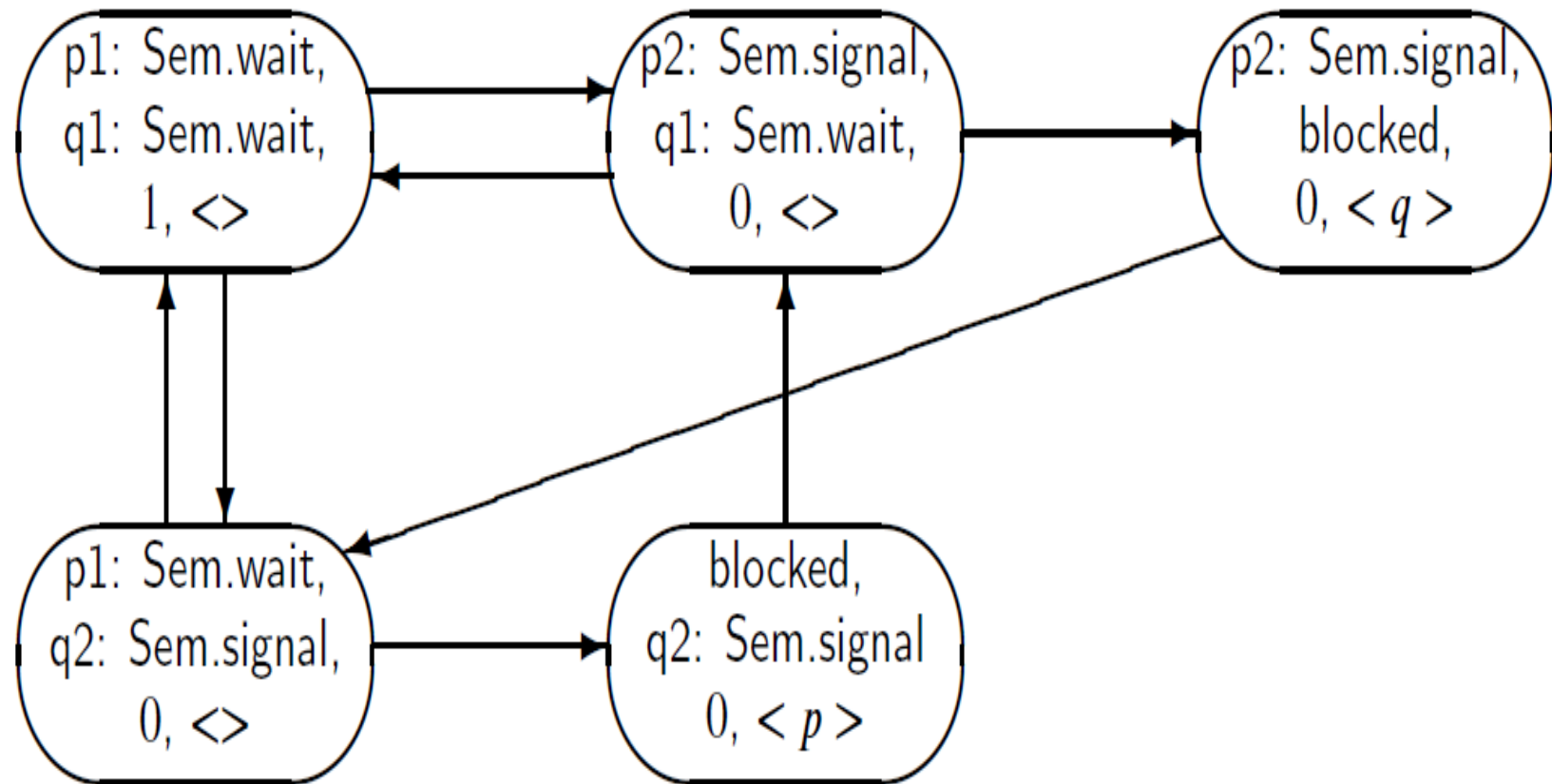
## Semaphore

- Wait may or may not block
- Signal always has an effect
- Signal unblocks an arbitrary blocked process
- A process unblocked by signal can resume execution immediately

## Monitor

- WaitC always blocks
- SignalC has no effect if queue is empty
- SignalC unblocks the process at the head of the queue
- A process unblocked by signalC must wait for the signalling process to leave monitor

# State diagram for the semaphore simulation



### Algorithm 7.3: Producer-consumer (finite buffer, monitor)

monitor PC

bufferType buffer  $\leftarrow$  empty

condition notEmpty

condition notFull

operation append(datatype V)

if buffer is full

waitC(notFull)

append(V, buffer)

signalC(notEmpty)

operation take()

datatype W

if buffer is empty

waitC(notEmpty)

W  $\leftarrow$  head(buffer)

signalC(notFull)

return W

# Producer Consumer

## Algorithm 7.3: Producer-consumer (finite buffer, monitor) (continued)

### producer

```
datatype D
loop forever
p1:   D ← produce
p2:   PC.append(D)
```

### consumer

```
datatype D
loop forever
q1:   D ← PC.take
q2:   consume(D)
```

# Emulating Semaphores using Monitors

monitor **Semaphore\_Emulation** is

S: Integer := S0;  
Not\_Zero : Condition;

Procedure **Semaphore\_Wait** is  
begin  
    if S=0 then Wait(Not\_Zero); end if;  
    S := S-1;  
end **Semaphore\_Wait**;

Procedure **Semaphore\_Signal** is  
begin  
    S := S+1; Signal(Not\_Zero);  
end **Semaphore\_Signal**;

end monitor



# Dining Philosophers Problem

## Algorithm 7.5: Dining philosophers with a monitor

monitor ForkMonitor

integer array[0..4] fork  $\leftarrow [2, \dots, 2]$

condition array[0..4] OKtoEat

operation takeForks(integer i)

if fork[i]  $\neq 2$

waitC(OKtoEat[i])

fork[i+1]  $\leftarrow$  fork[i+1] - 1

fork[i-1]  $\leftarrow$  fork[i-1] - 1

operation releaseForks(integer i)

fork[i+1]  $\leftarrow$  fork[i+1] + 1

fork[i-1]  $\leftarrow$  fork[i-1] + 1

if fork[i+1] = 2

signalC(OKtoEat[i+1])

if fork[i-1] = 2

signalC(OKtoEat[i-1])

# Dining Philosophers Problem

|   |
|---|
| Algorithm 7.5: Dining philosophers with a monitor (continued) |
| philosopher i   |
| loop forever  |
| p1: think   |
| p2: takeForks(i)  |
| p3: eat   |
| p4: releaseForks(i)   |

# Dining Philosophers Problem

- Monitor maintains an array of fork which counts the number of free forks available to each philosopher
- The takeForks operation waits on a condition variable until two forks are available
- It decrements the number of forks available to its neighbor before leaving the monitor
- After eating, a philosopher calls releaseForks which updates the array fork and checks if freeing these forks makes it possible to signal

# Readers Writers Problem

- Similar to Mutex problem where several processes are competing for access to a critical section
- Readers are processes which require to exclude writers but not other readers
- Writers are processes which require to exclude both but not readers and other writers
- Several processes can read data concurrently, but writing or modifying data must be done under Mutex to ensure consistency of Data

### Algorithm 7.4: Readers and writers with a monitor

monitor RW

integer readers  $\leftarrow 0$

integer writers  $\leftarrow 0$

condition OKtoRead, OKtoWrite

operation StartRead

if writers  $\neq 0$  or not empty(OKtoWrite)

waitC(OKtoRead)

readers  $\leftarrow$  readers + 1

signalC(OKtoRead)

operation EndRead

readers  $\leftarrow$  readers - 1

if readers = 0

signalC(OKtoWrite)

### Algorithm 7.4: Readers and writers with a monitor (continued)

operation StartWrite

if writers  $\neq$  0 or readers  $\neq$  0  
    waitC(OKtoWrite)  
writers  $\leftarrow$  writers + 1

operation EndWrite

writers  $\leftarrow$  writers - 1  
if empty(OKtoRead)  
    then signalC(OKtoWrite)  
    else signalC(OKtoRead)

**reader**

p1: RW.StartRead  
p2: read the database  
p3: RW.EndRead

**writer**

q1: RW.StartWrite  
q2: write to the database  
q3: RW.EndWrite

# Readers Writers Problem

- Monitor uses 4 variables
  - Readers: The number of threads currently reading the database after successfully executing StartRead but before executing EndRead
  - writers: The number of writers currently writing to the database after successfully executing StartWrite but before executing EndWrite
  - OktoRead: A condition variable for blocking readers until it is “OK to read”
  - OktoWrite: A condition variable for blocking writers until it is “OK to write”

# What's wrong with Semaphores

- They are essentially shared global variables.
- There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
- Access to semaphores can come from anywhere in a program.
- They serve two purposes, mutual exclusion and scheduling constraints.
- There is no control or guarantee of proper usage.



# Monitor – A Formal Definition

- 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.
- The lock also provides mutual exclusion for shared data.
- 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.

# Monitor Operations

- Encapsulates the shared data you want to protect.
- Acquires the mutex at the start.
- Operates on the shared data.
- Temporarily releases the mutex if it can't complete.
- Reacquires the mutex when it can continue.
- Releases the mutex at the end.