



# 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
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

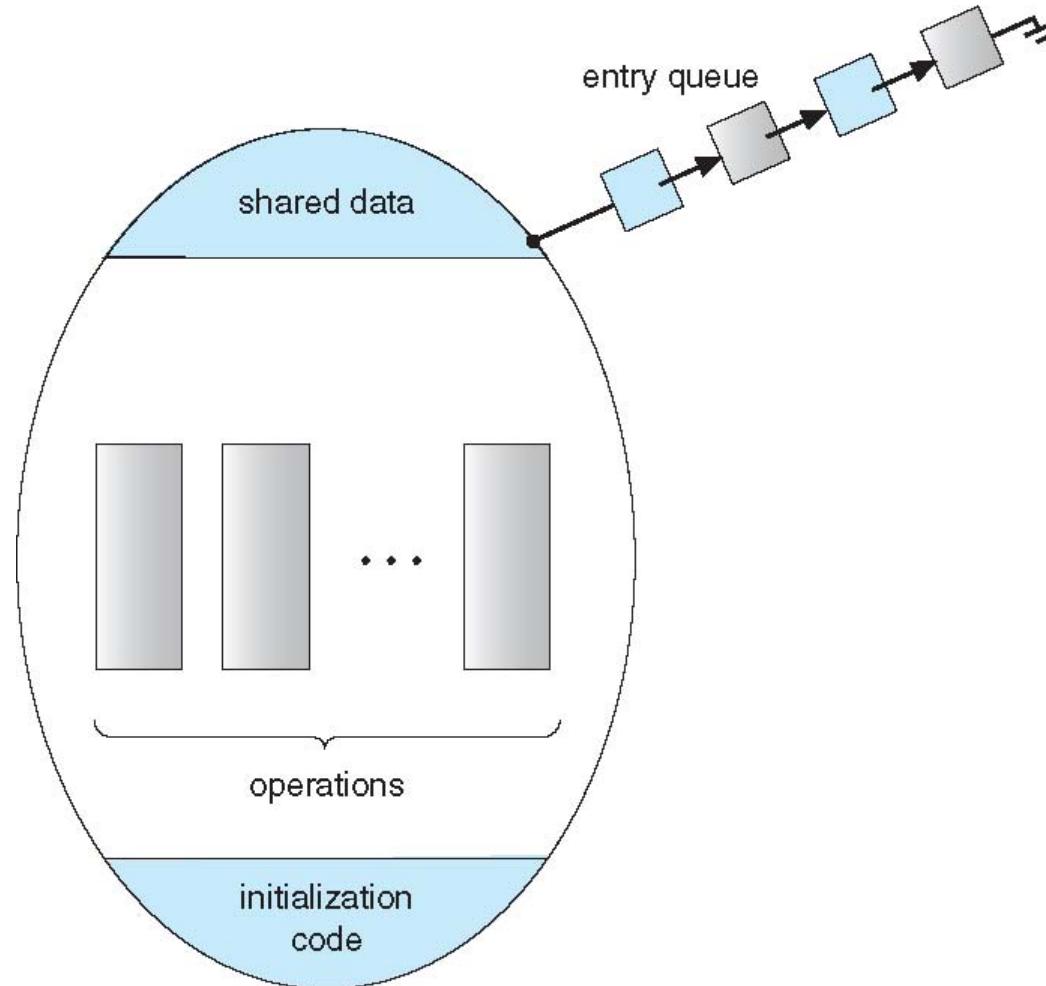
    procedure Pn (...) { ..... }

    Initialization code (...) { ... }
}
```





# Schematic view of a Monitor





# Condition Variables

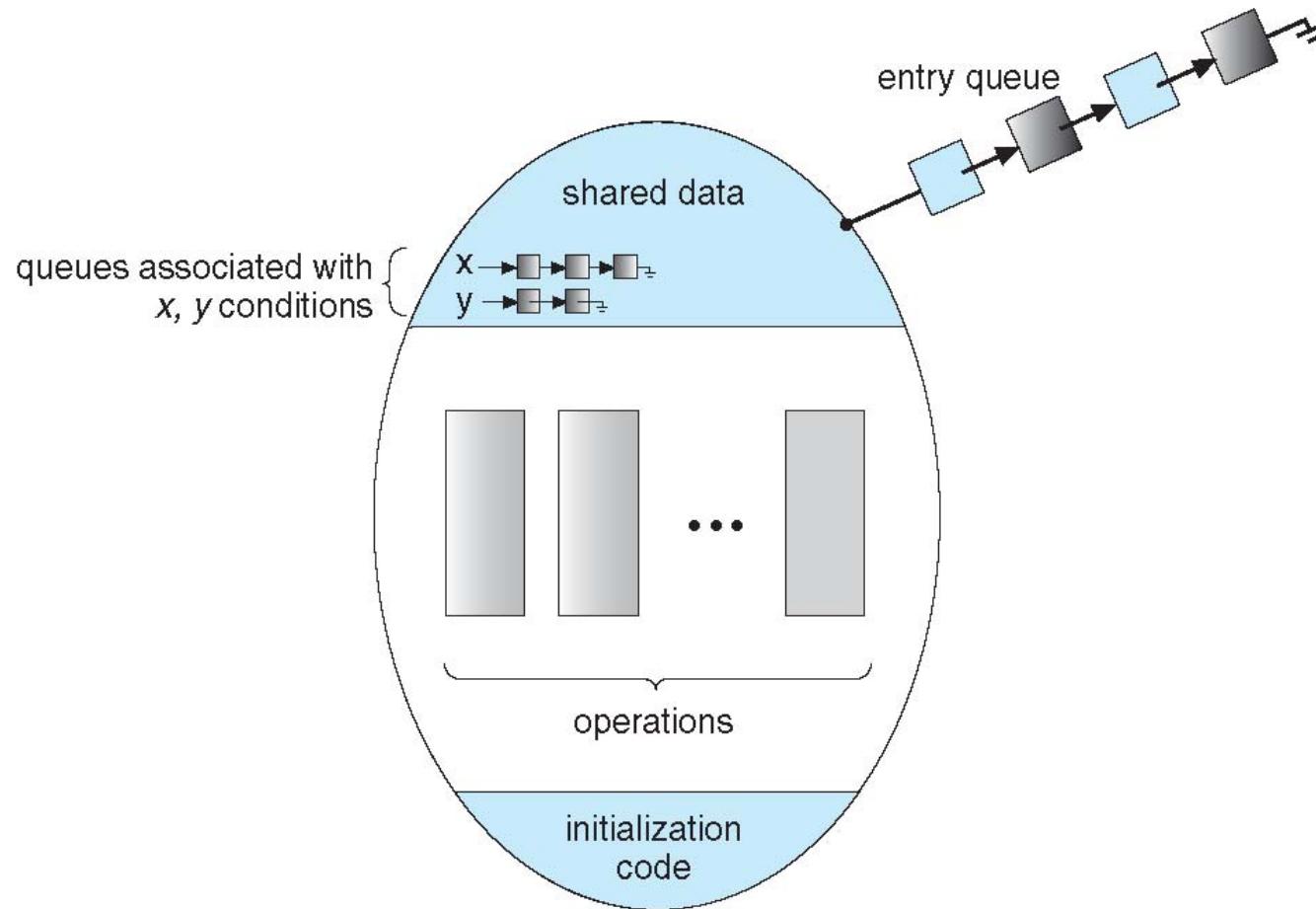
---

- **condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - ▶ If no **x.wait()** on the variable, then it has no effect on the variable





# Monitor with Condition Variables





# Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - ▶ P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java





# Solution to Dining Philosophers

---

- Each philosopher  $i \{0 \text{ to } 4\}$  invokes the operations **pickup ()** and **putdown ()** in the following sequence:

**DiningPhilosophers.pickup(i) ;**

**EAT**

**DiningPhilosophers.putdown(i) ;**

- No deadlock, but starvation is possible





# Solution to Dining Philosophers (Cont.)

```
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 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;  
}
```





# 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





# Resuming Processes within a Monitor

- If several processes queued on condition  $x$ , and  $x.signal()$  executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form  $x.wait(c)$ 
  - Where  $c$  is **priority number**
  - Process with lowest number (highest priority) is scheduled next





# Linux Synchronization

---

## ■ Linux:

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive

## ■ Linux provides:

- Semaphores
- atomic integers
- spinlocks
- reader-writer versions of both

## ■ On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption





# Pthreads Synchronization

---

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks





# Alternative Approaches

---

- Transactional Memory
- OpenMP
- Functional Programming Languages



# End of Chapter 5

