

Jin-Soo Kim  
([jinsoo.kim@snu.ac.kr](mailto:jinsoo.kim@snu.ac.kr))

Systems Software &  
Architecture Lab.

Seoul National University

Fall 2023

# Monitors



# Monitors (I)

- Monitor is a programming language construct that supports controlled access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Allows the safe sharing of an abstract data type among concurrent processes
- A monitor is a software module that encapsulates:
  - Shared data structures
  - \_\_\_\_\_ that operate on the shared data
  - Synchronization between concurrent processes that invoke those procedures
- Monitor protects the data from unstructured access
  - Guarantees only access data through procedures, hence in legitimate ways

# Monitors (2)

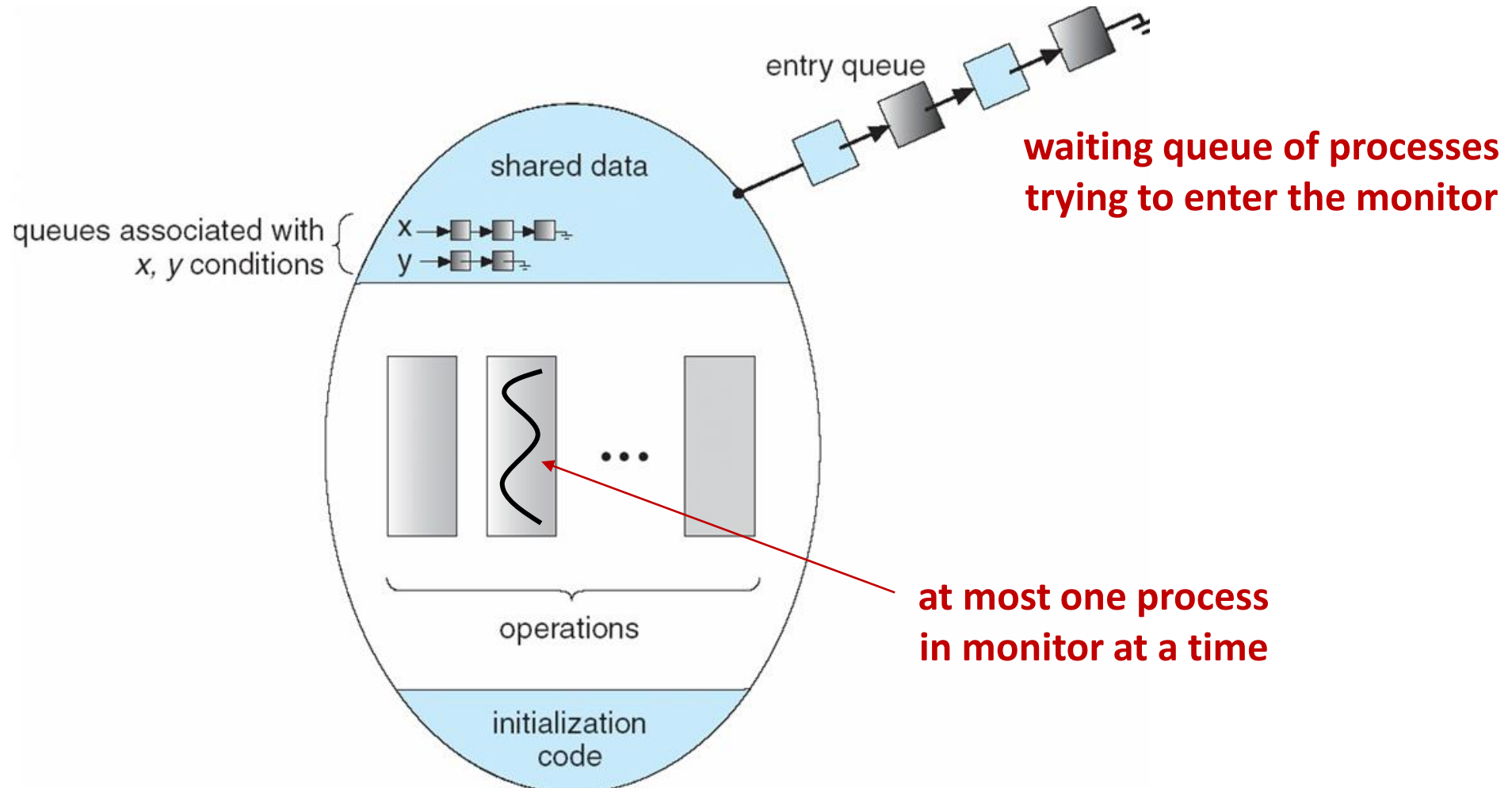
## ■ Mutual exclusion

- Only one process can be executing inside at any time
  - Thus, synchronization implicitly associated with monitor
- If a second process tries to enter a monitor procedure, it blocks until the first has left the monitor
  - More restrictive than semaphores, but easier to use most of the time

## ■ Condition variables

- Once inside, a process may discover it can't continue, and may wish to sleep, or allow some other waiting process to continue
- Condition variables are provided within monitor
  - Processes can wait or signal others to continue
  - Can only be accessed from inside monitor

# Monitors (3)



Source: A. Silberschatz et al., Operating System Concepts, 2008.

# Condition Variables

- Provide a mechanism to wait for events (a "rendezvous point")
- `wait(c)`
  - Release monitor lock, so somebody else can get in
  - Wait for somebody else to signal condition
  - Thus, condition variables have wait queues
- `signal(c)`
  - Wake up at most one waiting process
  - If no waiting processes, signal is lost
  - This is different from semaphores: no history!
- `broadcast(c)`
  - Wake up all waiting processes

# Bounded Buffer with Monitors

```
Monitor bounded_buffer {
    buffer resources[N];
    condition not_full, not_empty;

    procedure add_entry(resource x) {
        while (array "resources" is full)
            wait(not_full);
        add "x" to array "resources";
        signal(not_empty);
    }

    procedure remove_entry(resource *x) {
        while (array "resources" is empty)
            wait(not_empty);
        *x = get resource from array "resources";
        signal(not_full);
    }
}
```

# Monitors Semantics

## ■ Hoare monitors

- `signal(c)` immediately switches from the caller to a waiting thread, blocking the caller
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling

## ■ Mesa monitors

- `signal(c)` places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter runs again
  - Being woken up is only a hint that something has changed
  - Must recheck conditional case

# Monitors Semantics: Comparison

## Hoare monitors

```
if (notReady)
    wait(c);
```

## Mesa monitors

```
while (notReady)
    wait(c);
```

- Mesa monitors easier to use
  - More efficient
  - Fewer switches
  - Directly supports `broadcast()`
- Hoare monitors leave less to chance
  - When wake up, condition guaranteed to be what you expect



# Monitors using Semaphores

## ■ monitors

```
Semaphore mutex = 1;
Semaphore next = 0;
int next_count = 0;
struct condition {
    Semaphore sem;
    int count;
} x = {0, 0};

procedure F() {
    wait(mutex);
    ...
    Body of F
    ...
    if (next_count)
        signal(next);
    else
        signal(mutex);
}
```

```
procedure cond_wait(x) {
    x.count++;
    if (next_count)
        signal(next);
    else
        signal(mutex);
    wait(x.sem);
    x.count--;
}

procedure cond_signal(x) {
    if (x.count) {
        next_count++;
        signal(x.sem);
        wait(next);
        next_count--;
    }
}
```

# Monitors vs. Semaphores

- Condition variables do not have any \_\_\_\_\_, but semaphores do
- On a condition variable `signal()`, if no one is waiting, the signal is a no-op
  - If a thread then does `wait()` on the condition variable, it waits
- On a semaphore `signal()`, if no one is waiting, the value of the semaphore is increased
  - If a thread then does `wait()` on the semaphore, the value is decreased and the thread continues