Operating System

Ch06: Synchronization tool

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

Synchronization

- Threads cooperate in multithreaded programs
  - ✓ To share resources, access shared data structures
  - ✓ Also, to coordinate their execution

- For correctness, we have to control this cooperation
  - ✓ Must assume threads interleave executions arbitrarily and at different rates
    - ➢ Scheduling is not under application writers' control
  - ✓ We control cooperation using synchronization
    - ➢ Enables us to restrict the interleaving of execution
  - ✓ (Note) This also applies to processes, not just threads
    - ➢ And it also applies across machines in a distributed system

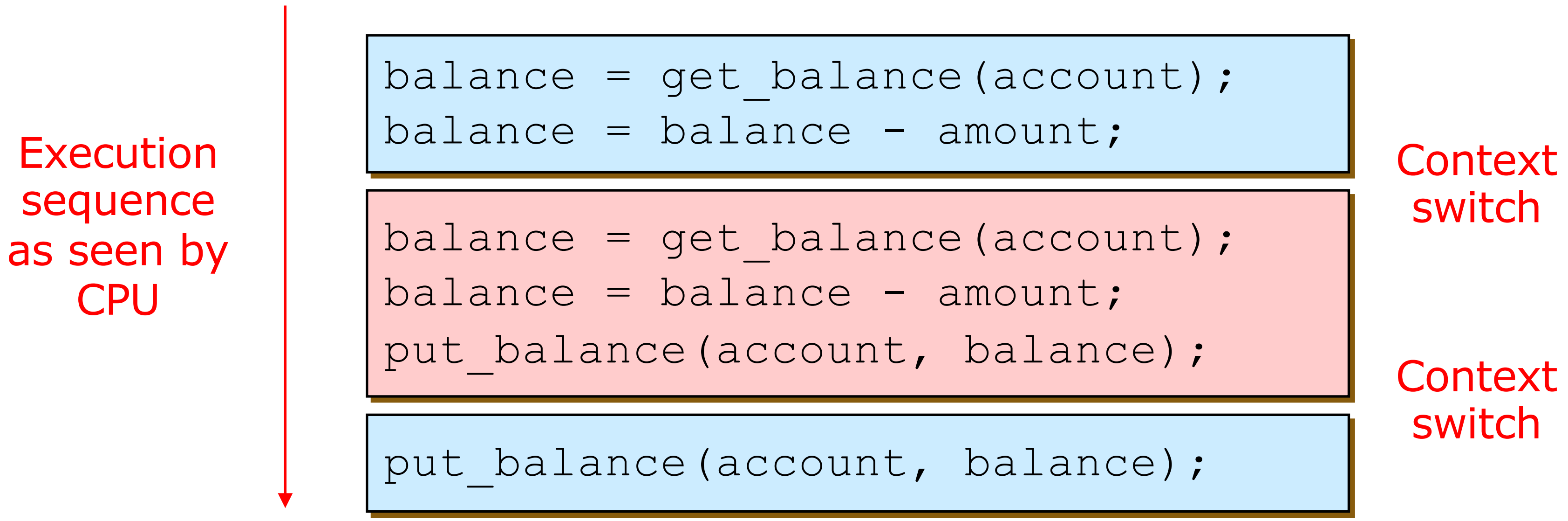
An Example

- Withdraw money from a bank account
  - ✓ Suppose you and your girl(boy) friend share a bank account with a balance of 1,000,000won
  - ✓ What happens if both go to separate ATM machines, and simultaneously withdraw 100,000won from the account?

```
int withdraw(account, amount)
{
   balance = get_balance(account);
   balance = balance - amount;
   put_balance(account, balance);
   return balance;
}
```

An Example

- Interleaved schedules
  - ✓ Represent the situation by creating a separate thread for each person to do the withdrawals
  - ✓ The execution of the two threads can be interleaved, assuming preemptive scheduling:



```
balance = get_balance(account);
balance = balance - amount;
```

```
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

Another Example

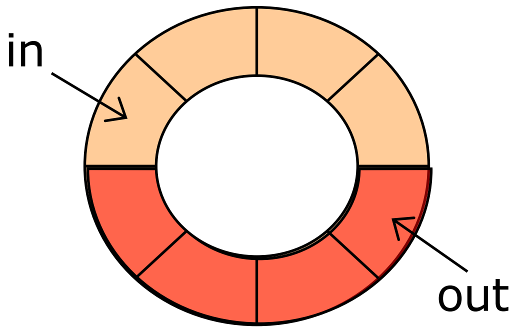
- Cooperating processes
  - ✓ Bounded buffer (Producer-Consumer)

Producer

```
void producer(data)
{

 while (count==N)
        ;
 buffer[in] = data;
 in = (in+1) % N;
 count++;

}
```

int count;

struct item buffer[N];
int in, out;



Consumer

```
void consumer(data)
{

 while (count==0)
        ;
 data = buffer[out];
 out = (out+1) % N;
 count--;

}
```

Another Example

- Producer-Consumer
  - ✓ The statement "**`count++`**" may be implemented in machine language as:

```
register1 = count
register1 = register + 1
count     = register1
```

  - ✓ The statement "**`count--`**" may be implemented as:

```
register2 = count
register2 = register - 1
count     = register2
```

  - ✓ Assume **`count`** is initially 5. One interleaving of statements is:

```
producer: register1 = count         (register1 = 5)
producer: register1 = register1 + 1 (register1 = 6)
consumer: register2 = count         (register2 = 5)
consumer: register2 = register2 – 1 (register2 = 4)
producer: count     = register1     (counter   = 6)
consumer: count     = register2     (counter   = 4)
```

  - ✓ The value of **`count`** may be either 4 or 6, where the correct result should be 5

Synchronization Problem

- Problem
  - ✓ Two concurrent threads (or processes) access a shared resource without any synchronization
  - ✓ Creates a race condition
    - ➢ The situation where several processes access and manipulate shared data concurrently
    - ➢ The result is non-deterministic and depends on timing
  - ✓ We need mechanisms for controlling access to shared resources in the face of concurrency
    - ➢ So that we can reason about the operation of programs
  - ✓ Synchronization is necessary for any shared data structure
    - ➢ buffers, queues, lists, etc.
  - ✓ Critical section problem

Requirements for Synchronization Tools

1. Mutual Exclusion
   - ✓ If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. Progress
   - ✓ If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes 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 after a process has made a request to enter its critical section and before that request is granted
   - ✓ Assume that each process executes at a nonzero speed
   - ✓ No assumption concerning relative speed of the *n* processes

Synchronization Tools

- Locks (low level mechanism)
  - ✓ Very primitive, minimal semantics, used to build others in OS
- Mutex lock (blocked lock)
- Semaphores
  - ✓ Basic, easy to get the hang of, hard to program with
- Monitors
  - ✓ High-level, requires language support, implicit operations
  - ✓ Easy to program with: Java “synchronized”
- Messages (in distributed systems)
  - ✓ Simple model of communication and synchronization based on (atomic) transfer of data across a channel

Locks

- A lock is an object (in memory) that provides the following two operations:
  - ✓ lock(): wait until lock is free, then grab it
  - ✓ unlock(): unlock, and wake up any thread waiting in lock()

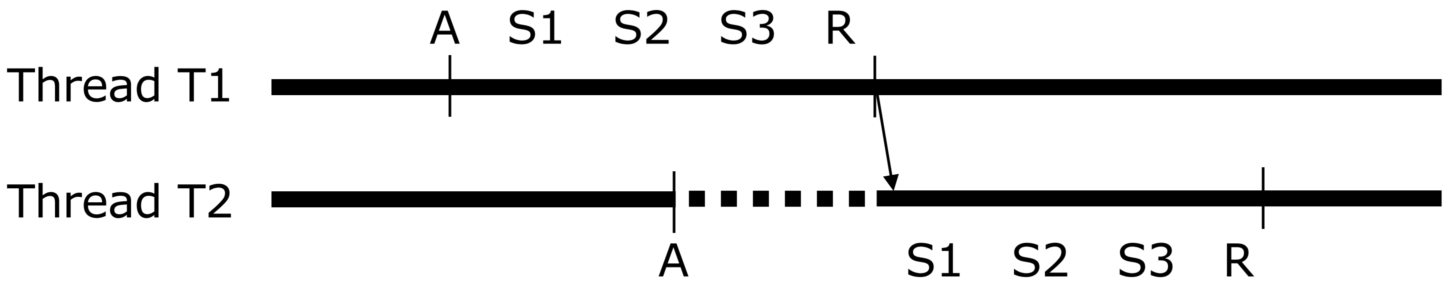
- Using locks
  - ✓ Lock is initially free
  - ✓ Call lock() before entering a critical section, and unlock() after leaving it
  - ✓ Between lock() and unlock(), the thread holds the lock
  - ✓ lock() does not return until the caller holds the lock
  - ✓ At most one thread can hold a lock at a time

- Locks can
  - ✓ spin (a spinlock: low level mechanism) or
  - ✓ block (a mutex: high level mechanism)

Using Locks

```
     int withdraw(account, amount)
     {
A        lock(lock);
S1       balance = get_balance(account);
S2       balance = balance - amount;
S3       put_balance(account, balance);
R        unlock(lock);
         return balance;
     }
```

Critical Section



Implementing Locks

- An initial attempt

```
struct lock { int held = 0; }

void lock(struct lock *l)   {
    while (l->held)
        ;
    l->held = 1;
}
void unlock(struct lock *l)    {
    l->held = 0;
}
```

The caller "busy-waits", or spins for locks to be released, hence spinlocks

✓ Does this work?

Implementing Locks

- Problem
  - ✓ Implementation of locks has a critical section, too!
    - ➢ The lock/unlock must be atomic
    - ➢ A recursion, huh?
  - ✓ Atomic operation
    - ➢ Executes as though it could not be interrupted
    - ➢ Code that executes “all or nothing”

- Solutions
  - ✓ Software-only algorithms
    - ➢ Algorithm 1, 2, 3 for two processes
    - ➢ Bakery algorithm for more than two processes
  - ✓ Hardware atomic instructions
    - ➢ Test-and-set, compare-and-swap, etc.
  - ✓ Disable/re-enable interrupts
    - ➢ To prevent context switches

Hardware Atomic Instructions

- Test-and-Set
  - ✓ Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {
  boolean rv = target;
  target = true;

  return rv;
}
```

Hardware Atomic Instructions

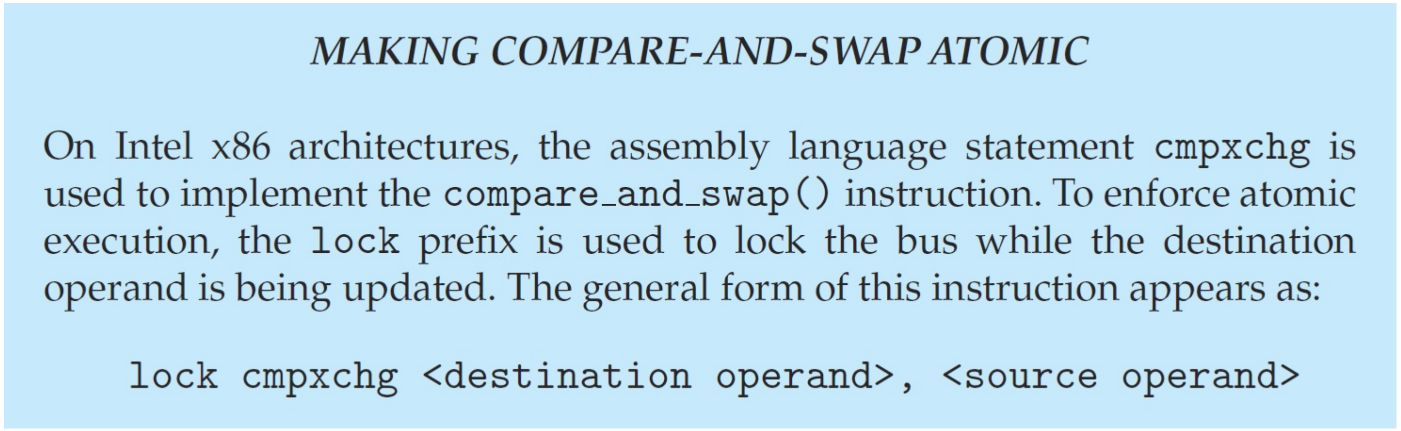
- Test-and-Set

```
struct lock { int held = 0; }


void lock(struct lock *l)  {
   while (TestAndSet(l->held))
        ;
}

void unlock(struct lock *l)   {
   l->held = 0;
}
```

Hardware Atomic Instructions

- Compare-and-Swap

```
boolean CompareAndSwap(boolean &a, boolean &b) {
  boolean rv = a;
  a = b;
  b = rv;
  return rv;
}
```



MAKING COMPARE-AND-SWAP ATOMIC

On Intel x86 architectures, the assembly language statement `cmpxchg` is used to implement the `compare_and_swap()` instruction. To enforce atomic execution, the `lock` prefix is used to lock the bus while the destination operand is being updated. The general form of this instruction appears as:

```
lock cmpxchg <destination operand>, <source operand>
```

Hardware Atomic Instructions

- Compare-and-Swap

```
struct lock { int held = 0; }


void lock(struct lock *l)  {
    key = true;
    while (CompareAndSwap(l->held, key))
          ;
}

void unlock(struct lock *l)   {
    l->held = 0;
}
```

Problems with Spinlocks

- Horribly wasteful !
  - ✓ If a thread is spinning on a lock, the thread holding the lock cannot make progress
  - ✓ the longer the critical section, the longer the spin
  - ✓ Greater the chances for lock holder to be interrupted
- How did the lock holder yield the CPU in the first place?
  - ✓ Lock holder calls yield() or sleep()
  - ✓ Involuntary context switch
- Only want to use spinlock as primitives to build higher-level synchronization constructs

Disabling Interrupts

- Implementing locks by disabling interrupts

```
void lock(struct lock *l)   {
    cli();            // disable interrupts;
}
void unlock(struct lock *l)   {
    sti();            // enable interrupts;
}
```

- ✓ Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- ✓ There is no state associate with the lock
- ✓ Can two threads disable interrupts simultaneously?

Disabling Interrupts

- What's wrong?
  - ✓ Only available to kernel
    - ➢ Why not have the OS support these as system calls?
  - ✓ Insufficient on a multiprocessor
    - ➢ Back to atomic instructions
  - ✓ What if the critical section is long?
    - ➢ Can miss or delay important events (e.g., timer, I/O)
  - ✓ Like spinlocks, only use to implement higher-level synchronization primitives

Implementing Locks (1)

- An initial attempt

```
struct lock { int held = 0; }

void lock(struct lock *l)   {
   while (l->held)
       ;
   l->held = 1;
}
void unlock(struct lock *l)    {
   l->held = 0;
}
```

  - ✓ This doesn't work

Implementing Locks (2)

- Disable/Re-enable interrupts

```
void lock(struct lock *l)   {
   cli();
}
void unlock(struct lock *l)    {
   sti();
}
```

  - ✓ Primitive lock for OS in single processor

Implementing Locks (3)

- Hardware atomic instructions

```
struct lock { int held = 0; }

void lock(struct lock *l)   {
   while (TestAndSet(l->held))
       ;
}
void unlock(struct lock *l)    {
   l->held = 0;
}
```

  - ✓ Primitive lock for OS in multi-processors

High-level Synchronization

- Motivation
  - ✓ Spinlocks and disabling interrupts are useful only for very short and simple critical sections
    - ➢ Wasteful otherwise
    - ➢ These primitives are “primitive” – don’t do anything besides mutual exclusion
  - ✓ Need higher-level synchronization primitives that
    - ➢ Block waiters
    - ➢ Leave interrupts enabled within the critical section
  - ✓ Two common high-level primitives:
    - ➢ Semaphores: binary (mutex) and counting
    - ➢ Monitors: mutexes and condition variables
  - ✓ We’ll use our “atomic” locks as primitives to implement them

Semaphores

- Semaphore
  - ✓ A counter used to provide access to a shared data object for multiple processes or threads
  - ✓ Two operations
    - ➢ wait or P
    - ➢ signal or V

- Synchronization procedure using semaphores
  - ✓ Test the semaphore that controls the resource
  - ✓ If the value of the semaphore is positive, the process can use the resource
    - ➢ The process decrements the semaphore value by 1, indicating that it has used on unit of the resource
  - ✓ If the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0
    - ➢ When the process wakes up, it returns to above step

Semaphore Usage

- Critical section synchronization using semaphores

```
Semaphore mutex = 1;

int withdraw(account, amount)
{
    wait(mutex);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    signal(mutex);
    return balance;
}
```

Semaphore Usage

- General synchronization using semaphores
  - ✓ Execute *B* in P_j only after *A* executed in P_i
  - ✓ Use semaphore *flag* initialized to 0
  - ✓ Code:

```
        Semaphore flag = 0;

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

Mutex Locks

- Binary semaphores can be implemented in the form of mutually exclusive lock (i.e. mutex lock)
  - ✓ Semaphore

```
Semaphore S = 1;
wait(S);
/* Critical Section */
signal(S);
```

  - ✓ Mutex lock

```
MutexLock L;
lock(L);
/* Critical Section */
unlock(L);
```

- Cf) Primitive lock for OS
  - ✓ Spinlock or disabling interrupts

Mutex Locks

- Synchronization using mutex locks

```
MutexLock mutex;

int withdraw(account, amount)
{
    lock(mutex);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    unlock (mutex);
    return balance;
}
```

Two Types of Semaphores

- *Binary* semaphore
  - ✓ integer value can range only between 0 and 1
  - ✓ can be simpler to implement
- *Counting* semaphore
  - ✓ integer value can range over an unrestricted domain
- Can implement a counting semaphore *S* as a binary semaphore

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
   int             value;
   struct process *L;
} semaphore;
```

- Assume two simple operations:
  - ✓ `block` suspends the process that invokes it
  - ✓ `wakeup(P)` resumes the execution of a blocked process ***P***

Semaphore Implementation

- No busy waiting

```
wait(semaphore *S) {
        lock(lock); S->value--; unlock(lock);
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
signal(semaphore *S) {
        lock(lock); S->value++; unlock(lock);
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Deadlock and Starvation

- **Deadlock**
  - ✓ two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let *S* and *Q* be two semaphores initialized to 1

```
        P0                  P1
    wait(S);            wait(Q);
    wait(Q);            wait(S);
       .                   .
       .                   .
       .                   .
   signal(S);          signal(Q);
   signal(Q);          signal(S);
```

- **Starvation** or indefinite blocking
  - ✓ A process may never be removed from the semaphore queue in which it is suspended

Problems with Semaphores

- Drawbacks
  - ✓ They are essentially shared global variables
    - ➢ Can be accessed from anywhere (bad software engineering)
  - ✓ There is no connection between the semaphore and the data being controlled by it
  - ✓ Used for both critical sections (mutual exclusion) and for coordination (scheduling)
  - ✓ No control over their use, no guarantee of proper usage

- Thus, hard to use and prone to bugs
  - ✓ Incorrect use of semaphore operations
    - ➢ signal(mutex) … wait(mutex) / wait(mutex) … wait(mutex) / …
  - ✓ Another approach: use programming language support
    - ➢ Critical region
    - ➢ Monitor

Monitors

- 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
  - ✓ procedures that operate on the shared data
  - ✓ synchronization between concurrent processes that invoke those procedures

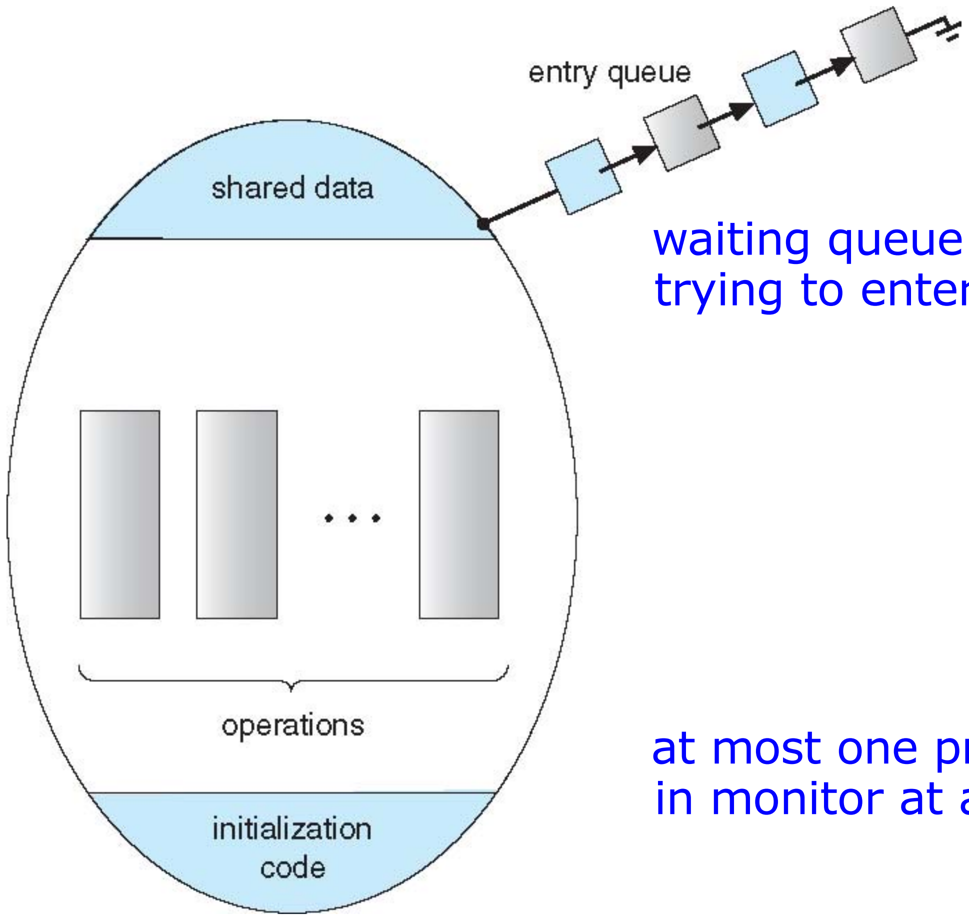
- Monitor protects the data from unstructured access
  - ✓ guarantees only access data through procedures, hence in legitimate ways

Monitors

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

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

Schematic View of a Monitor



waiting queue of processes trying to enter the monitor

at most one process in monitor at a time

Condition Variables

- To allow a process to wait within the monitor, a **condition** variable provides a mechanism to wait for events (a “rendezvous point”)

```
                 condition x, y;
```

- Condition variable can only be used with the operations **wait** and **signal**
  - ✓ The operation

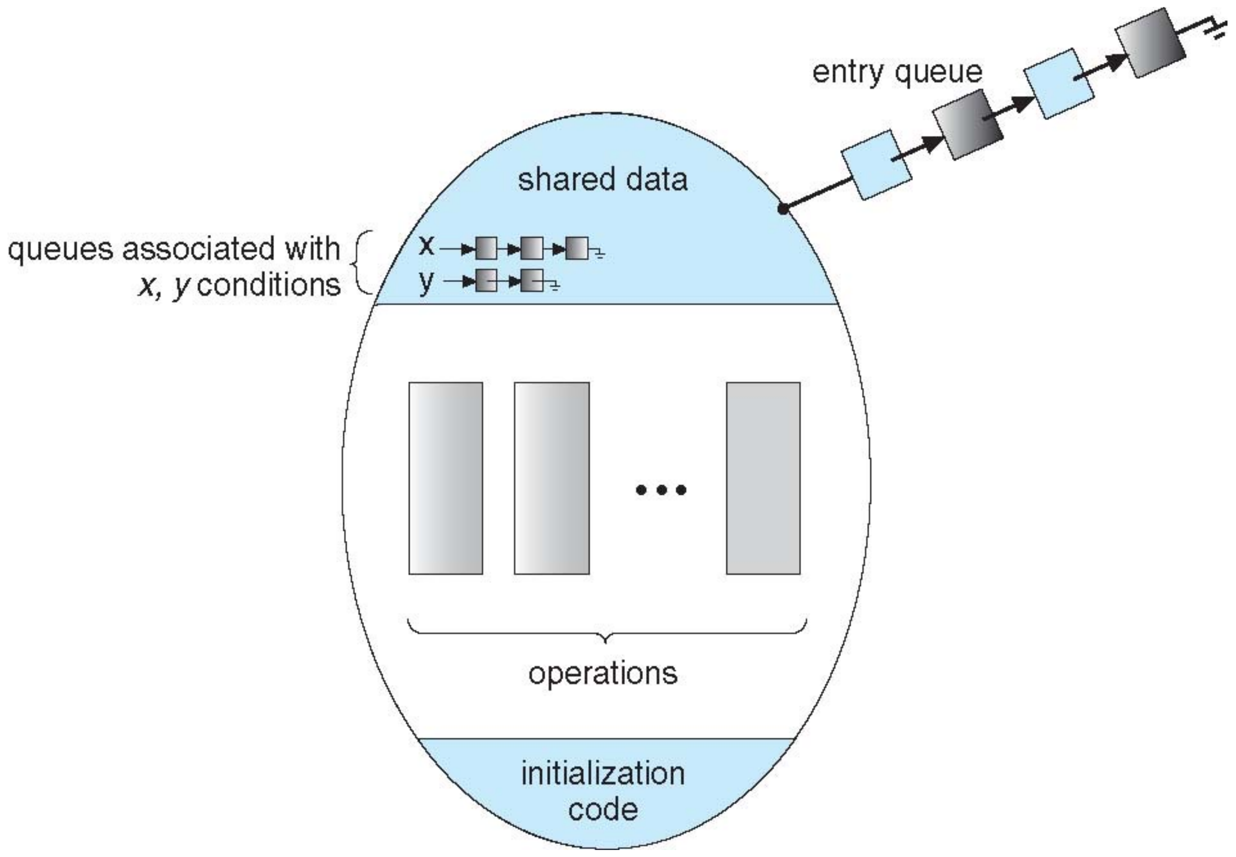
    ```
              x.wait();
    ```

    means that the process invoking this operation is suspended until another process invokes

    ```
              x.signal();
    ```

  - ✓ The **`x.signal`** operation resumes exactly one suspended process
  - ✓ If no process is suspended, then the **signal** operation has no effect

Monitor With Condition Variables



Comparison: Monitors and Semaphores

- Condition variables do not have any history, but semaphores do
  - ✓ On a condition variable signal(), if no one is waiting , the signal is a no-op

    (If a thread then does a condition variable wait(), it waits)
  - ✓ On a semaphore signal(), if no one is waiting, the value of the semaphore is increased

    (If a thread then does a semaphore wait(), the value is decreased and the thread continues)

Thank You!
Q&A