L19-SEMAPHORES AND MONITORS

Objectives of Process Synchronization

- To introduce the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- **To present both software and hardware solutions of the critical-section problem**
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Critical Section

- Each process must ask permission to enter **critical section** in **entry section**, may follow **critical section** with **exit section**, then **remainder section**
- General structure of process **P**

```
do {
    [entry section]
        critical section
    [exit section]
        remainder section
} while (true);
```

```
do {
 while (turn == j);
   critical section
 turn = j;
   remainder section
 } while (true);
```

Mutual Exclusion :: Progress :: Bounded Waiting

Semaphore

- Synchronization tool for processes to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

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

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

Semaphore Usage

- **Binary semaphore** – value can range only between 0 and 1
  - Represents single access to a resource
- **Counting semaphore** – integer value (unrestricted range)
  - Represents a resource with N concurrent access
- Consider P_1 and P_2 that require S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

P1:

```
S1;
signal(synch);
```

P2:

```
wait(synch);
S2;
```

Semaphore Implementation

- With each semaphore there is an associated waiting queue
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation

- Semaphore uses two atomic operations
- Each semaphore has a queue of waiting processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- When signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread

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

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

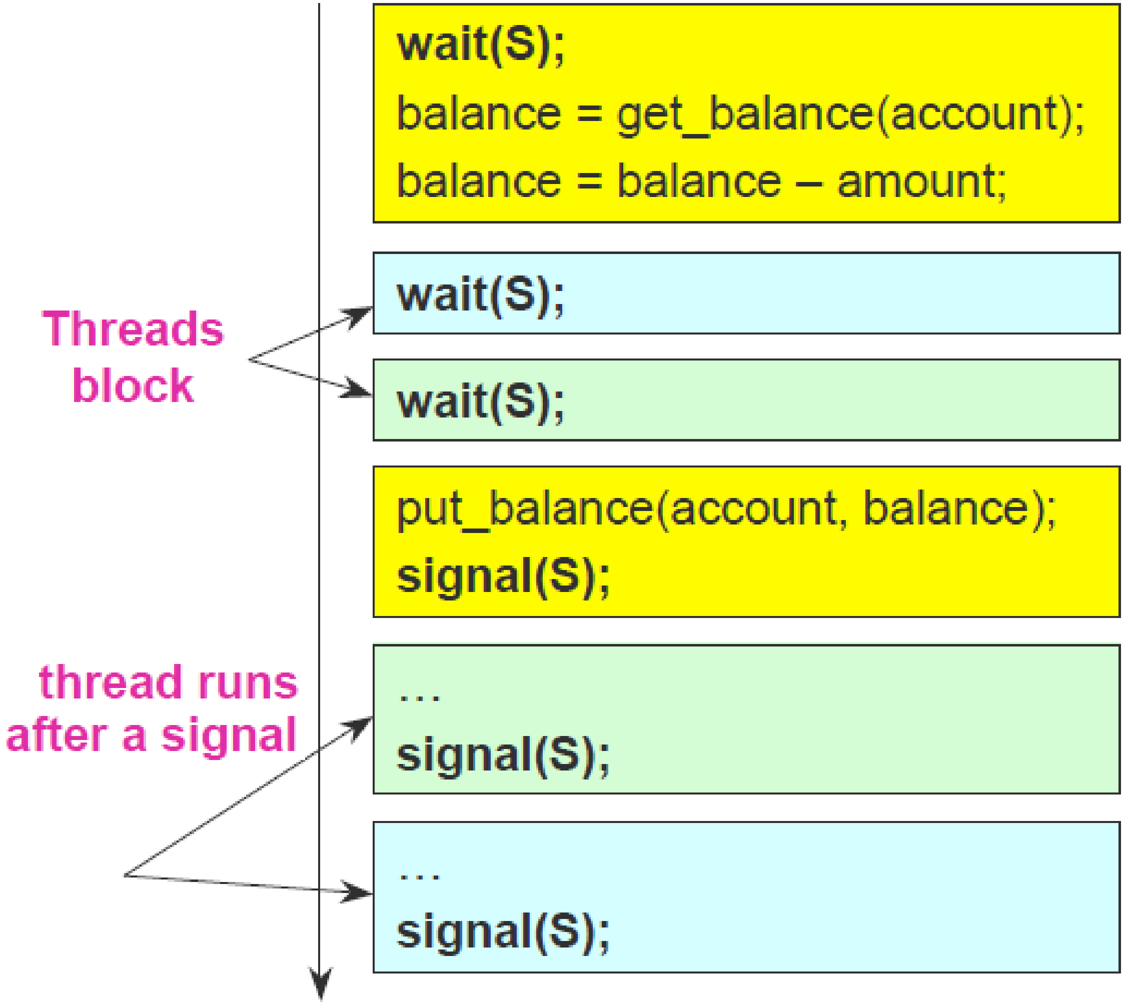
Semaphore Implementation

```
wait(semaphore *S)
{   S->value--;
    if (S->value < 0)
    {
        add this process to
        S->list;
        block();
    }
}
```

```
signal(semaphore *S)
{   S->value++;
    if (S->value <= 0)
    {
        remove a process P
      from S->list;
        wakeup(P);
    }
}
```

Semaphore Implementation

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```



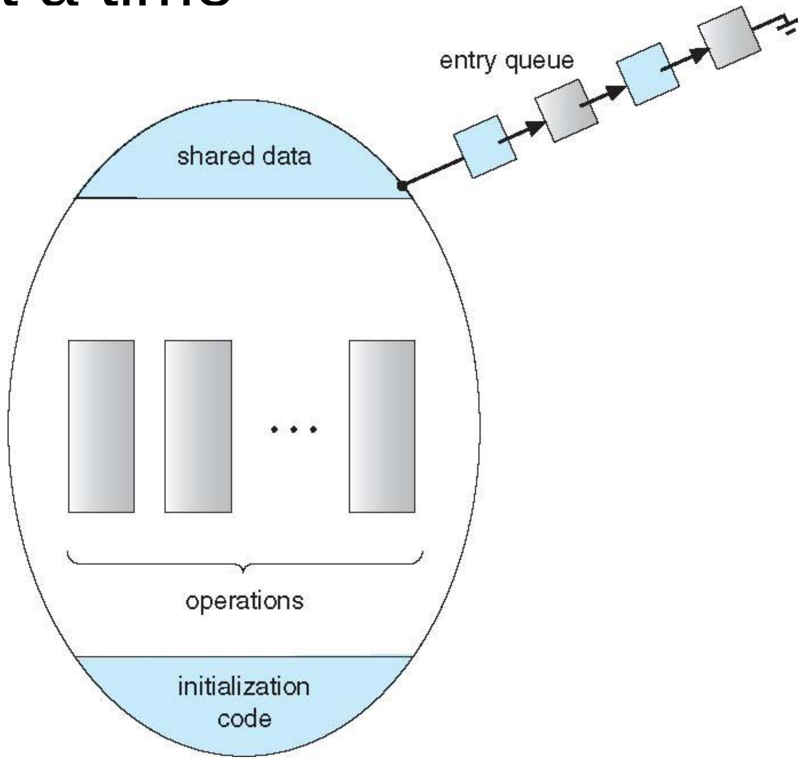
Monitors

- A monitor is a programming language construct that controls access to shared data
- Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent procedure invocations
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

Monitors

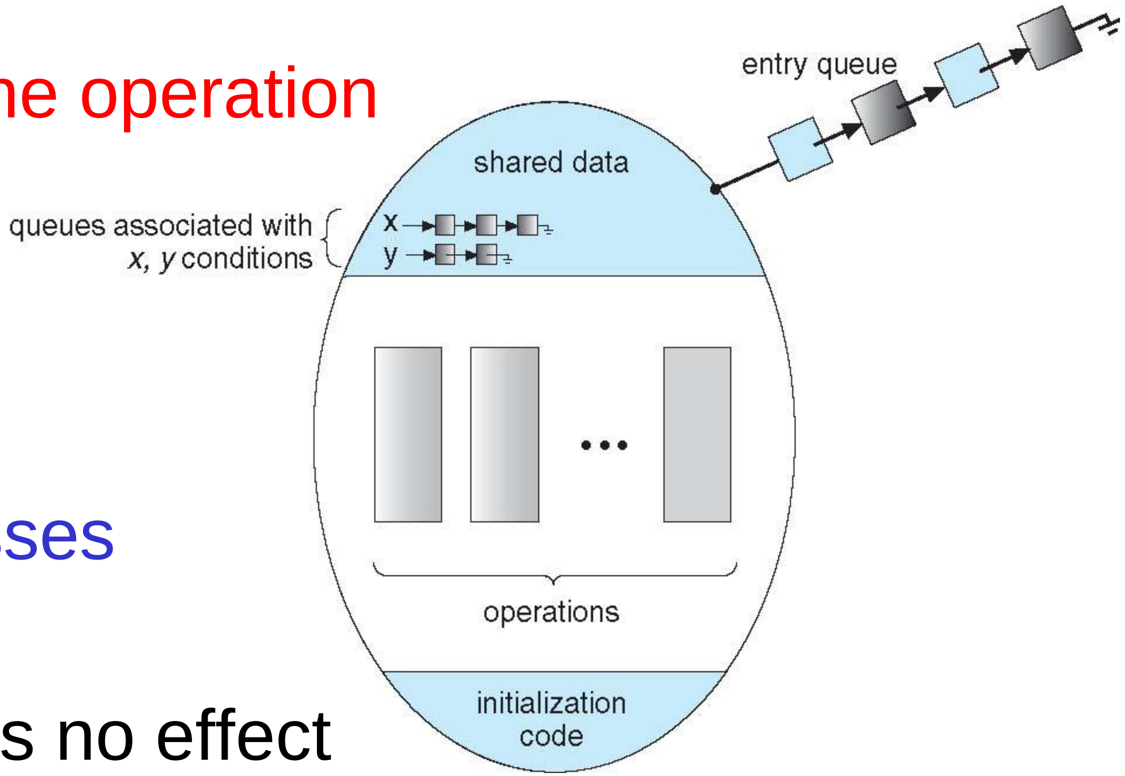
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type, internal variables only accessible by code within the procedure
- One process may be active within the monitor at a time

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



Condition Variables

- 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



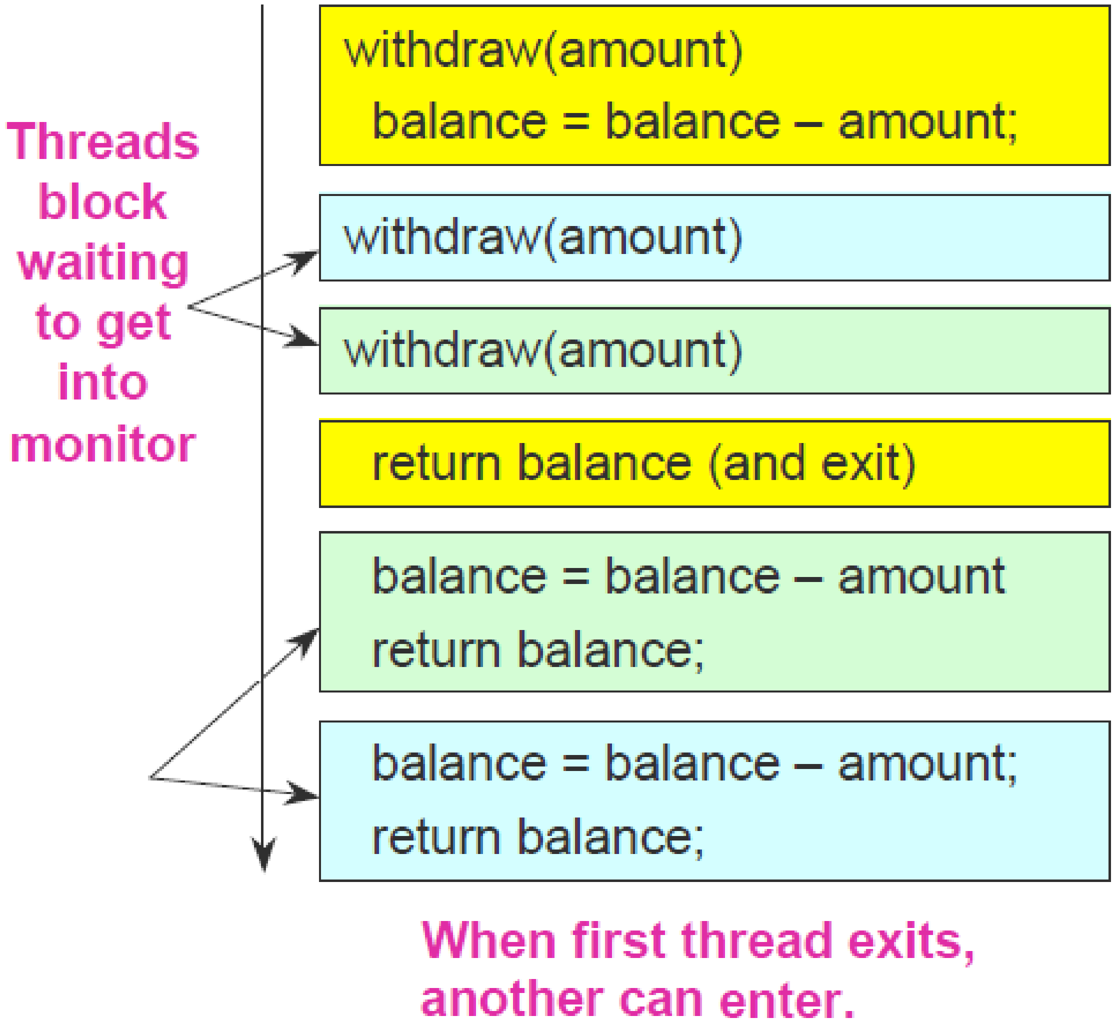
Condition Variables Choices

- 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

Implementation using Monitors

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance – amount;
    return balance;
  }
}
```



Thank You