Process Synchronization

CSCI 3431: Operating Systems

Agenda

- Assignment 2 out, due October 20
- Midterm next Wednesday
- Today's lecture
  - Process Synchronization Mechanisms
  - Monitors
  - Classic Synchronization Problems
  - Textbook Reading: 6.6, 6.7, 6.8

Producer-Consumer Problem

General Statement:

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at a time
- Only one producer or consumer may access the buffer at any one time

The Problem:

Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer

Common Concurrency Mechanisms

- Semaphore – integer value used for signaling among processes.
  - Initialize, increment, decrement operations may be performed on a semaphore
  - Operations are atomic
  - Decrement operation may result in the blocking of a process
  - Increment operation may result in the unblock of a process
  - Binary Semaphore – takes only the values 0 and 1
- Mutex – similar to a binary semaphore
  - With the restriction that the process that locks the mutex must be the one to unlock it
  - Condition variable: a data type used to block a process/thread until a particular condition holds

Semaphore Implementation

- Avoid busy waiting by putting the thread to sleep
- System can use a queue and block all threads that are waiting

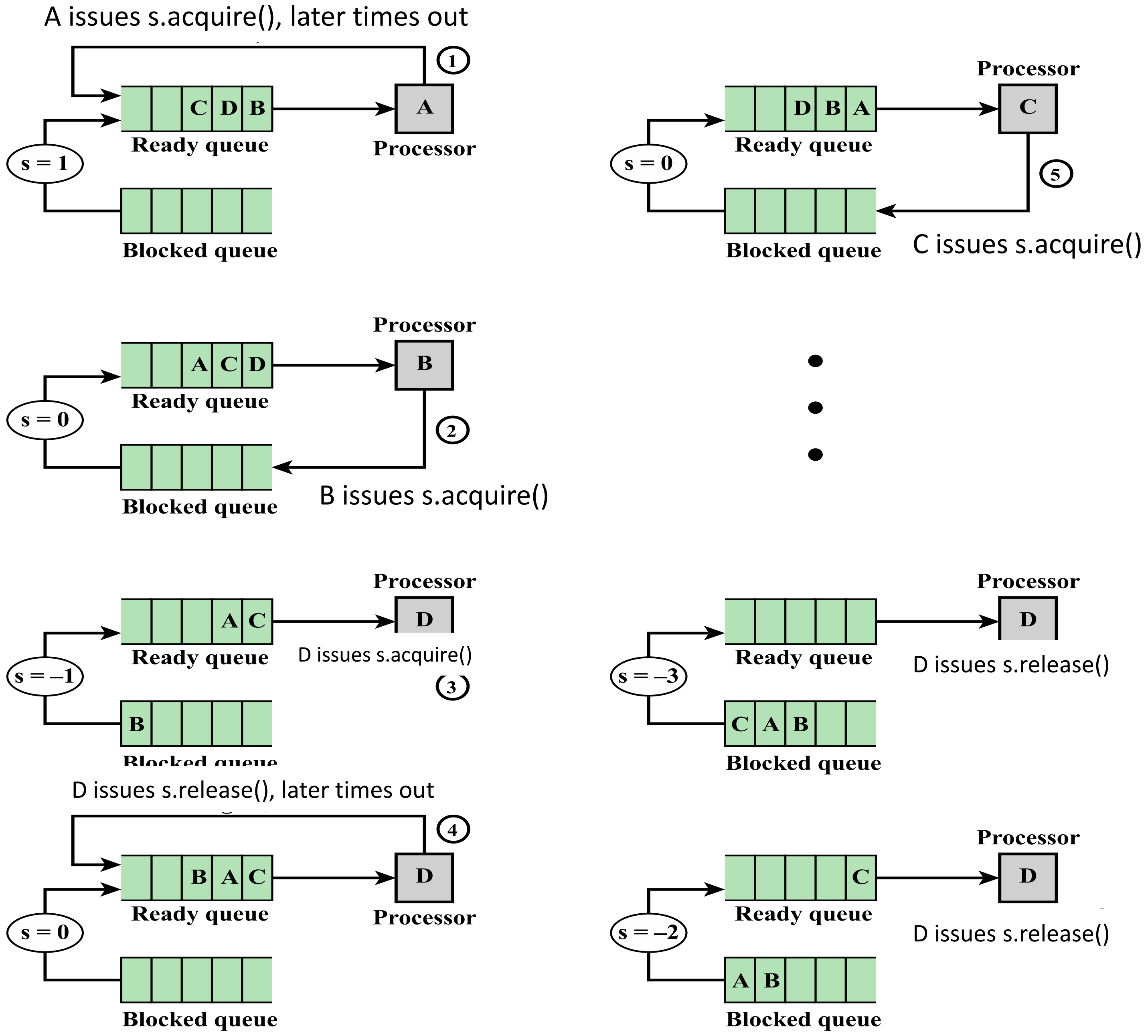
```
public class Semaphore{
  private int value;
  public Semaphore(int value) {
    this.value = value;
  }
....
 }
```

```
public synchronized void acquire() {
   while (value <= 0) {
     try {
       wait();
     }
     catch (InterruptedException e) { }
   }
  value--;
 }
```

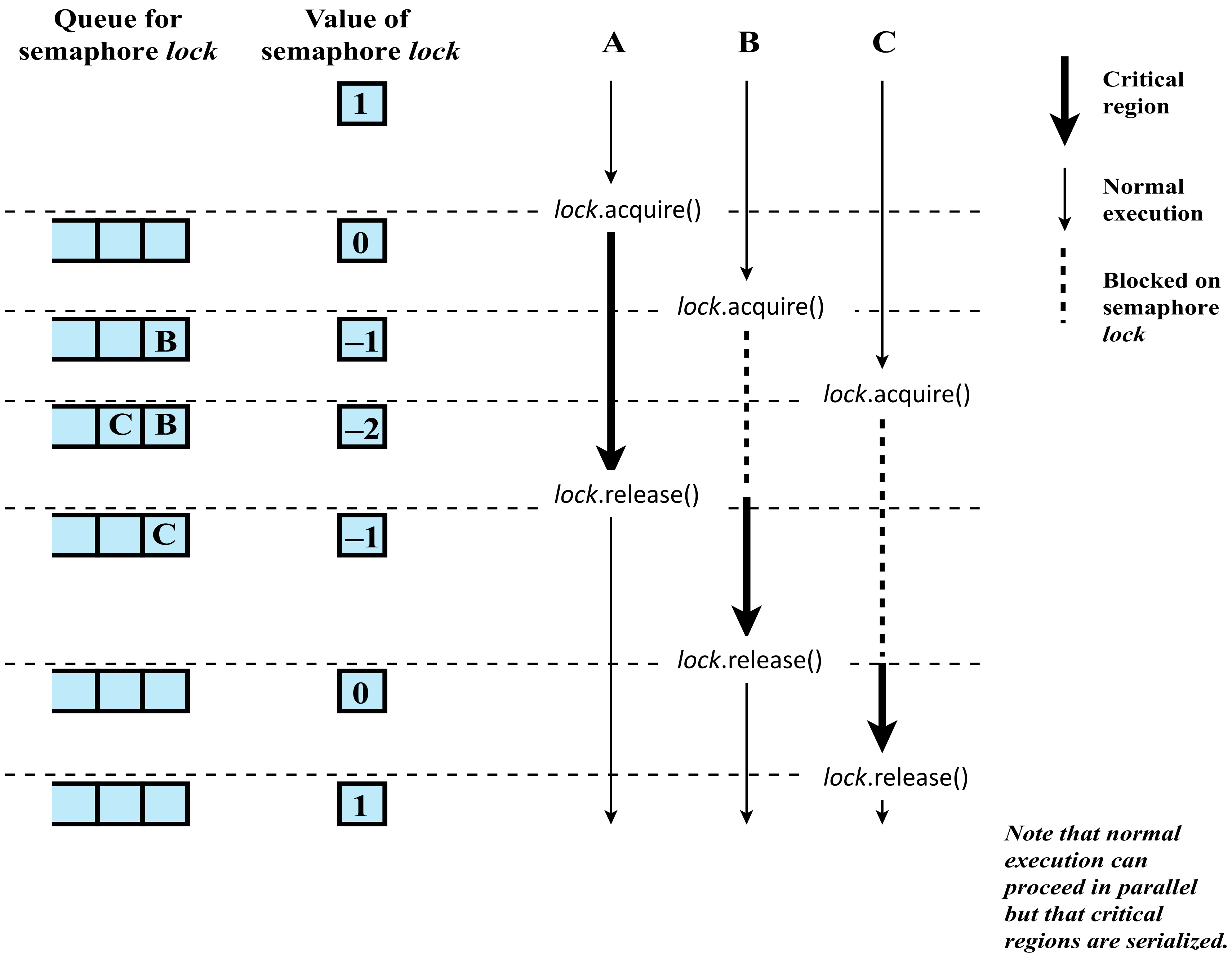
```
  public synchronized void release() {
    ++value;
    notify();
  }
```

- Wake up a waiting process

Example of Semaphore Mechanism



Semaphore used to protect access to shared data



Problems with Sempahores

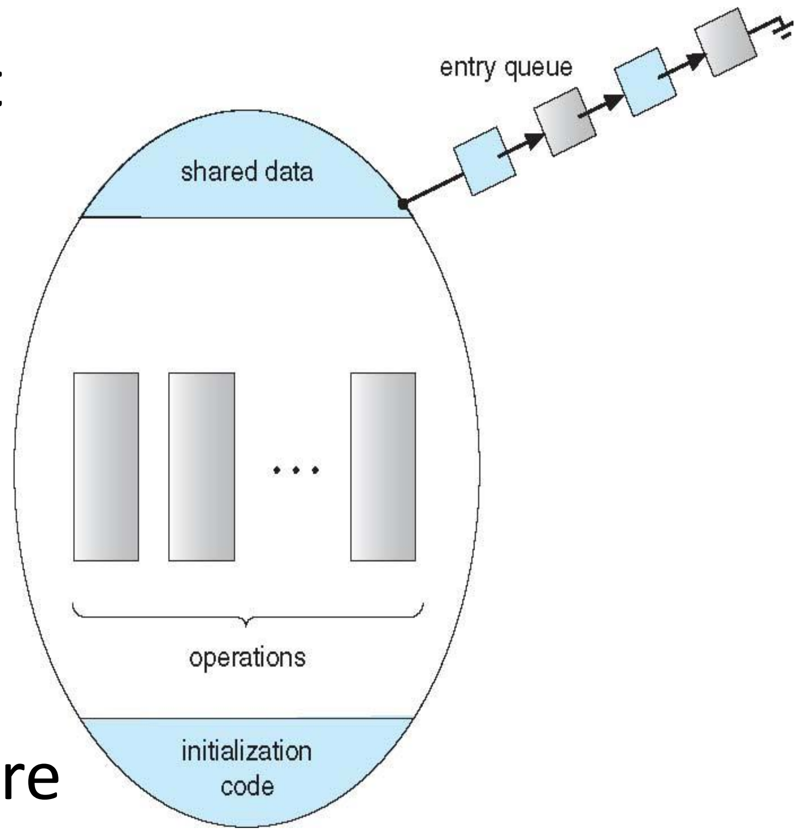
- Correct use of semaphore operations:
  - Correct ➔ mutex.acquire() …. mutex.release()
  - Incorrect ➔ mutex.acquire () or mutex.release() (or both)
  - Omitting either mutex.acquire() or mutex.release()

Monitor

- Provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
- Has also been implemented as a program library

Common Concurrency Mechanisms

- Monitor – programming language construct that encapsulates within an abstract data type:
  - Local data – variables that may only be accessed via access procedures
  - Access procedures – critical sections
  - Initialization code
- Only one process may be actively accessing the monitor at any one time
- A monitor may have a queue of processes that are waiting to access it



Monitor Characteristics

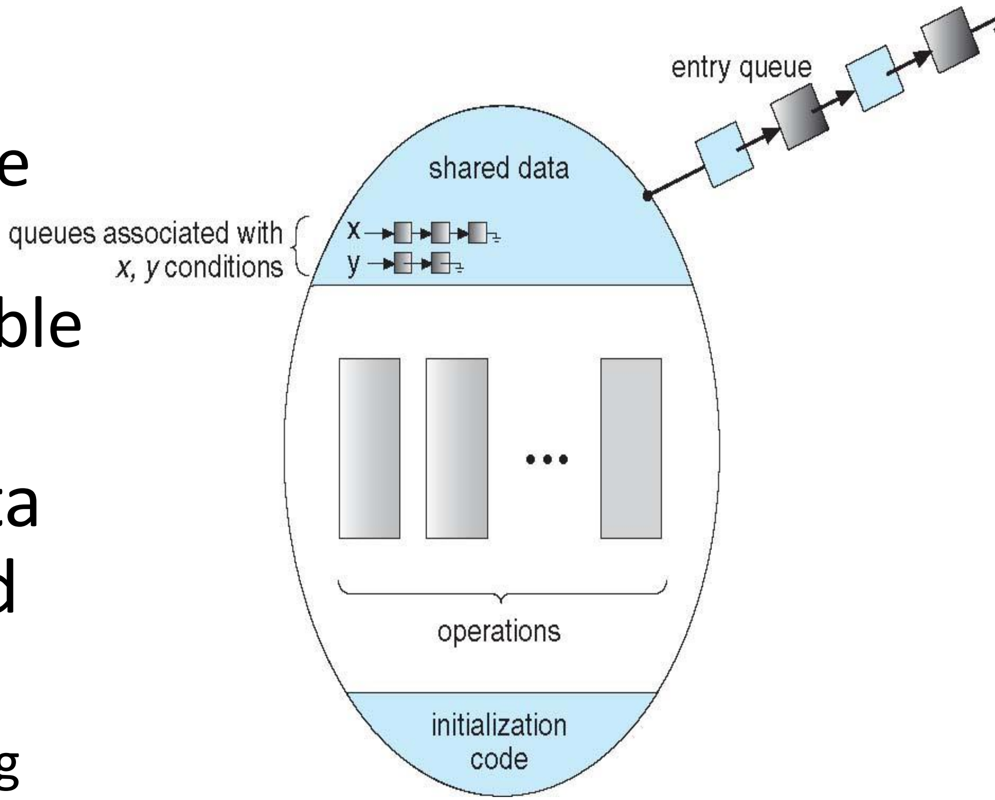
Local data variables are accessible only by the monitor's procedures and not by any external procedure

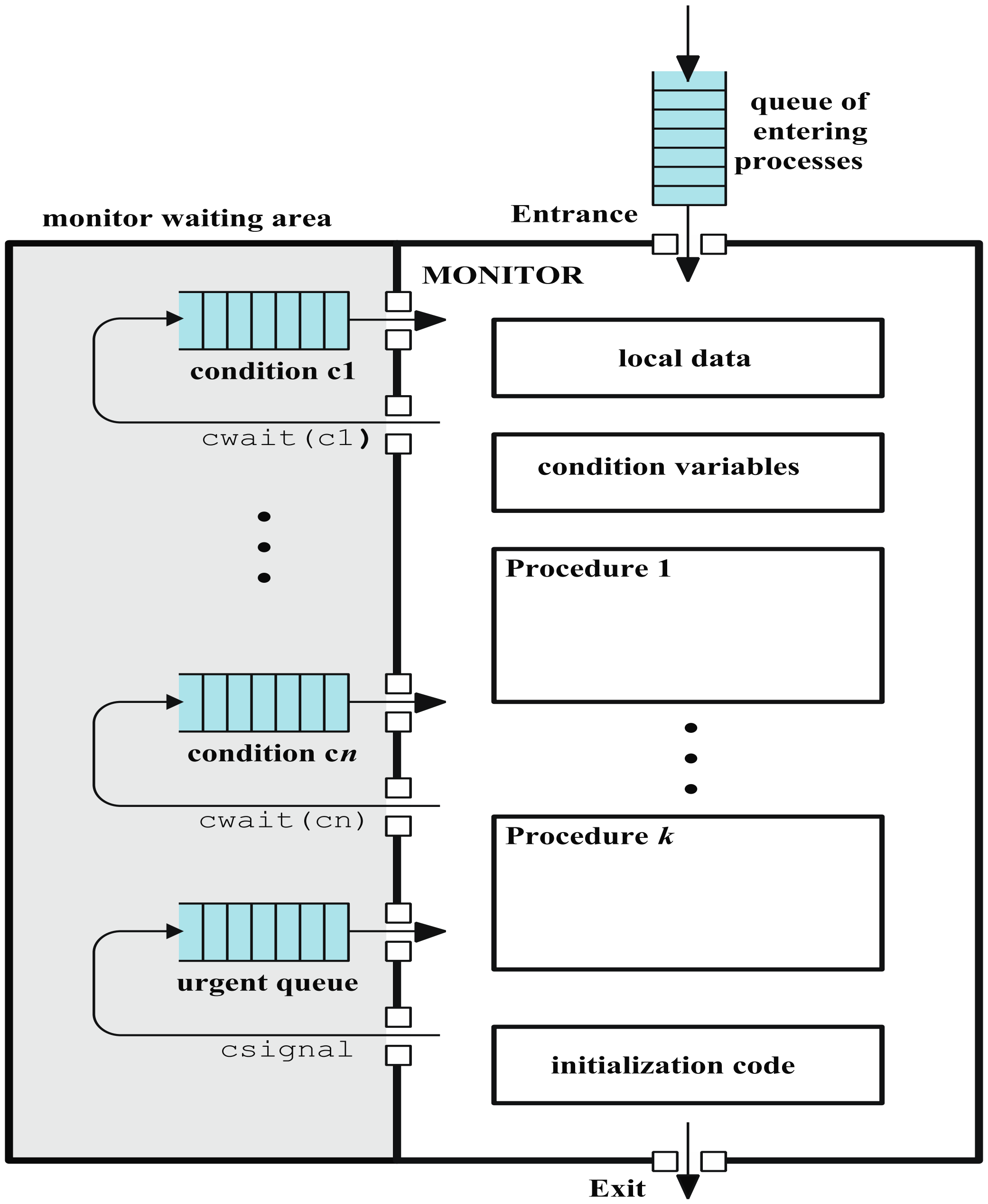
Process enters monitor by invoking one of its procedures

Only one process may be executing in the monitor at a time

Synchronization

- A monitor supports synchronization by the use of **condition variables (x,y)** that are contained within the monitor and accessible only within the monitor
  - Condition variables are a special data type in monitors which are operated on by two functions:
    - `x.wait()`: suspend execution of the calling process on condition `x`
    - `x.signal()`: resume execution of some process blocked that invoked `x.wait()` on the same condition `x`




queue of entering processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

Classic Problems of Synchronization

- Producer Consumer (Bounded-Buffer) Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Bounded-Buffer

```java
public class BoundedBuffer<E> implements
Buffer<E>{

private static final int   BUFFER_SIZE = 5;

private Semaphore mutex;
private Semaphore empty;
private Semaphore full;

private int count;
private int in, out;
private E[] buffer;



public BoundedBuffer(){
// buffer is initially empty
count = 0;
in = 0;
out = 0;

buffer = (E[]) new Object[BUFFER_SIZE];

mutex = new Semaphore(1);
empty = new Semaphore(BUFFER_SIZE);
full = new Semaphore(0);

}
```

//Fig. 6.9

```java
// producer calls this method
public void insert(E item) {

empty.acquire();
mutex.acquire();

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;

if (count == BUFFER_SIZE)
          System.out.println("Producer Entered " + item + " Buffer FULL");

        else
          System.out.println("Producer Entered " + item + " Buffer Size = " +
count);

mutex.release();
full.release();

}


// consumer calls this method
public E remove() {

full.acquire();
mutex.acquire();

// remove an item from the buffer
--count;
E item = buffer[out];
out = (out + 1) % BUFFER_SIZE;

if (count == 0)
           System.out.println("Consumer Consumed " + item + " Buffer EMPTY");

        else
          System.out.println("Consumer Consumed " + item + " Buffer Size = " +
count);

mutex.release();
empty.release();

return item;

}
```

// Fig. 6.10 – 6.11

Bounded-Buffer

```
public class Producer implements Runnable{
private  Buffer<Date> buffer;
   public Producer(Buffer<Date> buffer) {
       this.buffer = buffer;
}


public void run(){
 Date message;
  while (true) {
    System.out.println("Producer napping");
    SleepUtilities.nap();
    // produce an item & enter it into the buffer
    message = new Date();
    System.out.println("Producer produced " + message);
    buffer.insert(message);
   }
 }
}
//Fig. 6.12
```

```
public class Consumer implements Runnable{
private  Buffer<Date> buffer;
   public Consumer(Buffer<Date> buffer) {
      this.buffer = buffer;
   }


   public void run(){
   Date message;
      while (true) {
         System.out.println("Consumer napping");
         SleepUtilities.nap();
         // consume an item from the buffer
         System.out.println("Consumer wants to consume.");
         message = buffer.remove();
      }
   }
}
// Fig. 6.13
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore db initialized to 1
  - Integer readerCount initialized to 0

Dining-Philosophers Problem

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopStick [5] initialize