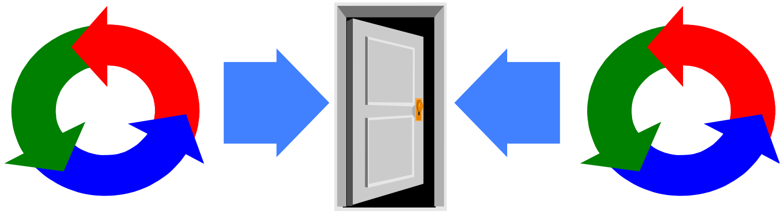
Chapter 5

Monitors & Condition Synchronization



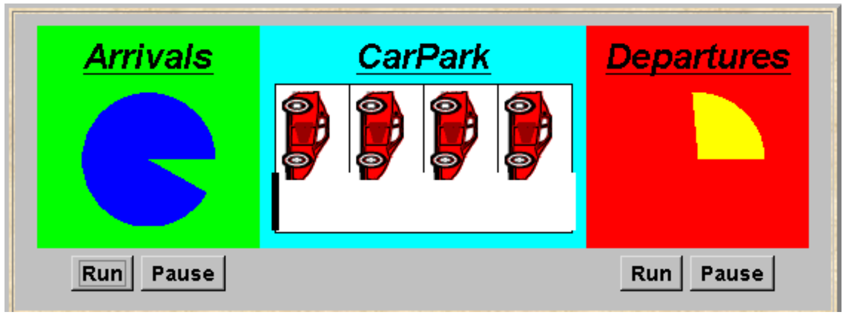
monitors & condition synchronization

Concepts: **monitors**:
encapsulated data + access procedures
mutual exclusion + condition synchronization
nested monitors

Models: guarded actions

Practice: private data and synchronized methods (exclusion).
wait(), notify() and notifyAll() for condition synch.
single thread active in the monitor at a time

5.1 Condition synchronization



A controller is required for a carpark, which only permits cars to enter when the carpark is not full and permits cars to leave when there it is not empty. Car arrival and departure are simulated by separate threads.

carpark model

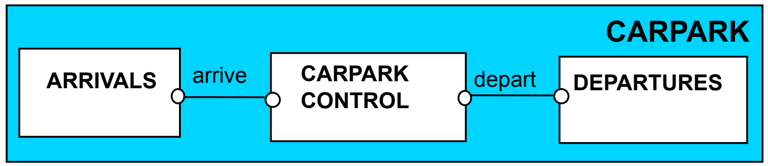
- Events or actions of interest?

  `arrive` and `depart`

- Identify processes.

  arrivals, departures and carpark control

- Define each process and interactions (structure).



carpark model

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

ARRIVALS   = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK =
      (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```

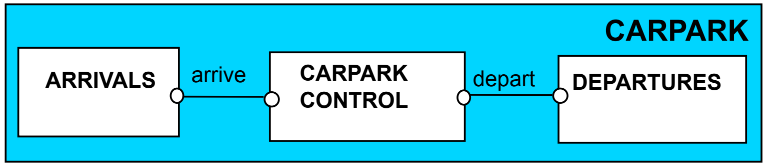
Guarded actions are used to control **arrive** and **depart**.

LTS?

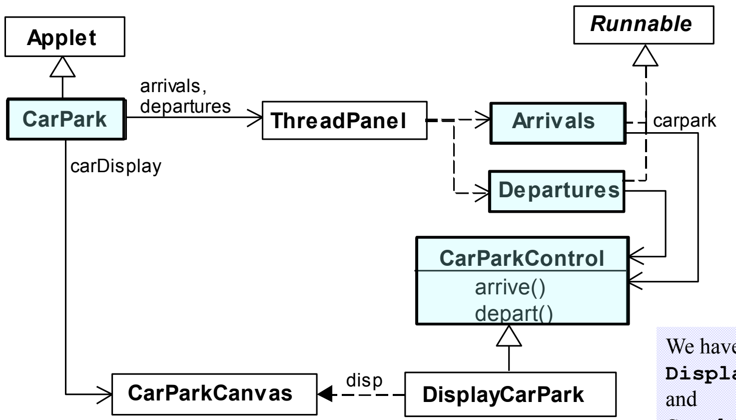
carpark program

- Model - all entities are processes interacting by actions
- Program - need to identify threads and monitors
  - thread - **active** entity which initiates (output) actions
  - monitor - **passive** entity which responds to (input) actions.

For the carpark?



carpark program - class diagram



We have omitted **DisplayThread** and **GraphicCanvas** threads managed by **ThreadPanel**.

carpark program

Arrivals and **Departures** implement **Runnable**, **CarParkControl** provides the control (condition synchronization).

Instances of these are created by the **start()** method of the **CarPark** applet :

```
public void start() {
  CarParkControl c =
     new DisplayCarPark(carDisplay,Places);
     arrivals.start(new Arrivals(c));
     departures.start(new Departures(c));
}
```

carpark program - `Arrivals` and `Departures` threads

```
class Arrivals implements Runnable {
  CarParkControl carpark;

  Arrivals(CarParkControl c) {carpark = c;}

  public void run() {
    try {
      while(true) {
        ThreadPanel.rotate(330);
        carpark.arrive();
        ThreadPanel.rotate(30);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly `Departures` which calls `carpark.depart()`.

How do we implement the control of `CarParkControl`?

Carpark program - `CarParkControl` monitor

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() {
    … --spaces; …
  }

  synchronized void depart() {
    … ++spaces; …
  }
}
```

mutual exclusion by synch methods

condition synchronization?

block if full? (spaces==0)

block if empty? (spaces==N)

condition synchronization in Java

Java provides a thread **wait set** per monitor (actually per object) with the following methods:

`public final void notify()`
Wakes up a single thread that is waiting on this object's wait set.

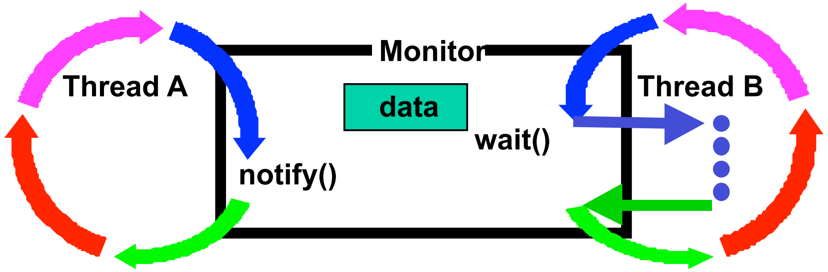
`public final void notifyAll()`
Wakes up all threads that are waiting on this object's wait set.

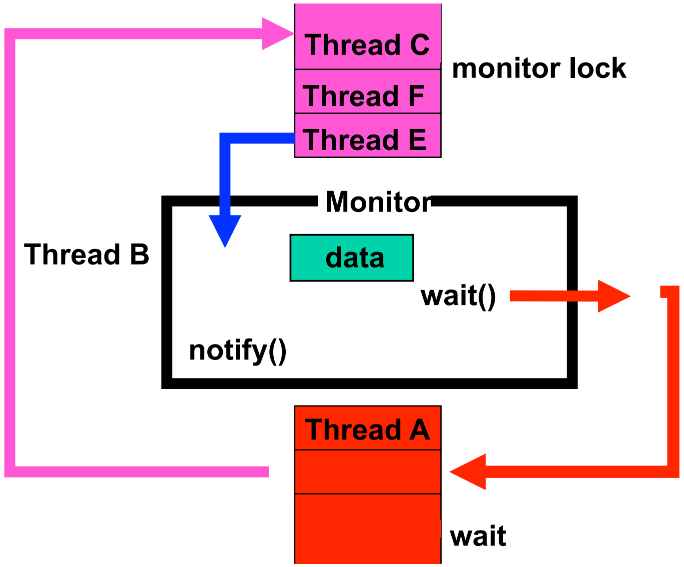
`public final void wait() throws InterruptedException`
Waits to be notified by another thread. The waiting thread releases the synchronization lock associated with the monitor. When notified, the thread must wait to reacquire the monitor before resuming execution.

condition synchronization in Java

We refer to a thread ***entering*** a monitor when it acquires the mutual exclusion lock associated with the monitor and ***exiting*** the monitor when it releases the lock.

Wait() - causes the thread to exit the monitor, permitting other threads to enter the monitor.





condition synchronization in Java

```
FSP:    when cond act -> NEWSTAT
```

```
Java:   public synchronized void act()
               throws InterruptedException
        {
         while (!cond) wait();
           // modify monitor data
         notifyAll()
        }
```

The **`while`** loop is necessary to retest the condition *cond* to ensure that *cond* is indeed satisfied when it re-enters the monitor.

`notifyall()` is necessary to awaken other thread(s) that may be waiting to enter the monitor now that the monitor data has been changed.

`CarParkControl` - condition synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();                   block if full
    --spaces;
    notifyAll();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();            block if empty
    ++spaces;
    notifyAll();
  }
}
```

Is it safe to use **`notify()`** *rather than* **`notifyAll()`***?*

models to monitors - summary

Active entities (that initiate actions) are implemented as **threads**.
Passive entities (that respond to actions) are implemented as **monitors**.

Each guarded action in the model of a monitor is implemented as a **`synchronized`** method which uses a while loop and **`wait()`** to implement the guard. The while loop condition is the negation of the model guard condition.

Changes in the state of the monitor are signaled to waiting threads using **`notify()`** or **`notifyAll()`**.

5.2 Semaphores

Semaphores are a low-level, primitive construct widely used for dealing with inter-process synchronization in operating systems. Semaphore ***s*** is an integer variable that can take only non-negative values.

The only operations permitted on ***s*** are *up(s)* and *down(s)*. Blocked processes are held in a FIFO queue.

down(s): **if** *s* >0 **then**
  decrement *s*
 else
  block execution of the calling process

up(s): **if** processes blocked on *s* **then**
  awaken one of them
 else
  increment *s*

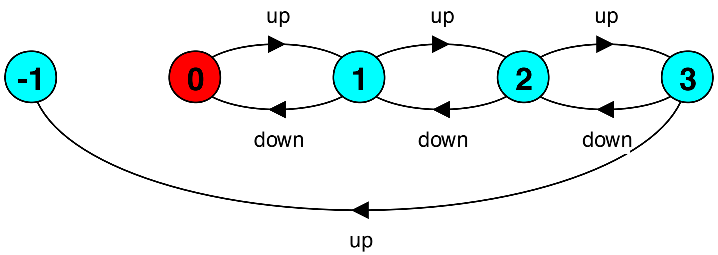
modeling semaphores

To ensure analyzability, we only model semaphores that take a finite range of values. If this range is exceeded then we regard this as an ERROR. N is the initial value.

```
const Max = 3
range Int = 0..Max

SEMAPHORE(N=0)  = SEMA[N],
SEMA[v:Int]     = (up->SEMA[v+1]
                  |when(v>0) down->SEMA[v-1]
                  ),
SEMA[Max+1]     = ERROR.
```

LTS?

modeling semaphores



Action **down** is only accepted when value v of the semaphore is greater than 0.

Action **up** is not guarded.

Trace to a violation:

up → up → up → up

semaphore demo - model

Three processes `p[1..3]` use a shared semaphore `mutex` to ensure mutually exclusive access (action `critical`) to some resource.

```
LOOP = (mutex.down->critical->mutex.up->LOOP).

||SEMADEMO = (p[1..3]:LOOP
             ||{p[1..3]}::mutex:SEMAPHORE(1)).
```

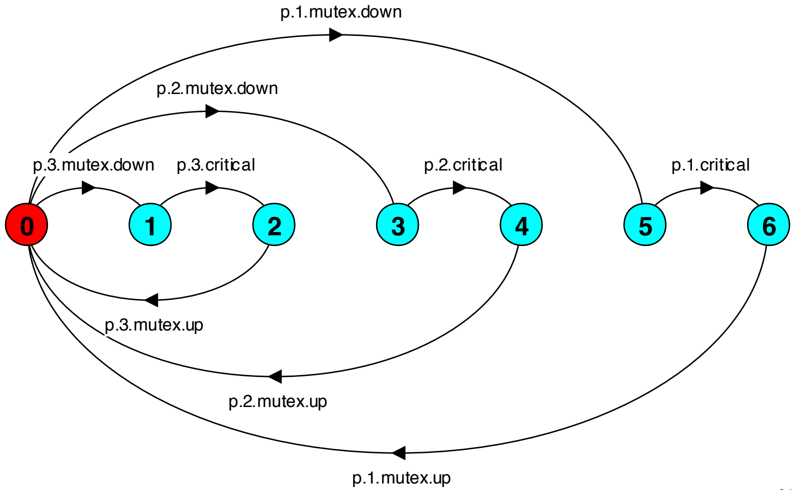
For mutual exclusion, the semaphore initial value is 1. ***Why?***

Is the **ERROR** *state reachable for* **SEMADEMO?**

Is a binary semaphore sufficient (i.e. **Max=1**) ***?***

LTS?

semaphore demo - model



semaphores in Java

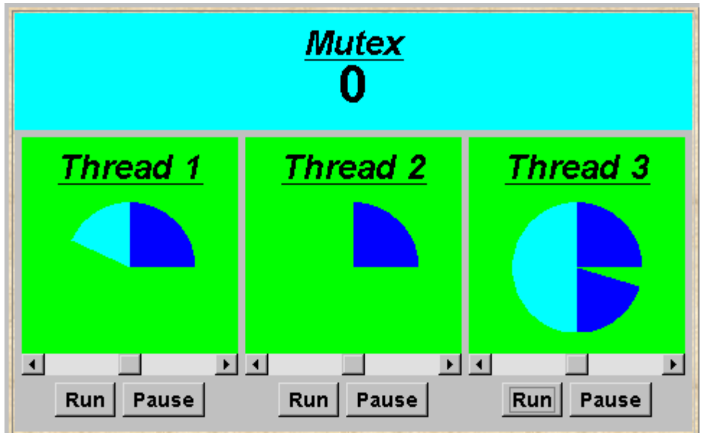
Semaphores are passive objects, therefore implemented as **monitors**.

(NOTE: In practice, semaphores are a low-level mechanism often used for implementing the higher-level monitor construct.

Java SE5 provides general counting semaphores)

```
public class Semaphore {
  private int value;

  public Semaphore (int initial)
    {value = initial;}

  synchronized public void up() {
     ++value;
     notifyAll();
  }

  synchronized public void down()
      throws InterruptedException {
    while (value== 0) wait();
    --value;
  }
}
```


SEMADEMO display



current semaphore value

thread 1 is executing critical actions.

thread 2 is blocked waiting.

thread 3 is executing non-critical actions.

SEMADEMO

What if we adjust the time that each thread spends in its critical section ?

- large resource requirement - *more conflict?*

  (eg. more than 67% of a rotation)?

- small resource requirement - *no conflict?*

  (eg. less than 33% of a rotation)?

Hence the time a thread spends in its critical section should be kept as short as possible.

SEMADEMO program - revised ThreadPanel class

```
public class ThreadPanel extends Panel {
   // construct display with title and rotating arc color c
   public ThreadPanel(String title, Color c) {…}
   // hasSlider == true creates panel with slider
   public ThreadPanel
   (String title, Color c, boolean hasSlider) {…}
   // rotate display of currently running thread 6 degrees
   // return false when in initial color, return true when in second color
   public static boolean rotate()
          throws InterruptedException {…}
   // rotate display of currently running thread by degrees
   public static void rotate(int degrees)
          throws InterruptedException {…}
   // create a new thread with target r and start it running
   public void start(Runnable r) {…}
   // stop the thread using Thread.interrupt()
   public void stop() {…}
}
```

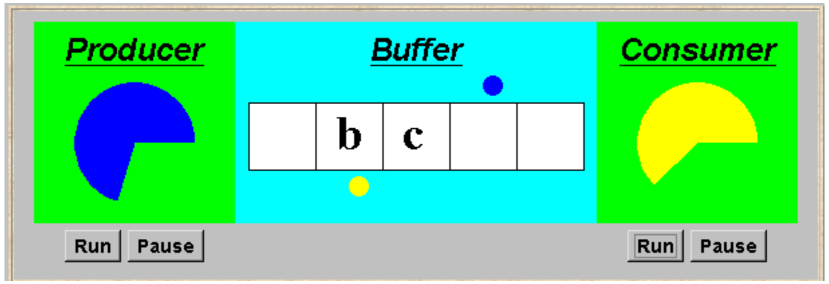

SEMADEMO program - MutexLoop

```
class MutexLoop implements Runnable {
  Semaphore mutex;

  MutexLoop (Semaphore sema) {mutex=sema;}

  public void run() {
    try {
      while(true)  {
        while(!ThreadPanel.rotate());
        mutex.down();            // get mutual exclusion
        while(ThreadPanel.rotate()); //critical actions
        mutex.up();              //release mutual exclusion
      }
    } catch(InterruptedException e){}
  }
}
```

Threads and semaphore are created by the applet `start()` method.

`ThreadPanel.rotate()` returns `false` while executing non-critical actions (dark color) and `true` otherwise.

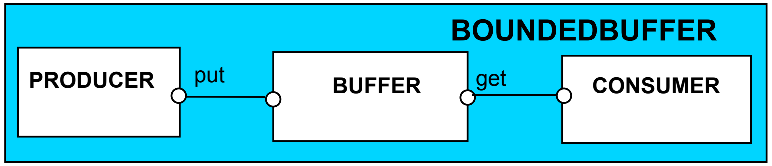
5.3 Bounded Buffer



A bounded buffer consists of a fixed number of slots. Items are put into the buffer by a *producer* process and removed by a *consumer* process. It can be used to smooth out transfer rates between the *producer* and *consumer*.

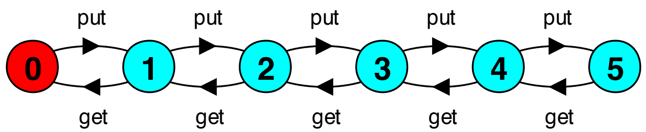
(see car park example)

bounded buffer - a data-independent model



The behaviour of BOUNDEDBUFFER is independent of the actual data values, and so can be modelled in a data-independent manner.

LTS:



bounded buffer - a data-independent model

```
BUFFER(N=5) = COUNT[0],
COUNT[i:0..N]
     = (when (i<N) put->COUNT[i+1]
       |when (i>0) get->COUNT[i-1]
       ).

PRODUCER = (put->PRODUCER).
CONSUMER = (get->CONSUMER).

||BOUNDEDBUFFER = (PRODUCER||BUFFER(5)||
CONSUMER).
```


bounded buffer program - buffer monitor

```
public interface Buffer <E> {…}

class BufferImpl <E> implements Buffer <E> {
    …
 public synchronized void put(E o)
            throws InterruptedException {
   while (count==size) wait();
   buf[in] = o; ++count; in=(in+1)%size;
   notifyAll();
 }
  public synchronized E get()
            throws InterruptedException {
   while (count==0) wait();
   E o =buf[out];
   buf[out]=null; --count; out=(out+1)%size;
   notifyAll();
   return (o);
   }
}
```

We separate the interface to permit an alternative implementation later.

bounded buffer program - producer process

```
class Producer implements Runnable {
  Buffer buf;
  String alphabet= "abcdefghijklmnopqrstuvwxyz";

  Producer(Buffer b) {buf = b;}

  public void run() {
    try {
      int ai = 0;
      while(true) {
        ThreadPanel.rotate(12);
        buf.put(alphabet.charAt(ai));
        ai=(ai+1) % alphabet.length();
        ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }
}
```

Similarly `Consumer` which calls `buf.get()`.

5.4 Nested Monitors

Suppose that, in place of using the ***count*** variable and condition synchronization directly, we instead use two semaphores ***full*** and ***empty*** to reflect the state of the buffer.

```
class SemaBuffer <E> implements Buffer <E> {
  …

  Semaphore full;  //counts number of items
  Semaphore empty; //counts number of spaces

  SemaBuffer(int size) {
    this.size = size; buf =(E[])new Object[size];
    full = new Semaphore(0);
    empty= new Semaphore(size);
  }
…
}
```

nested monitors - bounded buffer program

```
synchronized public void put(E o)
            throws InterruptedException {
   empty.down();
   buf[in] = o;
   ++count; in=(in+1)%size;
   full.up();
 }
 synchronized public E get()
             throws InterruptedException{
   full.down();
   E o =buf[out]; buf[out]=null;
   --count; out=(out+1)%size;
   empty.up();
   return (o);
 }
```

Does this behave as desired?

empty is decremented during a **put** operation, which is blocked if *empty* is zero; *full* is decremented by a **get** operation, which is blocked if *full* is zero.

nested monitors - bounded buffer model

```
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0)
                  )@{put,get}.
```

Does this behave as desired?

nested monitors - bounded buffer model

LTSA analysis predicts a possible **DEADLOCK**:

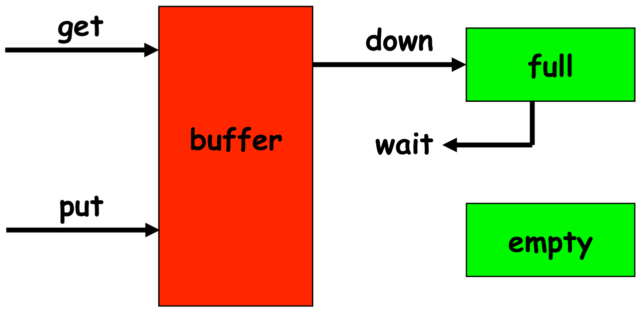
```
Composing
 potential DEADLOCK
States Composed: 28 Transitions: 32 in 60ms
Trace to DEADLOCK:
  get
```

The **Consumer** tries to **get** a character, but the buffer is empty. It blocks and releases the lock on the semaphore **full**. The **Producer** tries to **put** a character into the buffer, but also blocks. ***Why?***

This situation is known as the ***nested monitor problem*****.**

nested monitors - bounded buffer model

```
synchronized public Object get()
               throws InterruptedException{
   full.down(); // if no items, block!
   ...
 }
```



nested monitors - revised bounded buffer program

The only way to avoid it in Java is by careful design. In this example, the deadlock can be removed by ensuring that the monitor lock for the buffer is not acquired until ***after*** semaphores are decremented.

```
public void put(E o)
              throws InterruptedException {
   empty.down();
   synchronized(this){
     buf[in] = o; ++count; in=(in+1)%size;
   }
   full.up();
 }
```


nested monitors - revised bounded buffer model

```
BUFFER =  (put -> BUFFER
          |get -> BUFFER
          ).

PRODUCER =(empty.down->put->full.up->PRODUCER).
CONSUMER =(full.down->get->empty.up->CONSUMER).
```

The semaphore actions have been moved to the producer and consumer. This is exactly as in the implementation where the semaphore actions are ***outside*** the monitor .

Does this behave as desired?

Minimized LTS?

5.5 Monitor invariants

An **invariant** for a monitor is an assertion concerning the variables it encapsulates. This assertion must hold whenever there is no thread executing inside the monitor i.e. on thread **entry** to and **exit** from a monitor .

CarParkControl Invariant:		$0 \le spaces \le N$
Semaphore Invariant:		$0 \le value$
Buffer Invariant:		$0 \le count \le size$
	and	$0 \le in < size$
	and	$0 \le out < size$
	and	$in = (out + count)$ **modulo** $size$

Invariants can be helpful in reasoning about correctness of monitors using a logical *proof-based* approach. Generally we prefer to use a *model-based* approach amenable to mechanical checking .

5.6 Java Concurrency Utilities Package

java.util.concurrent includes **semaphores**, and **explicit locks with multiple condition variables.**

Monitors: implicit lock associated with each object, with methods *wait(), notify() and notifyAll()*

Lock interface: explicit lock objects, with methods *lock(), unlock(), tryLock(),* and ***newCondition()***

Condition objects: explicit lock synchronization objects, with methods *await(), signal() and signalAll()*

Conditions gives the effect of having multiple wait-sets per object.

bounded buffer – explicit lock with separate conditions

```
class BufferImpl <E> implements Buffer <E> {

   final Lock buflock = new ReentrantLock();
   final Condition notFull = buflock.newCondition();
   final Condition notEmpty = buflock.newCondition();
    …
 public void put(E o) throws InterruptedException {
     buflock.lock();
     try {while (count==size) notFull.await();
          buf[in] = o; ++count; in=(in+1)%size;
          notEmpty.signalAll();
     } finally {buflock.unlock();}
   }
  public E get() throws InterruptedException {
     buflock.lock();
     try {while (count==0) notEmpty.await();
          E o =buf[out];
          buf[out]=null;--count;out=(out+1)%size;
          notFull.signalAll();
          return (o);
     } finally {buflock.unlock();}
   }
}
```

notFull and notEmpty are conditions associated with buflock.

Processes wait separately: producers on notFull and consumers on notEmpty.

Summary

- Concepts
  - **monitors:** **encapsulated data + access procedures**
    mutual exclusion + condition synchronization
  - **nested monitors**
- Model
  - **guarded actions**
- Practice
  - **monitors: private data and synchronized methods in Java**
  - **wait(), notify() and notifyAll() for condition synchronization**
  - **single thread active in the monitor at a time**