Monitors

Lecture 5 of TDA384/DIT391

Principles of Concurrent Programming

Nir Piterman and Gerardo Schneider

Chalmers University of Technology | University of Gothenburg

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY

Today's menu

- Monitors
- Signaling disciplines
- Implementing monitors
- Monitors in Java
- Monitors: dos and don’ts

Today's menu

- Monitors
  - Common patterns of synchronization
  - Language constructs solving synchronization
- Implementing monitors
  - Issues and problems
  - In depth understanding
  - Choice of right constructs
- Monitors in Java
  - Language constructs solving synchronization

Learning outcomes

Knowledge and understanding:

- demonstrate knowledge of the issues and problems that arise in writing correct concurrent programs;
- identify the problems of synchronization typical of concurrent programs, such as race conditions and mutual exclusion

Skills and abilities:

- apply common patterns, such as lock, semaphores, and message-passing synchronization for solving concurrent program problems;
- apply practical knowledge of the programming constructs and techniques offered by modern concurrent programming languages;
- implement solutions using common patterns in modern programming languages

Judgment and approach:

- evaluate the correctness, clarity, and efficiency of different solutions to concurrent programming problems;
- judge whether a program, a library, or a data structure is safe for usage in a concurrent setting;
- pick the right language constructs for solving synchronization and communication problems between computational units.

Beyond semaphores

Semaphores provide a powerful, concise mechanism for synchronization and mutual exclusion

Unfortunately, they have several shortcomings:

- they are intrinsically global and unstructured: it is difficult to understand their behavior by looking at a single piece of code
- they are prone to deadlocks or other incorrect behavior: it is easy to forget to add a single, crucial call to `up` or `down`
- they do not support well different conditions

- In summary semaphores are a low-level synchronization primitive
- We will raise the level of abstraction

Monitors

Monitors

Monitors provide a structured synchronization mechanism built on top of object-oriented constructs – especially the notions of class, object, and encapsulation

In a monitor class:

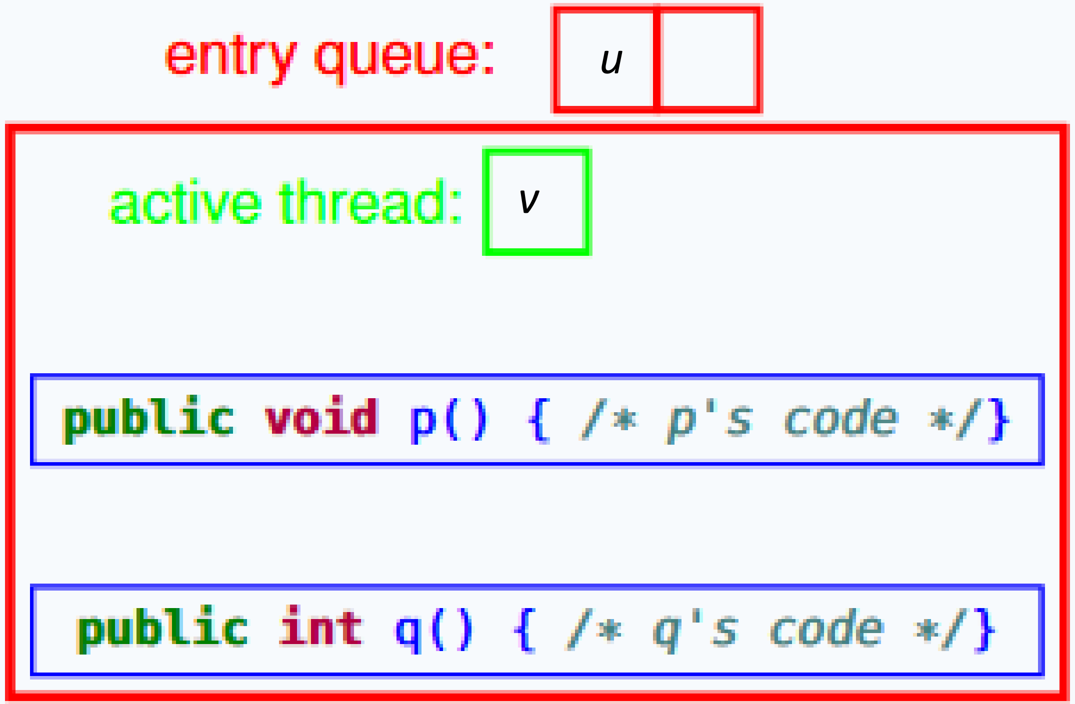
- attributes are private
- methods execute in mutual exclusion

A monitor is an object instantiating a monitor class that **encapsulates** synchronization mechanisms:

- attributes are shared variables, which all threads running on the monitor can see and modify
- methods define critical sections, with the built-in guarantee that at most one thread is active on a monitor at any time

Monitors: entry queue

Threads trying to access a monitor queue for entry; as soon as the active thread leaves the monitor the next thread in the entry queue gets exclusive access to the monitor



Monitors in pseudo-code

We declare monitor classes by adding the pseudo-code keyword to `monitor` regular Java classes

Note that `monitor` is **not** a valid Java keyword – that is why we highlight it in a different color – but we will use it to simplify the presentation of monitors

- Turning a pseudo-code monitor class into a proper Java class is straightforward:
  - mark all attributes as `private`
  - add locking to all public methods

Details on how to implement monitors in Java are presented later

Reminder: We also annotate monitor classes with invariants using the pseudo-code keyword `invariant`: **not** a valid Java keyword

Counter monitor

A shared counter that is free from race conditions:

```
monitor class Counter {
  int count = 0; // attribute, implicitly private

  public void increment() { // method, implicitly atomic
    count = count + 1;
  }

  public void decrement() { // method, implicitly atomic
    count = count - 1;
  }
}
```

The implementation of monitors guarantees that multiple threads executing `increment` and `decrement` run in mutual exclusion

Mutual exclusion for n threads

Mutual exclusion for n threads accessing their critical sections is straightforward to achieve using monitors: every monitor method executes uninterruptibly because at most one thread is running on a monitor at any time

- A proper monitor implementation also guarantees starvation freedom

```
monitor class CriticalSection {
 T1 a1; T2 a2; ... // shared data

 public void critical1() {
   // t$_1$'s critical section
 }
 // ...
 public void criticaln() {
   // t$_n$'s critical section
 }
}
```

```
CriticalSection cs;
```

thread t_k

```
while (true) {
  cs.criticalk();
  // non-critical section
}
```

Condition variables

For synchronization patterns more complex than mutual exclusion, monitors provide condition variables

A condition variable is an instance of a class with interface:

```
interface Condition {
  void wait();         // block until signal
  void signal();       // signal to unblock
  boolean isEmpty();   // is no thread waiting on this condition?
}
```

A monitor class can declare condition variables as attributes (private, thus only callable by methods of the monitor)

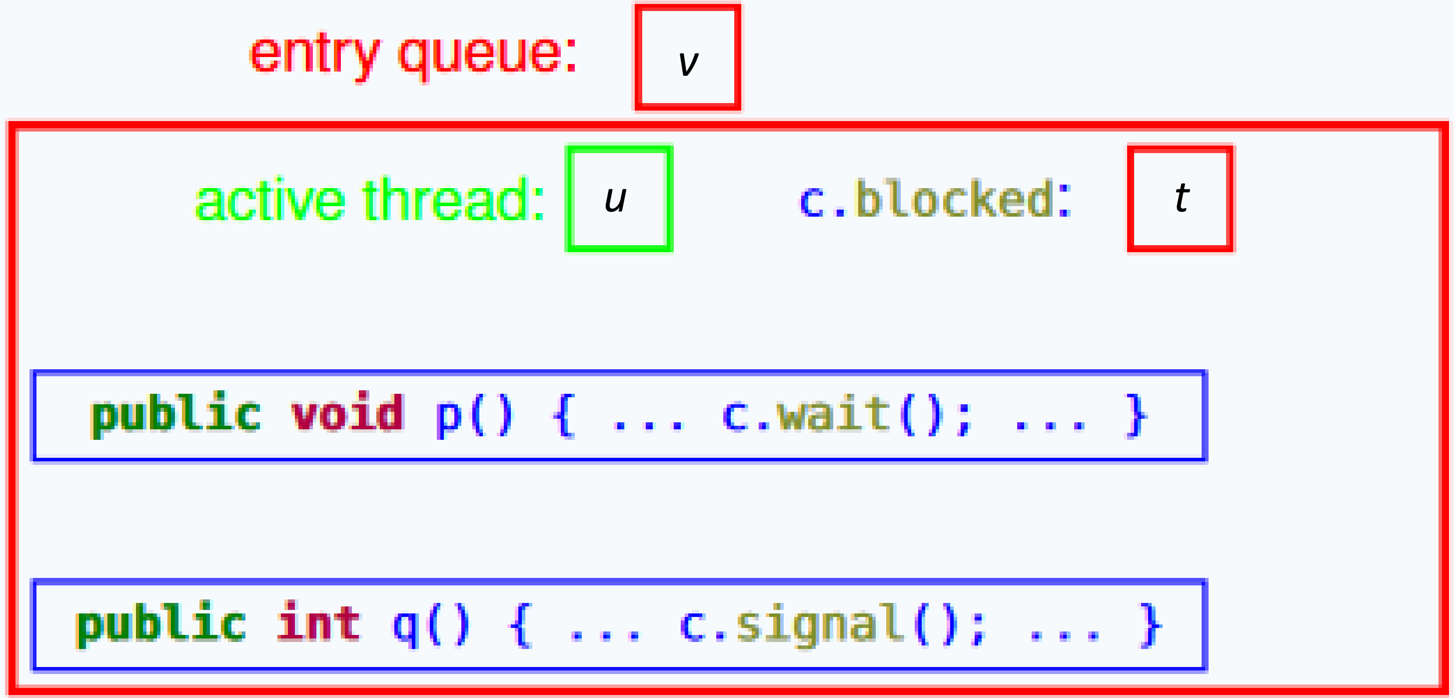
Every condition variable `c` includes a FIFO queue `blocked`:

- `c.wait()` blocks the running thread, appends it to `blocked`, and releases the lock on the monitor
- `c.signal()` removes one thread from `blocked` (if it's not empty) and unblocks it
- `c.isEmpty()` returns **true** iff `blocked` is empty

Condition variables

Every condition variable `c` includes a FIFO queue `blocked`:

- `c.wait()` <u>blocks</u> the running thread, appends it to `blocked`, and releases the lock on the monitor
- `c.signal()` removes one thread from `blocked` (if it's not empty) and unblocks it



Producer-consumer problem: recap

```
interface Buffer<T> {
    // add item to buffer; block if full
    void put(T item);

    // remove item from buffer; block if empty
    T get();

    // number of items in buffer
    int count();
}
```

Producer-consumer problem: implement Buffer such that:

- producers and consumers access the buffer in mutual exclusion
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

Producer-consumer with monitors: unbounded buffer

An implementation of producer-consumer with an unbounded buffer using <u>monitors</u>.

```
monitor class MonitorBuffer<T> implements Buffer<T> {
   Collection storage = ...; // any collection (list, set, ...)
   Condition notEmpty = new Condition(); // signal when not empty

   public void put(T item) {
      storage.add(item)             // store item
      notEmpty.signal();            // signal buffer not empty
   }
   public T get() {
      if (storage.count() == 0)
         notEmpty.wait();           // wait until buffer not empty
      return storage.remove(); // retrieve item
   }
  invariant { #storage.add == #notEmpty.signal }
}
```

No effect if there are no waiting consumers

Get in queue waiting for an item

Number of added elements to buffer equals number of signaling

No other changes to the state of the monitor

Exactly one thread woken up.

Assumption:

Producer-consumer with monitors: bounded buffer

Producer-consumer with a bounded buffer (`capacity` is the maximum size) uses two condition variables

```
monitor class BoundedMonitorBuffer<T> extends MonitorBuffer<T> {
   Condition notFull = new Condition();  // signal when not full

   public void put(T item) {
     if (storage.count() == capacity)
       notFull.wait();           // wait until buffer not full
     super.put(item);            // do as in MonitorBuffer.put(item)
   }

   public T get() {
     T item = super.get();       // do as in MonitorBuffer.get()
     notFull.signal()            // signal buffer not full
     return item;
   }
}
```

Assumption:
Exactly one thread woken up.
No other changes to the state of the monitor.

Signaling disciplines

Signaling disciplines

When a thread s calls `signal()` on a condition variable, it is executing inside the monitor

Since no more than one thread may be active on a monitor at any time, the thread u unblocked by s cannot enter the monitor immediately

The signaling discipline determines what happens to a signaling thread s after it unblocks another thread u by signaling

Two main choices of signaling discipline:

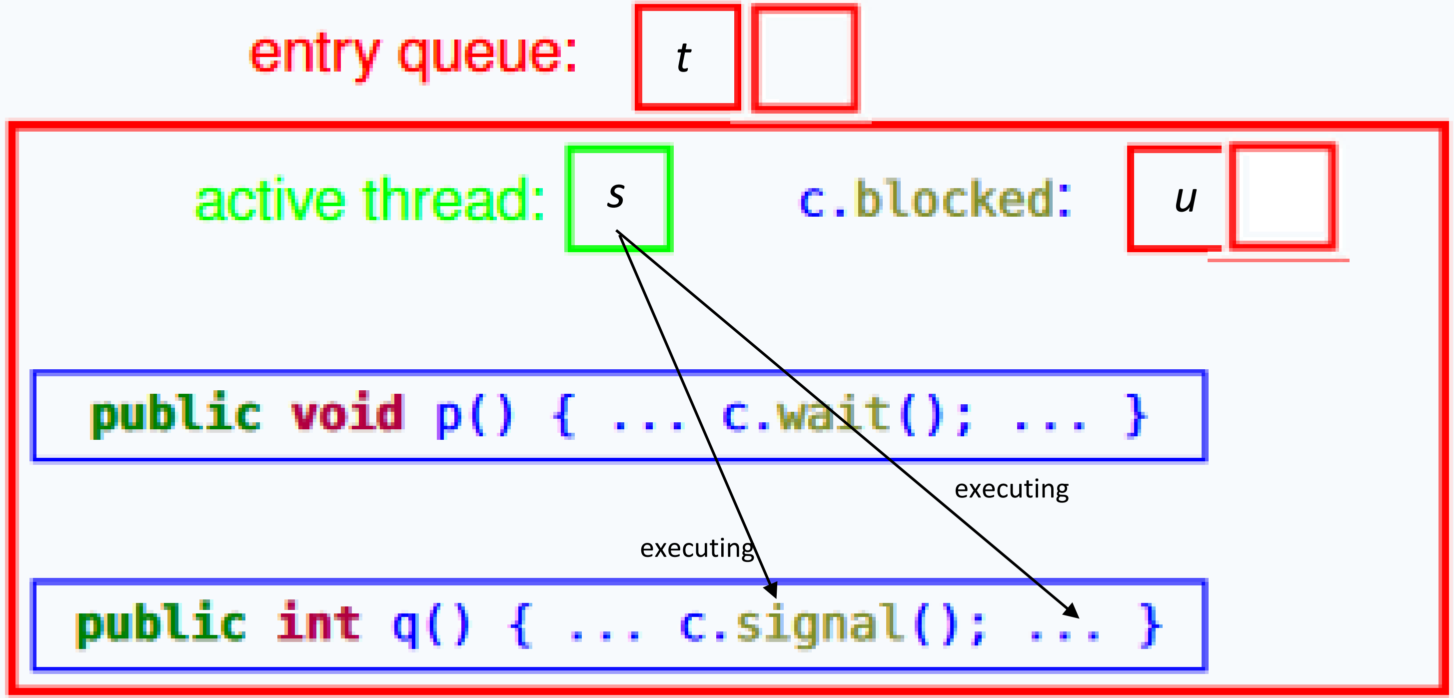
signal and continue: s continues executing;
u is moved to the entry queue of the monitor

signal and wait: s is moved to the entry queue of the monitor
u resumes executing (it silently gets the monitor's lock)

Signal and continue

Under the signal and continue discipline:

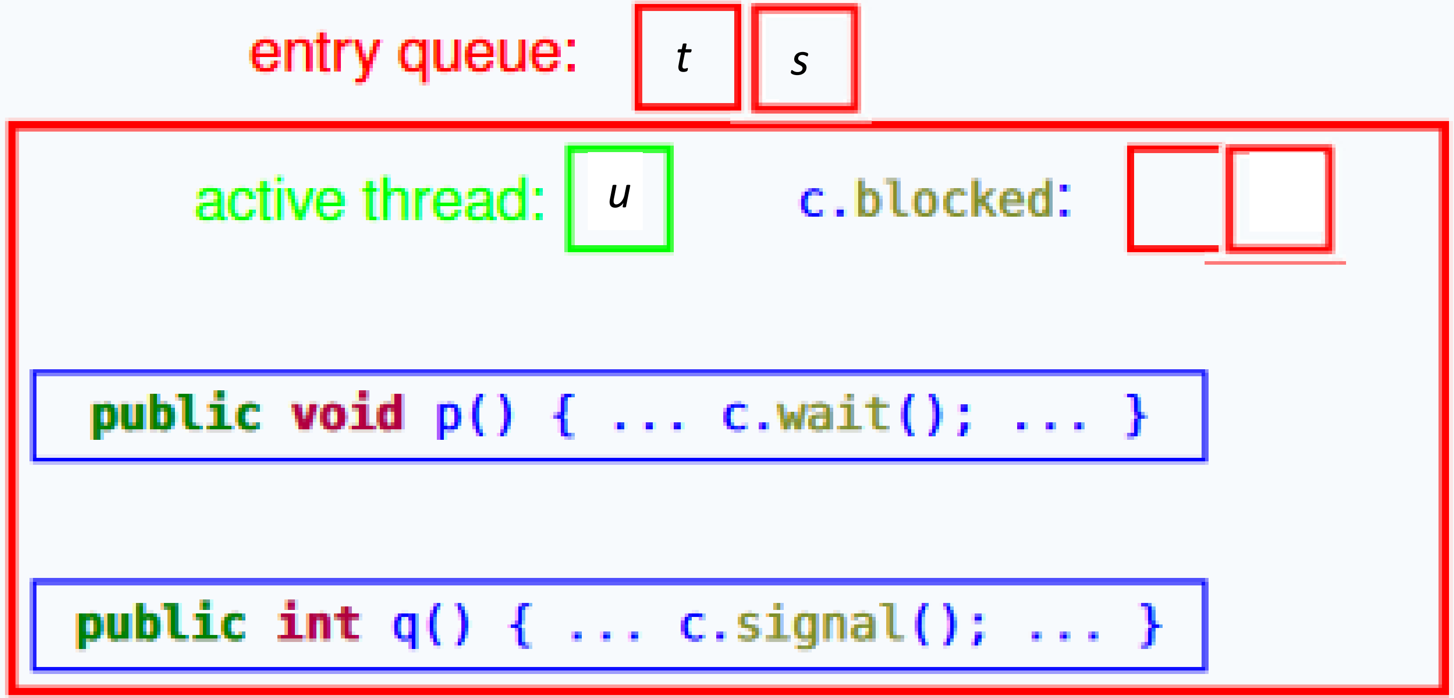
- the <u>unblocked</u> thread u is moved to the monitor's entry queue
- the <u>signaling</u> thread s continues executing



Signal and wait

Under the signal and wait discipline:

- the signaling thread s is moved to the monitor's entry queue
- the unblocked thread u resumes executing



Condition checking under different signaling disciplines

Under the **signal and wait** discipline, it is guaranteed that the signaled condition holds when the unblocked thread resumes execution – because it immediately follows the signal

In contrast, under the **signal and continue** discipline, the signaled condition may no longer hold when the unblocked thread u resumes execution – because the signaling thread, or other threads, may change the state while continuing

- Correspondingly, there are different patterns for waiting on a condition variable signaled as `if (!buffer.isEmpty()) isNotEmpty.signal():`

Signal and wait:

```
// check once
if (buffer.isEmpty())
  isNotEmpty.wait();
// here !buffer.isEmpty()
```

Signal and continue:

```
// recheck after waiting
while (buffer.isEmpty())
  isNotEmpty.wait();
// here !buffer.isEmpty()
```

Signal all

The signal and continue discipline does not guarantee that a thread resuming execution after a `wait` will find that the condition it has been waiting for is true: the signal is only a "hint"

- In spite of this shortcoming, most (if not all) implementations of monitors follow the signal and continue discipline – mainly because it is simpler to implement

Monitors following signal and continue typically also offer a condition-variable method:

```
void signalAll(); // unblock all threads blocked on this condition
```

This tends to be inefficient, because many threads will wake up only to discover the condition they have been waiting for is still not true, but works correctly with the waiting pattern using a loop (which is still not as inefficient as busy waiting!)

More signaling disciplines

The signaling discipline determines what happens to a signaling thread s after it unblocks another thread u by signaling

Two variants of signal and continue and signal and wait are also sometimes used:

urgent signal and continue: s continues executing;
u is moved to the front of the entry queue of the monitor

signal and urgent wait: s is moved to the front of the entry queue of the monitor;
u resumes executing

To be precise:

- An urgent thread gets ahead of "regular" threads, but may have to queue behind other urgent threads that are waiting for entry
- This is implemented by adding a `urgentEntry` queue to the monitor, which has priority over the "regular" `entry` queue

Signaling disciplines: **Summary**

A signaling discipline defines what happens to three sets of threads:

S: signaling threads

U: unblocked threads

E: threads in the entry queue

Write $X > Y$ to denote that threads in set X have priority over threads in set Y

- Then, different signaling policies can be expressed as:

signal and continue	$S > U = E$
urgent signal and continue	$S > U > E$
signal and wait	$U > S = E$
signal and urgent wait	$U > S > E$

Other combinations are also possible, but most of them do not make much sense in practice

Implementing monitors

Monitors from semaphores

We give an overview of how to implement monitors using semaphores

- This also rigorously defines the semantics of monitors:
  - Every monitor class uses a strong semaphore `entry` to model the entry queue
  - Every monitor method acquires `entry` upon entry and releases it upon exit

```
monitor class Counter {
  int x = 0;
  public void inc() {
    x = x + 1;
  }
}
```

```
class Counter {
  // strong/fair semaphore, initially 1
  Semaphore entry = new Semaphore(1, true);
  private int x = 0;
  public void inc() {
    entry.down();
    x = x + 1;
    entry.up();
  }
}
```

Condition variables: Waiting

Every condition variable uses a queue blocked of threads waiting on the condition

```
abstract class WaitVariable implements Condition {
  Queue blocked = new Queue<Thread>(); // queue of blocked threads

  // block until signal
  public void wait() {
    entry.up();               // release monitor lock
    blocked.add(running);     // enqueue running thread
    running.state = BLOCKED;  // set state as blocked
  }

  // is no thread waiting?
  public boolean isEmpty() { return blocked.isEmpty(); }
}
```

Reference to running thread

Condition variables: Signal and continue

```
class SCVariable extends WaitVariable {
  // signal to unblock
  public void signal() {
    if (!blocked.isEmpty()) {
      Thread u = blocked.remove();  // u is the unblocked thread
      entry.blocked.add(u);         // u gets moved to entry queue
      // the running, signaling thread continues executing
    }
  }
}
```

The thread signaling continues its execution

Condition variables: Signal and wait

```
class SWVariable extends WaitVariable {
  // signal to unblock
  public void signal() {
    if (!blocked.isEmpty()) {
      entry.blocked.add(running);  // the running, signaling thread
                                   // gets moved to entry queue
      Thread u = blocked.remove(); // u is the unblocked thread
      u.state = READY;             // set state as ready to run
      running.state = BLOCKED;     // set state as blocked
      // the unblocked, signaled thread resumes executing
    }
  }
}
```

Semaphores from monitors

```
monitor class StrongSemaphore implements Semaphore {
 int count;
 Condition isPositive = new Condition(); // is count > 0?

 public void down() {
   if (count > 0)
     count = count - 1;
   else isPositive.wait();
 }


 public void up() {
   if (isPositive.isEmpty())
     count = count + 1;
   else isPositive.signal();
 }
}
```

Can we implement semaphores using monitors?

Each signal matches a **wait**;
thus no decrement or increment
in the `else` branches

Semaphores from monitors: A theoretical result

The result that monitors can implement semaphores (and vice versa) is important theoretically: **no expressiveness loss**

However, implementing a lower-level mechanism (semaphores) using a higher-level one (monitors) is impractical because it is likely to be inefficient

- If you have monitors use it (do not implement semaphores)

As usual, if you need monitors or semaphores use the efficient library implementations available in your programming language of choice

- Do not reinvent the wheel!

Monitors in Java

Two kinds of Java monitors

Java does not include full-fledged monitor classes, but it offers support to implement monitor classes following some programming patterns

There are two sets of monitor-like primitives in **Java**:

- language based: has been included since early versions of the Java language
- library based: has been included since Java 1.5

We have seen bits and pieces of both already, since they feature in simpler synchronization primitives as well

Language-based monitors

A class `JM` can implement a monitor class `M` as follows:

- every attribute in `JM` is **`private`**
- every method in `JM` is **`synchronized`** – which guarantees it executes atomically

```
monitor class M {
  int x, y;


  public void p()
  { /* ... */ }

  public int q()
  { /* ... */ }
}
```

```
class JM {
  private int x, y;


  public synchronized void p()
  { /* ... */ }

  public synchronized int q()
  { /* ... */ }
}
```

This mechanism does **not guarantee** fairness of the entry queue associated with the monitor: `entry` may behave like a set

Language-based condition variables

Each language-based monitor implicitly include a single condition variable with signal and continue discipline:

- calling `wait()` blocks the running thread, waiting for a signal
- calling `notify()` unblocks any one thread waiting in the monitor
- calling `notifyAll()` unblocks all the threads waiting in the monitor

```
monitor class M {
  int x; Condition isPos;
  public void p()
  { while (x < 0)
       isPos.wait(); }
  public int q()
  { if (x > 0)
       isPos.signal(); }
}
```

```
class JM {
  private int x;
  public synchronized void p()
  { while (x < 0)
       wait(); }
  public synchronized int q()
  { if (x > 0)
       notify(); }
}
```

It does **not guarantee** fairness of the blocked threads queue: `blocked` may behave like a set

How to wait in a language-based monitor

Calls to `wait()` always must be inside a loop checking a condition

- There are multiple reasons to do this:
  - Under the signal and continue discipline, the signaled condition may be no longer true when an unblocked thread can run
  - Since the `blocked` queue is not fair, the signaled condition may be "stolen" by a thread that has been waiting for less time
  - Since there is a single implicit condition variable, the signal may represent a condition other than the one the unblocked thread is waiting for
  - In Java (and other languages), spurious wakeups are possible: a waiting thread may be unblocked even if no thread signaled.

Library-based monitors

A class `LM` can implement a monitor class `M` using explicit locks:

- add a private `monitor` attribute – a fair lock
- every method in `CM` starts by locking `monitor` and ends by unlocking `monitor` – which guarantees it executes atomically

```
monitor class M
{
  int x, y;
  public void p()
  { /* ... */ }
}
```

```
class LM {
  private final Lock monitor = new ReentrantLock(true); // fair lock
  private int x, y;

  public void p()
  {
    monitor.lock();
    /* ... */
    monitor.unlock();
  }
}
```

This mechanism **guarantees** fairness of the entry queue associated with the monitor: `blocked` behaves like a queue

Library-based condition variables

Condition variables with signal and continue discipline can be generated by a monitor's lock:

```
monitor class M {

  Condition isXPos
    = new Condition();
  Condition isYPos
    = new Condition();

  int x, y;
  // ...
}
```

```
class JM {
  private final Lock monitor
    = new ReentrantLock(true);
  private final Condition isXPos
    = monitor.newCondition();
  private final Condition isYPos
    = monitor.newCondition();

  private int x, y;
  // ...
}
```

Library-based condition variables (cont'd)

Each library-based condition variable `c` has *signal and continue* discipline:

- calling `c.await()` blocks the running thread, waiting for a signal
- calling `c.signal()` unblocks any one thread waiting on `c`
- calling `c.signalAll()` unblocks all the threads waiting on `c`

- When `signalAll()` is called, the ordering of lock reacquisition is also fair (same order as in `blocked`) – provided the lock itself is fair
- These methods must be called while holding the lock used to generate the condition variable; otherwise, an `IllegalMonitorStateException` is thrown

This mechanism **guarantees** fairness of the queue of blocked threads associated with the condition variable: `blocked` behaves like a queue

How to wait in a library-based monitor

Calls to `await()` always must be inside a loop checking a condition

There are multiple reasons to do this (compare to the case of language-based monitors):

- Under the signal and continue discipline, the signaled condition may not be longer true when an unblocked thread can run
- In Java (and other languages), spurious wakeups are possible: a waiting thread may be unblocked even if no thread signaled

Threads, interrupted

Waiting operations (in monitors as well as in semaphores) may be **interrupted** by some low-level code that calls a thread's `interrupt()` method

- This is apparent in the signature of the waiting methods, which typically may throw an object of type `InterruptedException`: interrupting a waiting thread wakes up the thread, which has to handle the exception
- We normally ignore the case of interrupted threads, since it belongs to lower-level programming
  - When calling waiting primitives, you typically propagate the exception to the main method (or simply catch and ignore it)

Threads, interrupted (cont'd)

It is important that programs ensure that an interrupted thread still leaves the system in a consistent state by releasing all locks it holds

- In language-based monitors, an interrupted thread in a `synchronized` method automatically releases the monitor’s lock
- In library-based monitors, use a `finally` block to release the monitor’s lock in case of exception:

```
class LM {
  private final Lock monitor = new ReentrantLock(true);


  public void p() {
    monitor.lock();
    try { /* ... */ }
    finally { monitor.unlock(); }
  }
}
```

Monitors: dos and don'ts

Nested monitor calls

What happens if a method in monitor `M` calls a method *n* in monitor `N` (with condition variable `cN`)? Different rules are possible:

1. Prohibit nested calls
2. Release lock on `M` before acquiring lock on `N`
3. Hold lock on `M` while also locking `N`
   3.1 When waiting on `cN` release both locks on `N` and on `M`
   3.2 When waiting on `cN` release only lock on `N`

- Rules 3 are prone to deadlock – especially rule 3.2. – because deadlocks often occur when trying to acquire multiple locks

- Java monitors (both language- and library-based) follow the deadlock-prone rule 3.2
  - **Rule of thumb**: avoid nested monitor calls as much as possible
  - Note that if `N` is the same object as `M`, nested calls are not a problem (the implicit locks are reentrant)

Monitors: Pros

- Monitors provide a structured approach to concurrent programming, which builds atop the familiar notions of objects and encapsulation

- This raises the level of abstraction of concurrent programming compared to semaphores.

- Monitors introduce separation of concerns when programming concurrently:
  - mutual exclusion is implicit in the use of monitors,
  - condition variables provide a clear means of synchronization.

Monitors: Cons

- Monitors generally have a larger performance overhead than semaphores
  - Performance must be traded against error proneness

- The different signaling disciplines are a source of confusion, which tarnishes the clarity of the monitor abstraction. In particular, signal and continue is both less intuitive (because a condition can change before a waiting thread has a chance to run on the monitor) and the most commonly implemented discipline

- For complex synchronization patterns, nested monitor calls are another source of complications