http://cs.lth.se/EDA040

Lectures during HT 2011: EDA040

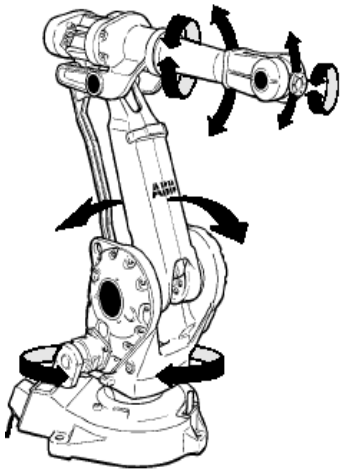
Concurrent and Real-time programming

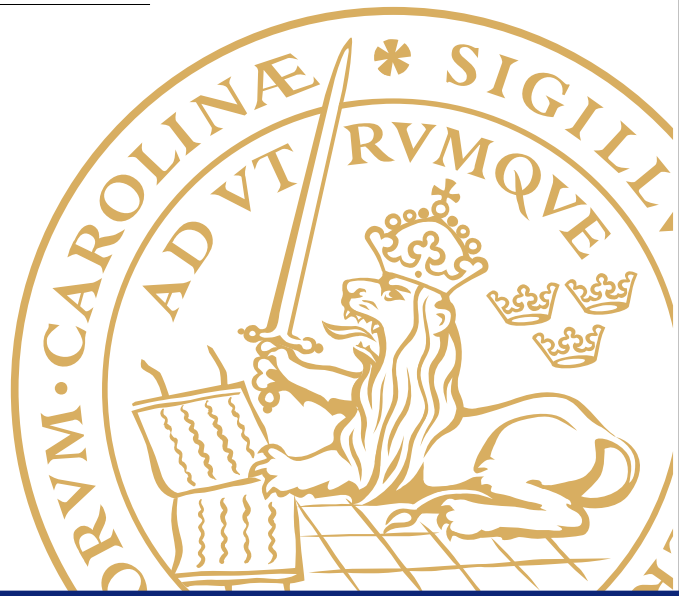
Department of Computer Science, Lund

klas@cs.lth.se

Lecture 4 [F4], part A: Mailboxes; message-based communication and synchronization

---





A special type of monitor is the buffer, forming a *mailbox for messages*

- *While monitors in general are for operations on shared data, a monitor with operations post (called by a producer thread) and fetch (called by consumer thread) comprises a data flow.*
- *Data can provide information and/or synchronization.*
- *Originally and traditionally data is then referred to as **messages**, and the buffer is a **mailbox**.*
- *Between threads (the same program and memory space) a message can be an* `Object` *ref.*

```
class Buffer // Providing mailbox
{
  synchronized void post(Object obj)
  {
    while (buff.size()>=maxSize) {
      wait();
    }
    if (buff.isEmpty()) notifyAll();
    buff.add(obj);
  }

  synchronized Object fetch()
  {
    while (buff.isEmpty()) {
      wait();
    }
    if (buff.size()>=maxSize) notifyAll();
    return buff.remove();
  }
}
```

Message sending - Mailboxes

- *Producer-Consumer relations* between threads are very common:
- Asymmetric synchronization (signaling) ; the producer should be allowed to continue without having to wait for the consumer.
- Transfer information – data referred to as a message.
- Thus, asynchronous communication (signaling plus data transfer) that provides **Buffering** and Thread/**Activity interaction**.
- Additionally, for complex systems today:
- **Distribution**: Threads are, or need to be prepared for being, distributed over several computers with network communication.
- **Encapsulation**: Concurrent and real-time properties of objects (handling timeout/overrun/exceptions etc.) requires means for message passing between concurrently running objects/threads.

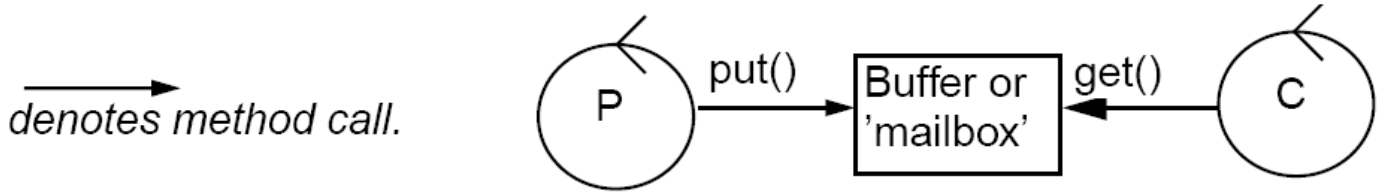
Message sending - Mailbox

Situations with Producer-Consumer relations between threads are very common.

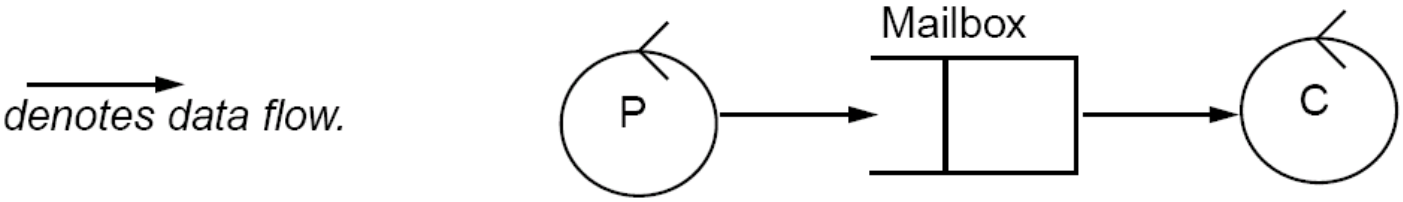
We want to achieve:

- Asymmetric synchronization - i.e. the producer should be allowed to continue without having to wait for the consumer.
- Transfer information - a message

So far we have used a Monitor/Buffer to achieve this:

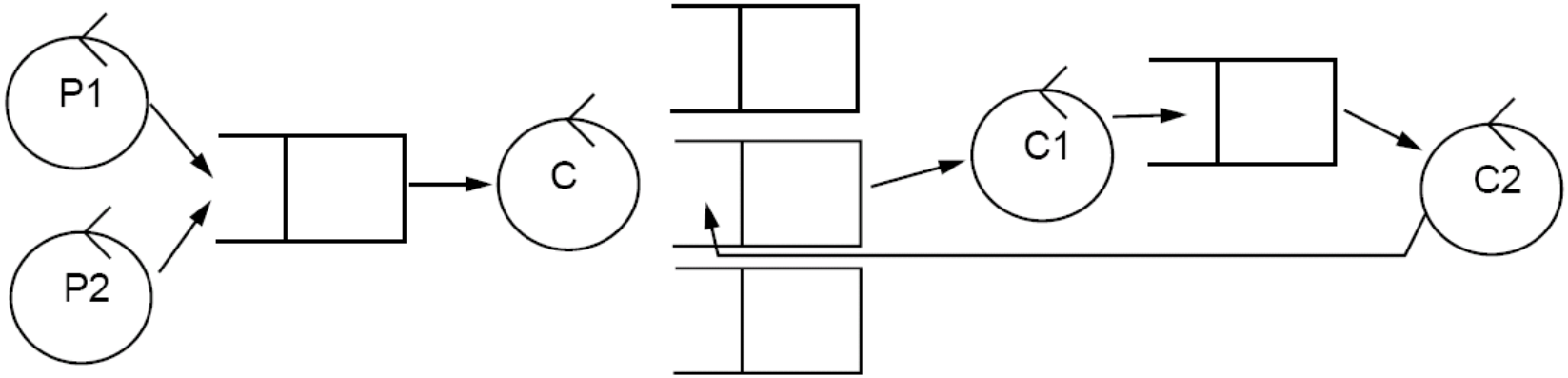


We introduce a special name, Mailbox (brevlåda), for this way of using a Monitor. We draw it somewhat differently:



Systems of mailboxes

Communication between threads often forms a network of mailboxes.



- The same principle for (operating system) processes and threads.
- A thread can put a letter in several mailboxes.
- A mailbox can, in Java, handle various types of messages - subclassing of message (`RTEvent`).
- A thread has in most cases only one mailbox it reads from (otherwise problems - fetching a messag is a blocking operation).
- Message objects need to be ***serialized*** (transformed into a stream of bytes) in order to be sent to another OS process.
- Within a OS process (between threads), we can send pointers/references or a copy of the object.
- How does the receiver know that another thread does not modify the contents of a message???

IPC (Inter-Process Communication)

	Local	Distributed
'Synchronous'	Object-method call	RPC/RMI
Synchronous	Monitor-method call	Database
Asynchronous	Event buffer	Stream (pipe/file/socket)

Synchronous handling of Event:

- Event model in Java (AWT and Beans. NOT concurrent).
- Corresponding EventObject for realtime: **RTEvent**
- Corresponding synchronous event handling in `se.lth.cs.realtime`:
  See class documentation for `RTEventListener`, `RTEventListenerList` and `JThread`.
- Single-threaded 'synchronous' event handling is not a central issue in the course.

Unbounded mailbox with copy-on-send

Advantages with unbounded buffers:

Automatic buffering – dynamical allocation of messages (can be a problem)

Practical when every message can be dealt with separately

The same mechanism can be used for communication between OS processes running on the same computer or different ones (distributed systems):

- Does not assume shared memory, the messages can be copied.
- Asymmetric synchronization.

Disadvantage:

Often unpractical when immediate response is required (i.e. synchronous communication).

Notice:

If GC is unavailable, it is desired to decrease the required GC work, a ’pool’ is often used for recycled messages.

The amount of messages IS limited (limited amount of memory) - a high priority thread running amok can fill the entire memory / run out of messages.

We use a ’bounded buffer’ in the form of **RTEventBuffer** in shared memory.

Mailbox == Monitor == Semaphore

A Mailbox can easily be implemented using a Monitor

Also a Semaphore is sort of a monitor.

Suppose we only send empty messages, then a Mailbox is equivalent to a Semaphore:

- The value of the counter of the Semaphore corresponds to the number of messages in the mailbox.
- Send message - give()
- Receive message - take()

All three constructions are thus equally powerful, but practical in different situations.

Events as messages

`java.util.EventObject` comprises an event class that is suitable for messages, providing a `transient` (will be `null` outside JVM) **`source`**, referring to the sending object/thread.

`se.lth.cs.realtime.event.`**`RTEvent`** is a subclass that, as `java.awt.InputEvent,` has a **`timestamp`**, expressing object age.

- We use such timestamped events for asynchronous communication between threads.

Note that graphics such as swing is basically single-threaded!

The RTEvent class

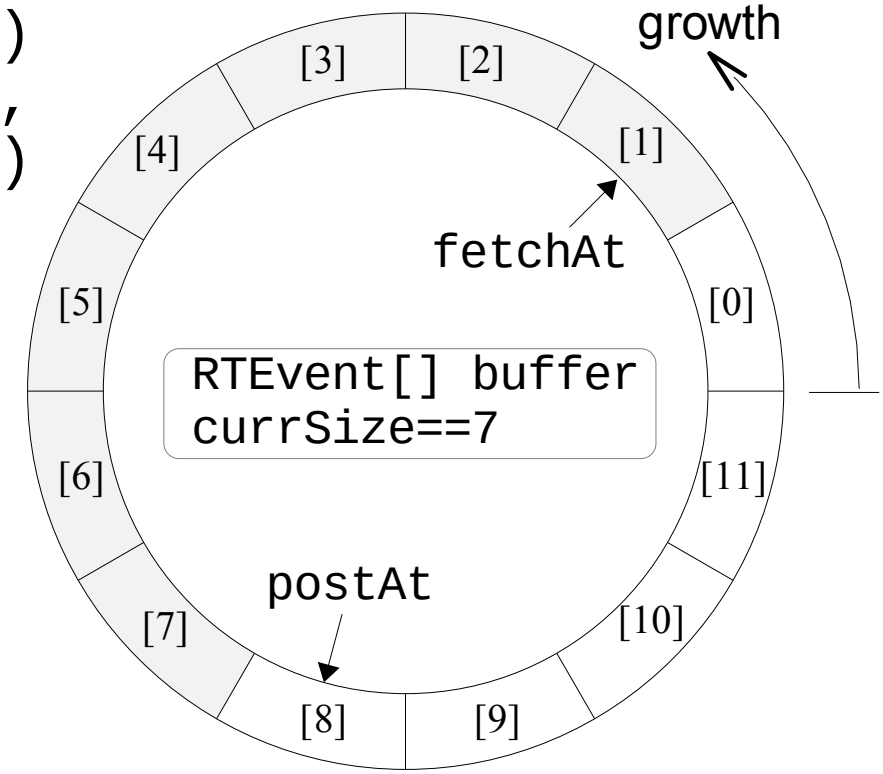
```
public abstract
class RTEvent extends EventObject {
  protected long timestamp;                    // Creation time in ms.
  protected volatile transient Object owner; // Responsible thread.
  public RTEvent();                 // Use current Thread & TimeMillis.
  public RTEvent(Object source); // Set source, default timestamp.
  public RTEvent(long ts);          // Set timestamp, default source
  public RTEvent(Object source, long ts); // Override both defaults.
  public final Object getOwner()
  public double getSeconds()
  public long getMillis()
  public long getNanos()
}
```

The RTEventBuffer class

- As `RTEvent`, part of `se.lth.cs.realtime.`**`event`**
- Constructors:

```
public RTEventBuffer()
RTEventBuffer(int maxSize)
RTEventBuffer(int maxSize,
              Object lock)
```

- Example with `maxSize==12` and `currSize==7`, internal attributes:
- Obtain message/event by
  `RTEvent` **`fetch`**`()`
  or specific final methods.
- Send message/event by
  `RTEvent` **`post`**`(RTEvent ev)`
  or specific final methods.



More RTEventBuffer / mailbox

- Blocking and non-blocking methods for posting and fetching messages:

```
doPost(RTEvent e)    // Add e to queue, blocks if the queue is full.
tryPost(RTEvent e)   // Adds to the queue, without blocking if full.
doFetch()            // Fetch from queue, block if empty.
tryFetch()           // Fetch without blocking if empty.
awaitEmpty()         // Waits for buffer to become empty.
awaitFull()          // Waits for buffer to become full.
isEmpty()            // Checks if buffer is empty.
isFull()             // Checks if buffer is full.
```

- The **try**-Post/Fetch returns `null` if the buffer is non-full/empty, and the supplied/next event otherwise, respectively.
- The attributes are declared protected in order to make it possible to create subclasses with revised functionality.

A producer

```
class Producer extends Thread {
  Consumer receiver;
  MyMessage msg;
  public Producer(Consumer theReceiver) {
    receiver = theReceiver;
  }
  public void run() {
    while (true) {
      char c = getChar();
      msg = new MyMessage(c);
      receiver.putEvent(msg);
    }
  }
}
```

```
class MyMessage extends RTEvent {
  character ch;
  public MyMessage(char data) {
    super(); // Set time stamp;
    ch = data;
  }
}
```

A consumer

```
class Consumer extends Thread {
  RTEventBuffer mailbox;
  public Consumer(int size) { mailbox = new RTEventBuffer(size); }
  public void putEvent(RTEvent ev) {
    mailbox.post(ev); // In context of Producer
  }
  public void run() {
    RTEvent m;
    while (true) {
      m = mailbox.fetch(); // In context of Consumer
      if (m instanceof MyMessage) {
        MyMessage msg = (MyMessage) m;
        useChar(msg.ch);
      } else { ... // Handle other messages
      };
    } // ...
```

The JThread utility class

- Part of the `se.lth.cs.realtime` package
- Subclass of `java.lang.Thread`; thus it is a **J**ava **Thead**, hence **`JThread`**.
- Encapsulates an `RTEventBuffer`, exposed via a public **`putEvent`** method.
- Default `run` method is a cyclic call of `perform`
- Internally the `perform` (or `run`) method refers to the `mailbox` attribute like

```
event = mailbox.doFetch();
```

- Methods `sleepUntil` and `terminate` are also provided (compare lab).