http://cs.LTH.se/EDA040

Real-Time and Concurrent Programming

Lecture 5 (F5):

Part1: Deadlock   Part 2: Messages

Klas Nilsson

2015-09-29

Part I

Deadlock

Content

Background

Mutual exclusion means that a thread can be delayed

- A thread will not be allowed to enter a critical region (using a shared resource) as long as it is occupied by another thread.
- For consistency (concurrency correctness), predictability (real-time correctness), and for efficiency (embedded computing), access of such a locked resource may not be interrupted or subject to a *roll-back*.

Hence, we have no preemption on resources (only on time as in preemptive scheduling; not to be confused).

- If the blocking never ends we have a **Deadlock** (Swe: Dödläge)

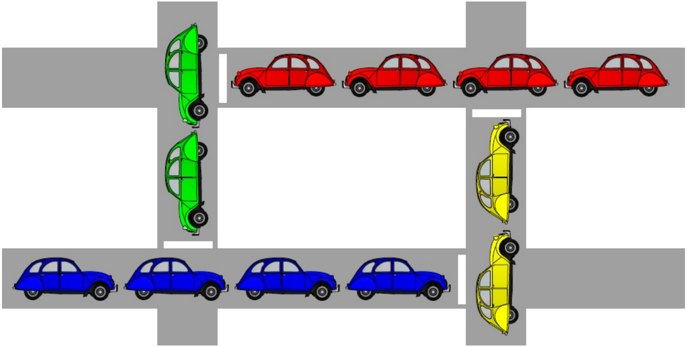
Wikipedia: Deadlock refers to a specific condition when two or more processes are each waiting for the other to release a resource, or more than two processes are waiting for resources in a circular chain.

The problem

If waiting can be or is circular:

- When several threads can be waiting for each other we have a Deadlock risk.
- When several threads are waiting for each other we have a Deadlock.

Thus, circular wait appears to be related too deadlock:



Example: deadlock with semaphores

P1	**P2**
`S1.take();`	`S2.take();`
`S2.take();`	`S1.take();`
`...`	`...`
`S2.give();`	`S1.give();`
`S1.give();`	`S2.give();`

P1:	P2:	
S1.take();		
	S2.take();	
	S1.take();	*blocked*
S2.take();		*blocked*

- *Deadlock may occur if*: one thread performs a take, followed by a context switch (swe: trådbyte).

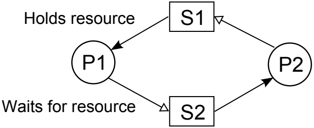
Example cont'd: resource allocation graph

P1

```
S1.take();
S2.take();
...
S2.give();
S1.give();
```

P2

```
S2.take();
S1.take();
...
S1.give();
S2.give();
```



Method: draw resources (boxes) and threads (circles). Draw arrows for **hold** (filled) + **wait** (outlined).

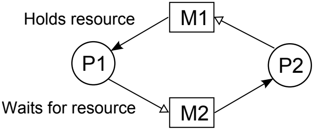
Deadlock with monitors

P1

```
m1.op1();
```

P2

```
m2.op1();
```




```
class M1 {
  synchronized void op1() {
    m2.op2();
  }
  synchronized void op2() {
    wait();
  }
}
```

```
class M2 {
  synchronized void op1() {
    m1.op2();
  }
  synchronized void op2() {
    wait();
  }
}
```

Necessary conditions for deadlock

Necessary conditions for deadlock to occur:

1. Mutual Exclusion - only one thread can access a resource at a time.
2. Hold and Wait - a thread can reserve a resource and wait for another.
3. No *resource* preemption - a thread can not be forced to release held resources.
4. Circular Wait - thread-resources dependencies must be circular.

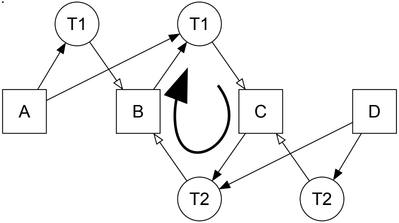
Monitor - satisfied conditions:

1. Monitor - one thread only is allowed to enter at a time.
2. Call of an operation in a monitor from inside a monitor operation in another monitor.
3. A monitor can only be released if a thread voluntarily waits (`wait()`) or exits the monitor.
4. But 4? Must prevent circular wait that can result in deadlock.

Resource allocation graphs

Tool for detecting circular hold-wait situations and to determine under which conditions deadlock can occur.

1. Draw resources
2. Draw all hold-wait situations (arrows from each held resource to a thread marker, arrows from thread marker to resource waited for)
3. Circular? Then risk for deadlock. The number of 'hold-wait' links in the circular chain shows how many and which threads are required for deadlock.



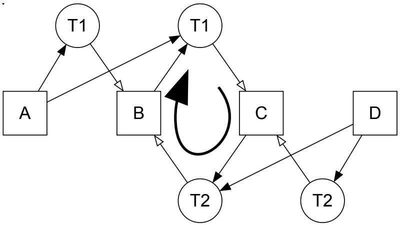
Resource allocation graphs - example

Thread 1

```
A.take();
B.take();
C.take();
C.give();
B.give();
A.give();
```

Thread 2

```
D.take();
C.take();
B.take();
B.give();
C.give();
D.give();
```



Conclusion: Deadlock possible when T1 is waiting for C and T2 is simultaneously waiting for B! Circular wait!

Monitors in Concurrent Pascal

Concurrent Pascal (Per-Brinch Hansen 1979) only has monitors and the rule: *no forward references*, i.e. the program we looked at earlier is illegal:

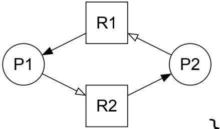
```
class M1 {
  synchronized void op1() {
    // Illegal since it introduces
    // a forward reference
    m2.op2();
  }
  synchronized void op2() { ... }
}
class M2 {
  synchronized void op1() {
    m1.op2();
  }
  synchronized void op2() { ... }
}
```

- Monitors, Semaphores, etc. are often referred to as *resources*.
- Generally: all resources is assigned a (partial) order, only allocate from lower to higher.
- M2 can call M1 but not the other way around.

Limitations in the language - a good idea?

- Deadlock impossible in Concurrent Pascal with Monitors as resources.
- Often inefficient and unpractical - might be necessary to prematurely allocate resources just to satisfy the demands on allocation order.
- Easy to implement ones own resource management using Monitors:

```
/*monitor*/ class R {
  boolean occupied;
  synchronized void request() {
    while (occupied) wait();
      occupied = true;
    }
  synchronized void release() {
    occupied = false;
    notify();
  }
}
```




```
R R1,R2;
class P1 extends Thread {
  ...
  R1.request();
  R2.request();
  ...
}
class P2 extends Thread {
  ...
  R2.request();
  R1.request();
  ...
}
```

Avoiding deadlock

- Language and library support not applicable.
- Run-time deadlock detection: Not suitable for real-time or embedded systems (useful in a generic OS, but here it would be too late).
- Instead, remove the risk for deadlock:
  - Create and analyze the resource-allocation graph.
  - Arrange the order of allocations, according to a resource ordering, preferably without extending the locking times.
  - If really necessary: add logic that prevents the dead-lock.

Notions

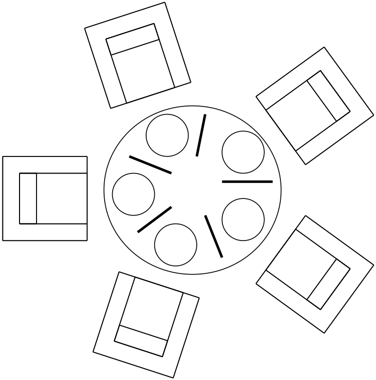
- Deadlock (swe: dödläge)
  - When several resources attempts to allocate the same resource one must be able to get it.
  - Bad enough if there exists an execution order such that Deadlock occurs - even if it happens only seldom. The system locks, hangs, nothing happens. Can apply to subsystems.
- Starvation (swe: svält)
  - If a thread attempts to allocate a resource it must be able to get it eventually.
  - We renounce the 'no starvation' property in favor of priority; less important activities might suffer from starvation.
- Livelock
  - Occurs when several threads attempts to allocate the same resource but none actually gets it due to the execution pattern.
  - Behaves like Deadlock, but if you study the system closely the threads actually run. They do no meaningful work though.

Dining Philosophers problem

- The life of a philosopher is boring:

```
class Philosopher
  while
    think();
    preProto();
    eat();
    postProto();
```

- Logical spaghetti:
  - Two forks are required to eat
- Solution requirements:
  - No deadlock
  - No starvation
  - Efficient



Semaphore solution 1

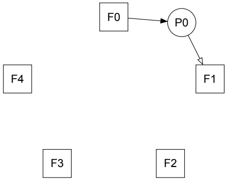
```
Semaphore[] fork = new MutexSem[5];
for (int i=0; i<5; i++) fork[i] = new MutexSem();
```

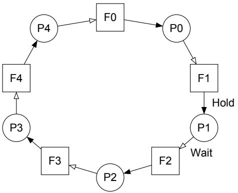
```
class Philosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}

  public void run() {
    while (true) {
      think();
      fork[i].take();
      fork[(i+1)%5].take();
      eat();
      fork[i].give();
      fork[(I+1)%5].give();
    }
  }
}
```

- Eat() acts as a critical region for a pair of forks, but for different pairs of forks
  - mutual exclusion for both
- Can this solution cause Deadlock? Two resources is required for each activity (hold-wait satisfied) so we have to make a more detailed analysis.

Philosophers 1 - deadlock?

- Draw a complete allocation graph, all Hold-Wait dependencies





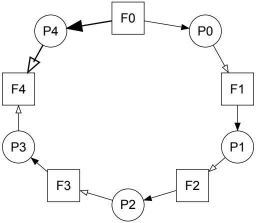
- Circular → unsafe program, Deadlock can occur.
- Step 2: Can we present a scenario where deadlock occurs or prove that the situation can not occur in practice?
- Scenario: Suppose all 5 philosophers starts simultaneously, takes their left forks and then (all of them) waits for their right forks.
- This solution can thus cause deadlock.
- Can we solve the problem in a better way?
  - avoid circularity or
  - make sure that all Hold-Wait can not occur simultaneously?

Philosophers 2 - one left handed philosopher

```
Semaphore[] fork = new MutexSem[5];
for (int i=0; i<5; i++) fork[i] = new MutexSem();
```

```
class Philosopher
  extends Thread {
  int i;
  Philosopher(int i)
    {this.i=i;}
  public void run() {
    while (true) {
      think();
      fork[i].take();
      fork[i+1].take();
      eat();
      fork[i].give();
      fork[i+1].give();
    }
  }
}
```

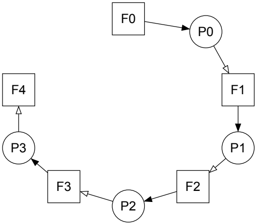
```
class LeftPhilosopher
  extends Thread {
  int i;
  LeftPhilosopher(int i)
    {this.i=i;}
  public void run() {
    while (true) {
      think();
      fork[0].take();
      fork[4].take();
      eat();
      fork[4].give();
      fork[0].give();
    }
  }
}
```



- No circular dependency
- No deadlock

Philosophers 3 - only four chairs

```
Semaphore[] fork = new MutexSem[5];
for (int i=0; i<5; i++) fork[i] = new MutexSem();
Semaphore room = new CountingSem(4);

class Philosopher extends Thread {
  int i;
  Philosopher(int i) {this.i=i;}
  public void run() {
    while (true) {
      think();
      room.take();
      fork[i].take();
      fork[(i+1)%5].take();
      eat();
      fork[i].give();
      fork[(i+1)%5].give();
      room.give();
    }
```



Complete allocation graph cyclic, ‘unsafe’ as before, but: At most four Hold-Wait can be active simultaneously → at least one philosopher can eat, no deadlock possible.

Philosophers 4 - polite philosophers

A philosopher only picks up the forks and starts to eat if BOTH forks are free.

- Implemented using a monitor or a *MultistepSem*.
  - Trivially deadlock free since no Hold-Wait situations occur.
  - but, starvation possible.
- Suppose two philosophers, e.g. 1 and 3 agrees to eat alternating:
  - I.e. philosopher 1 eats until philosopher 3 has begun to eat, and the other way around
  - Now will philosopher 2 never have two forks free at the same time, i.e. **philosopher 2 will starve!!**

Part II

Message-based communication and synchronization

6 Mailboxes and messages
- Buffering and asynchronous interaction
- System aspects

7 Events and Buffers
- Messages within a program – Event objects

8 Examples
- Dataflows: Producer – Consumer

The buffering monitor as 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

Reasons for message-based interactions between threads:

- Producer-Consumer relations (data flows) 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 message content.
- Thus, asynchronous communication (signaling plus data transfer) that provides **Buffering** and Thread/Activity interaction.

Additionally, for complex systems today:

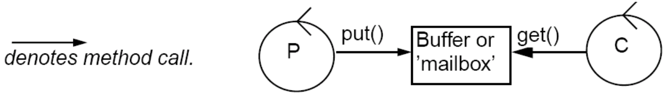
- **Distribution:** Threads are, or need to be prepared for being, distributed over several computers with network communication.
- **Encapsulation:** Concurrent and real-time properties of objects (handling timeout/overrun/exceptions etc.) requires means for message passing between concurrently running objects/threads.

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

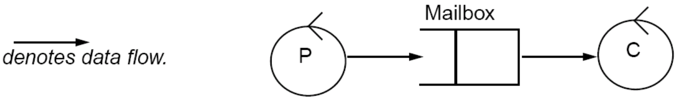
We want to achieve:

- 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:

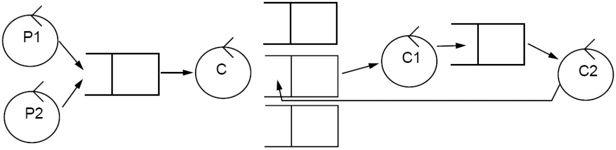


We introduce a special name, Mailbox (brevlåda), for this way of using a Monitor. We draw it somewhat differently:



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 message 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, Swing Beans) basically 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

- Flexible code; size of buffer does not need to be decided.
- Thread safety; sent message not accessable by sender.
- The same mechanism can be used for communication between OS processes running on the same computer or different ones (distributed systems), since shared memory is not assumed.

Disadvantages

- Higher risk for running out of memory, detected later. (Memory is limited, so better set fixed bounds early.)
- Often unpractical when immediate response is required (i.e. synchronous communication).
- Increased memory use, CPU for copying, and GC work.
- Recycling via message pools difficult to implement.

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.

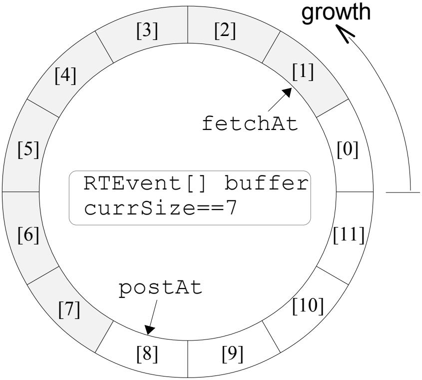
Note that graphics such as swing is basically single-threaded!

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 for `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;
  }
}
```

Note: Buffering is hidden by `putEvent` as of the receiving thread.

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 Java `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 lab1).