Synchronization concepts

We will describe classical synchronization concepts, used in operating system design. By showing, how these concepts can be realized in Java, programming applications can be done independently of operating system boundaries.

Table of contents

1. Overview 3
2. Semaphore 5
  2.1. Mutual exclusion 8
  2.2. Execution sequence of Threads 13
  2.3. Additive Semaphores 23
  2.4. Semaphore Groups 28
3. Message Queues 33
  3.1. Buffer of N elements 33
  3.2. Message queue implementation 38
4. Pipes 42

5. Dining philosophers 46

1. Overview

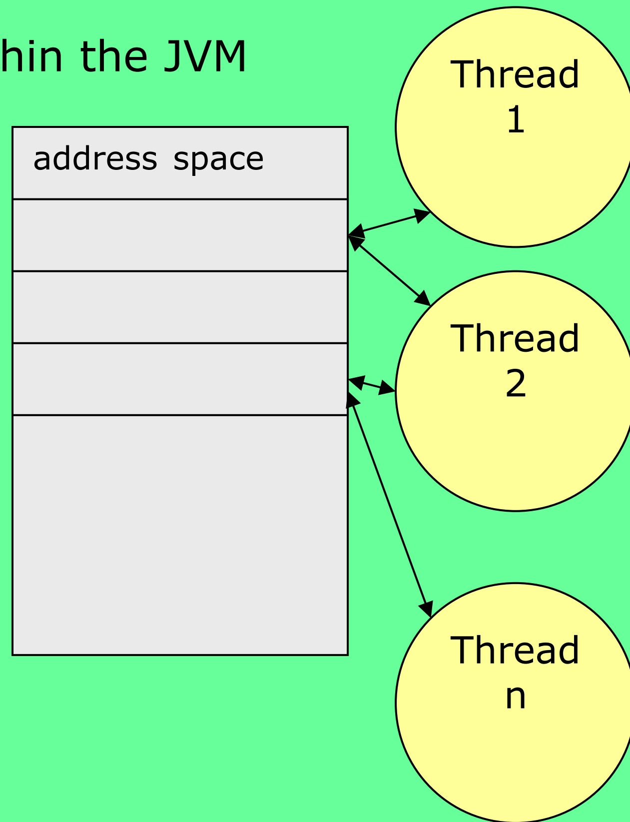
Processes frequently need to **communicate** with other processes. An **operating system** has built in features for the **interprocess communication** (IPC). In some operating systems, processes are working together often share some common storage that each one can read and write. Other operating systems let processes do not share such common storage.

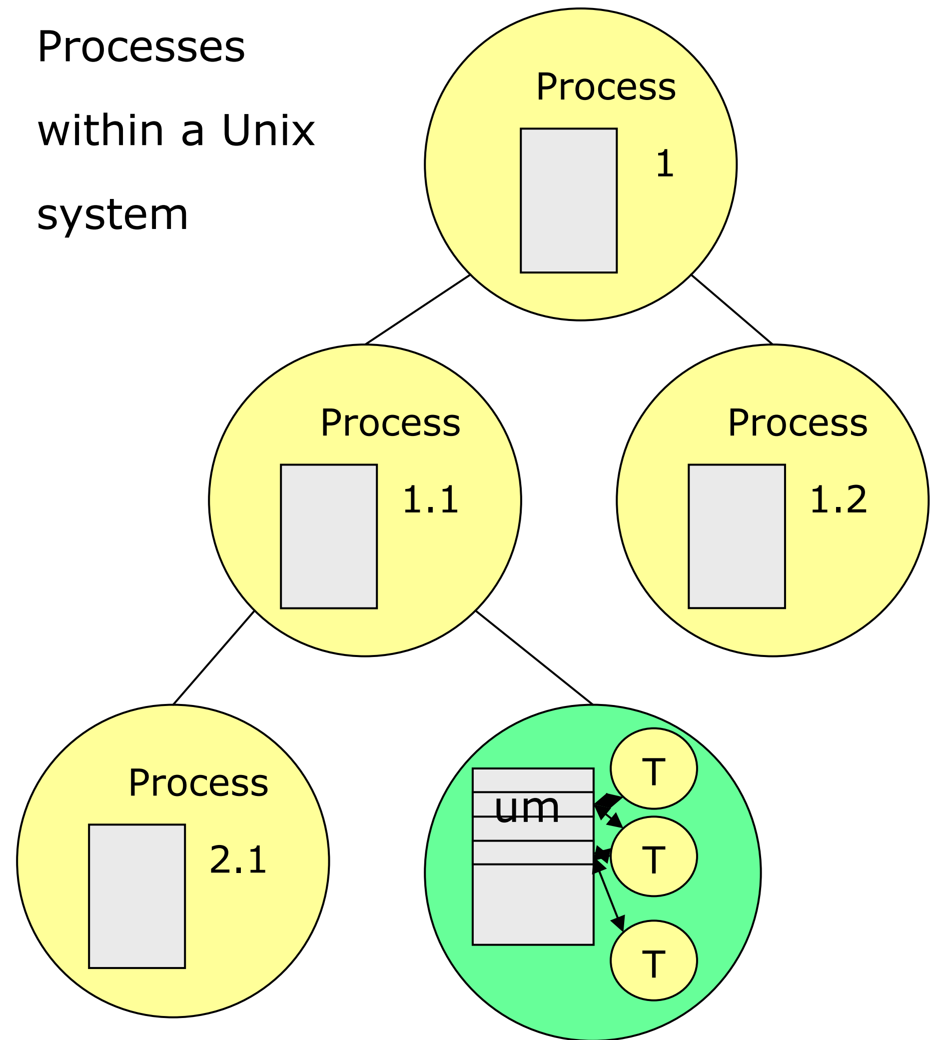
In Unix, **each process** has its own **address space**. When a Java program runs under Unix, the JVM manages the threads, created within this Java program.

We will consider how the operating system communication concepts

- Semaphores,
- Message Queues and
- Pipes

can be realized using Java Threads. In this scenario, we always use threads of one operating system process, as demonstrated into the picture:


Threads
within the JVM
address space
Thread 1
Thread 2
Thread n


Processes
within a Unix
system
Process 1
Process 1.1
Process 1.2
Process 2.1
um
T
T
T

2. Semaphore

A situation where two or more threads are reading or writing some shared data and the final result depends on who runs precisely when, is called **race condition** (see our bank account example)

Several solutions to avoid race conditions have been implemented. We consider semaphores.

In 1965 E.W. Dijkstra suggested the concept of **semaphores** to synchronize processes:

A semaphore is an **integer variable** to count the number of wakeups saved for future use.

A semaphore could have the **value 0**, indication that **no wakeups** where saved, or some **positive value** if one or more **wakeups** are **pending**.

Dijkstra proposed having two **operations**, **DOWN** and **UP** (generalizations of SLEEP and WAKEUP).

The operation **DOWN** on a semaphore checks to see if the value is greater than 0, it **decrements** the value and continues. If the value is 0, the thread is put to sleep (wait). Checking the value, changing it, and possibly going to sleep is all done as single, indivisible, **atomic action**.

The **UP** operation **increments** the value of the semaphore addressed. If one ore more threads where sleeping on that semaphore, unable to complete an earlier DOWN operation, **one of them** is chosen by the system and is allowed to **complete its DOWN** operation (wakeup). Thus, after UP on a semaphore with thread sleeping on it, the semaphore will still be 0, but there will be one fewer thread sleeping on it. Incrementing the semaphore and wakeup are also indivisible.

So far we have the concept. We now have to realize that behavior in Java.

The idea is:

> A **semaphore** is realized as a **class** with an **integer attribute** and **methods** for **UP** and **DOWN**. A call of DOWN blocks the calling thread, if the integer would become negative. A UP invocation wakes up a waiting thread.

```
$ cat Semaphores.java
public class Semaphore {
        private int value;

        public Semaphore(int init) {
                if(init < 0)
                        init = 0;
                value = init;
        }

        public synchronized void down() { // Dijkstra's operation p=down
                while(value == 0) {
                        try {
                                wait();
                        }
                        catch(InterruptedException e) {}
                }
                value--;
        }

        public synchronized void up() { // Dijkstra's operation v=up
                value++;
                notify();
        }
}
$
```

Now we can use semaphores to solve a problem implementing parallel programs: **mutual exclusion**.

2.1. Mutual exclusion

Mutual exclusion is some way of making sure that if one thread is using a shared object, the other threads will be excluded from doing the same thing. Mutual exclusion scenarios can be implemented in Java using `synchronized`. Here, we make use of the semaphore class, realized before.

We construct a program, where a **critical section** may be entered from **at most** one thread. The **activities** within the critical section are simulated by letting the thread **sleep** – in a real life application, the access to the shared date would occur in that section.

We realized the program in a general way to be able to specify **the number of threads**, which can simultaneously enter the critical section as **argument** of the **main** method.

```
$ cat MutualExclusion.java
class MutexThread extends Thread {
      private Semaphore mutex;

      public MutexThread(Semaphore mutex, String name) {
            super(name);
            this.mutex = mutex;
            start();
      }

      public void run() {
            while(true) {
                  mutex.down();
                  System.out.println("Enter critical section: " + getName());
                  try {
                        sleep((int)(Math.random() * 100));
                  }
                  catch(InterruptedException e) {}
                  System.out.println("Leave critical section: " + getName());
                  mutex.up();
            }
      }
}
```

```
down() {
   while(value == 0) {
     wait();
   }
   value--;
}
```

a semaphore protects the critical section

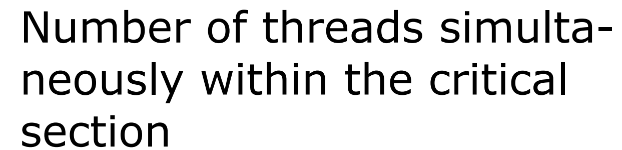
critical section

```
up() {
  value++;
  notify();
}
```

```
public class MutualExclusion {
      public static void main(String[] args) {
          int noThreadsInCriticalSection=1;
          if (args.length != 1) {
              System.err.println(
                 "usage: java MutualExclusion <NoThreadsInCriticalSection>");
                        System.exit(1);
              } else
                  noThreadsInCriticalSection = Integer.parseInt(args[0]);

              Semaphore mutex = new Semaphore(noThreadsInCriticalSection);
              for(int i = 1; i <= 10; i++) {
                    new MutexThread(mutex, "Thread " + i);
              }
      }
}
$
```

Calling the program produces an output, showing that only as many threads can enter the critical section as specified as argument of the call:

```
$ java MutualExclusion 1
Enter critical secton: Thread 1          // 1
Leave critical section: Thread 1         // 0
Enter critical secton: Thread 1          // 1
Leave critical section: Thread 1         // 0
Enter critical secton: Thread 3          // 1
Leave critical section: Thread 3         // 0
Enter critical secton: Thread 1          // 1
Leave critical section: Thread 1         // 0
Enter critical secton: Thread 6          // 1
Leave critical section: Thread 6         // 0
…
CTR C
$ java MutualExclusion 3
Enter critical secton: Thread 1          // 1
Enter critical secton: Thread 2          // 2
Enter critical secton: Thread 3          // 3
Leave critical section: Thread 1         // 2
Enter critical secton: Thread 1          // 3
Leave critical section: Thread 1         // 2
Enter critical secton: Thread 4          // 3
Leave critical section: Thread 4         // 2
Enter critical secton: Thread 1          // 3
Leave critical section: Thread 3         // 2
…
```



What happens, calling: `java MutualExclusion 0` ?

Classroom exercise:

Use the following program of exercise one to protect the critical section using the class semaphore.

```
class Even {
        private int n = 0;
        public int next() {         // POST?: next is always even
                ++n;
                try { Thread.sleep(10);
                } catch(InterruptedException e)  { }
                ++n;
                return n;
        }
}
public class Even2 extends Thread {
        private Even e;
        public Even2(Even e) {
                this.e = e;
        }
        public  void run() {
                for (int i = 1 ; i <= 10;  i++) {
                        System.out.println("result: " + e.next());
                }
        }
        public static void main(String[] args) {
                Even e = new Even();
                Even2 t1 = new Even2(e); Even2 t2 = new Even2(e);
                t1.start(); t2.start();
      }
}
```

```
$ java Even2
result: 3
result: 5
result: 7
…
result: 23
result: 25
result: 27
result: 29
result: 31
result: 33
result: 35
result: 37
result: 39
result: 40
$
```

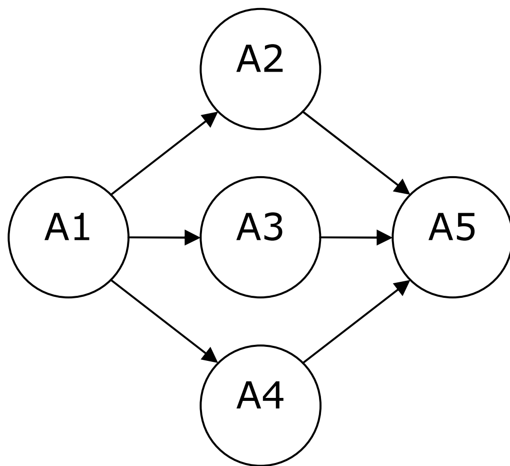
2.2. Execution sequence of Threads

In some application scenarios threads start working in parallel, however there are **dependencies** where some threads have to be finished before other threads are able to continue.

To visualize these dependencies, a dependency graph can be used:

- **nodes** of the graph are **threads**,
- an **edge** exists **from** thread **T1 to** thread **T2**, if **T1** must **have finished** its activities before T2 is able to start.

Example of a dependency graph:

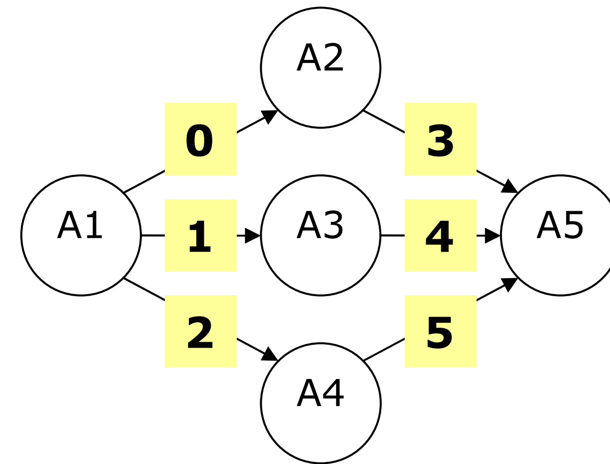


A1 has to be terminated before A2, A3 or A4 can start.

Start of A5 depends on termination of A2, A3 and A4.

We try to realize that situation in Java using semaphores:

We use for each **edge** of the dependency graph a **semaphore**. All semaphores are grouped in the array **`sems`** as shown on the right.

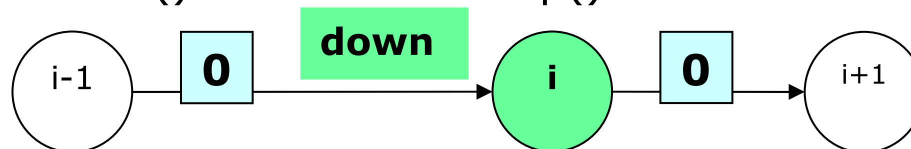


Before doing an action, the **down** operation of all “incoming” semaphores is performed. **After** the action, we perform the **up**-operation for all “outgoing” semaphores.

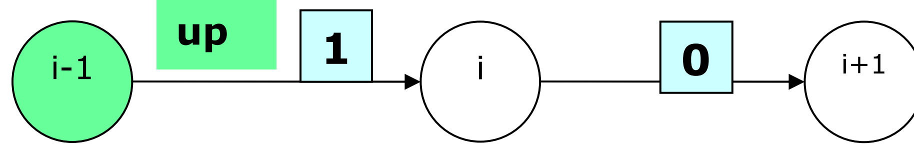
To make implementation easy, all threads use a reference to the semaphore array, even if not all threads use each semaphore. Further, we have to initialize the array with 0 (why?).

From the point of view of thread i, we do:

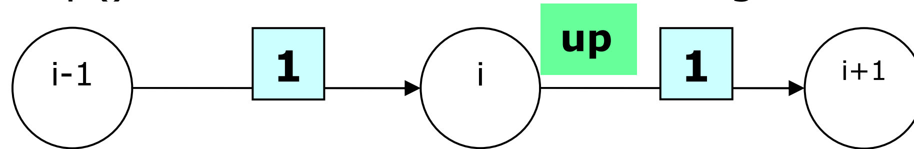
1. i.down(): wait until i-1.up()



2. i-1.up(): i is able to start its actions

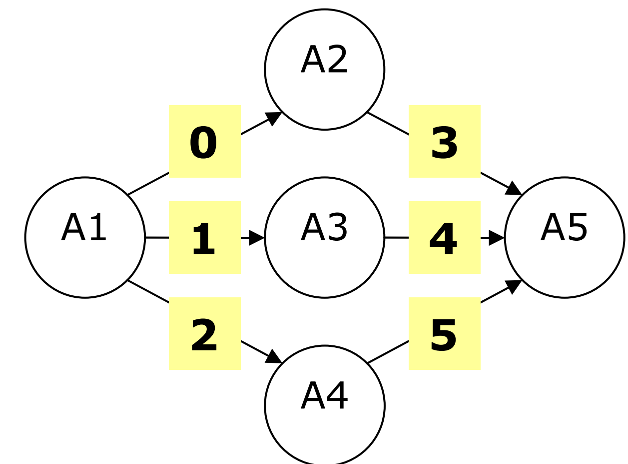


3. i.up(): i+1 is able to do something

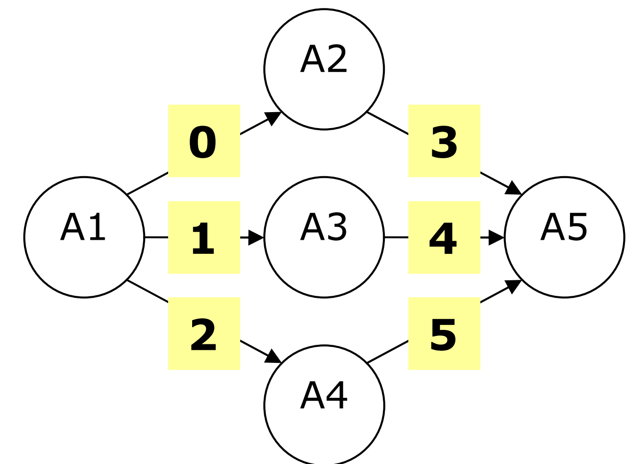


The realization in Java is not complicate:

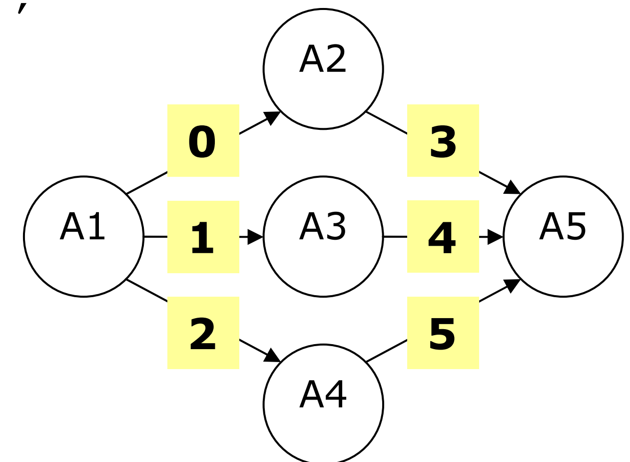
```
$ cat TimingRelation
class T1 extends Thread {
        private Semaphore[] sems;

        public T1(Semaphore[] sems) {
                this.sems = sems;
                start();
        }

        private void a1() {
                System.out.println("a1");
                 try {
                         sleep((int)(Math.random() * 10));
                 } catch(InterruptedException e) {}
        }

        public void run() {
                a1();
                sems[0].up();
                sems[1].up();
                sems[2].up();
        }
}
```



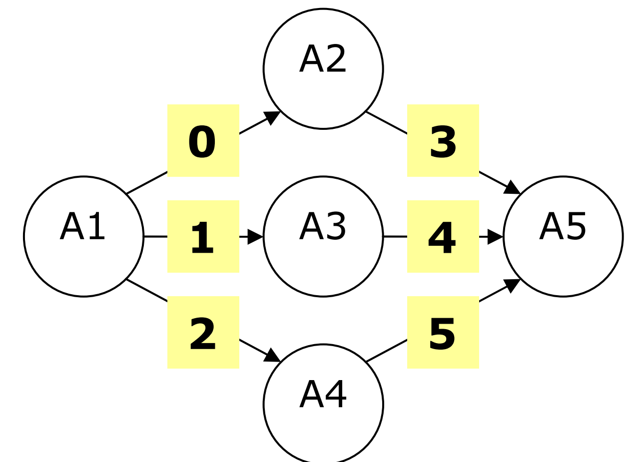
```java
class T2 extends Thread {
        private Semaphore[] sems;

        public T2(Semaphore[] sems) {
                this.sems = sems;
                start();
        }

        private void a2() {
                System.out.println("a2");
                 try {
                         sleep((int)(Math.random() * 10));
                 } catch(InterruptedException e) {}
        }

        public void run() {
                sems[0].down();
                a2();
                sems[3].up();
        }
}
```



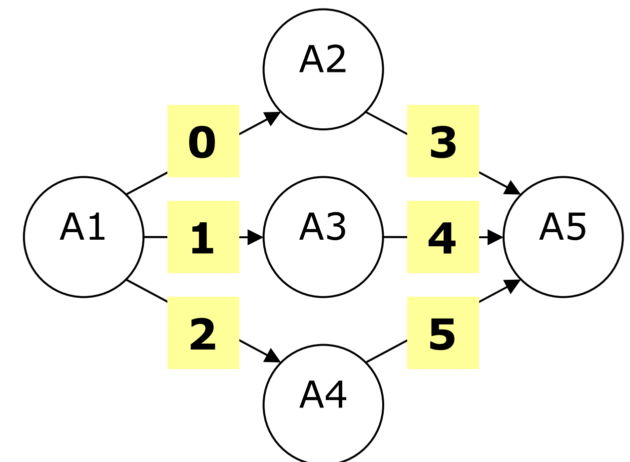
```java
class T3 extends Thread {
        private Semaphore[] sems;

        public T3(Semaphore[] sems) {
                this.sems = sems;
                start();
        }

        private void a3() {
                System.out.println("a3");
                 try {
                         sleep((int)(Math.random() * 10));
                 } catch(InterruptedException e) {}
        }

        public void run() {
                sems[1].down();
                a3();
                sems[4].up();
        }
}
```



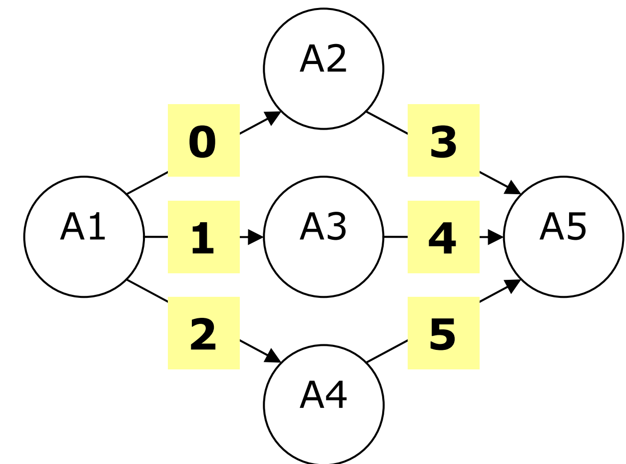
```
class T4 extends Thread {
        private Semaphore[] sems;

        public T4(Semaphore[] sems) {
                this.sems = sems;
                start();
        }

        private void a4() {
                System.out.println("a4");
                 try {
                        sleep((int)(Math.random() * 10));
                 } catch(InterruptedException e) {}
        }

        public void run() {
                sems[2].down();
                a4();
                sems[5].up();
        }
}
```



```
class T5 extends Thread {
        private Semaphore[] sems;

        public T5(Semaphore[] sems) {
                this.sems = sems;
                start();
        }

        private void a5() {
                System.out.println("a5");
                 try {
                         sleep((int)(Math.random() * 10));
                 } catch(InterruptedException e) {}
        }

        public void run() {
                sems[3].down();
                sems[4].down();
                sems[5].down();
                a5();
        }
}
```



```
public class TimingRelation {
        public static void main(String[] args) {
                Semaphore[] sems = new Semaphore[6];
                for(int i = 0; i < 6; i++){
                        sems[i] = new Semaphore(0);
                }
                new T1(sems);
                new T2(sems);
                new T3(sems);
                new T4(sems);
                new T5(sems);
        }
}
$ java TimingRelation
a1
a3
a2
a4
a5
$ java TimingRelation
a1
a2
a3
a4
a5
$
```



Classroom exercise

2.3. Additive Semaphores

A semaphore is an integer, where down- and up-operation decrements and increments by one respectively.

Additive semaphores are a generalization of that concept: the down- and up-operation can be done by **arbitrary steps**.

First, we let the generalization be compatible with the semaphore implementation:

```
$ cat AdditiveSemaphore.java
public class AdditiveSemaphore {
        private int value;

        public AdditiveSemaphore(int init) {
                if(init < 0)
                        init = 0;
                value = init;
        }

        public void down() {
                down(1);
        }

        public void up() {
                up(1);
        }
```

Methods down and up have one integer argument, specifying the value to decrease or increase. It has to be positive; otherwise a down operation would increment the semaphore.

```
public synchronized void down(int x) {
    if(x <= 0)
        return;
    while(value - x < 0) {
        try {
            wait();
        }
        catch(InterruptedException e) {}
    }
    value -= x;
}
```

The while loop checks, whether after subtraction of the argument x, the semaphore would become negative; in this case, we will wait.

```java
        public synchronized void up(int x) {
                if(x <= 0)
                        return;
                value += x;
                notifyAll(); //NOT notify
        }

        public void change(int x) {
                if(x > 0)
                        up(x);
                else if(x < 0)
                        down(-x);
        }
} // end of class AdditiveSemaphore
```

Method change can be used to invoke a down or up operation, depending on the sign of its argument.

Analyzing the code shows that an **additive semaphore** performs the decrement and increment operation **indivisible**. That means for example, one `down(n)` operation is **not** the same as n `down(1)` operations:

Let's assume, we have 2 threads T1 and T2 using one additive semaphore. Its actual value is assumed to be 4. Both would like to decrement by 3.

1. **Correct** program fragment:
   Both T1 and T2 invoke **`down(3)`**.
   T1 is chosen and `down(3)` has finished.
   T2 blocks calling `down(3)`. That's what we want!

2. Program fragment of an **incorrect** solution:
   Both T1 and T2 invoke **`down(1); down(1);down(1)`**.
   T1 is chosen and the calls "`down(1)`; `down(1)`" have finished (Semaphore == 2), but the last call of "`down(1)`;" is still open;
   now JVM switches to T2.
   T2 as well calls "`down(1)`; `down(1)`", thus Semaphore == 0. Now the last call of "`down(1)`" let **T2** become **blocked** (wait);
   now JVM switches back to T1
   Now the open call of "`down(1)`" let T1 also become blocked (wait);
   -> DEADLOCK: T1 waits for T2 and simultaneously T2 waits for T1

2.4. Semaphore Groups

Additive semaphores increment and decrement its values in an **atomic manner**. This principle "everything or nothing" is the motivation for semaphore groups.

Note, in Unix this semaphore groups are just called semaphores.

A semaphore group can be seen as an act of generalizing additive semaphores: **one invocation** of the method **change** increments or decrements a **set of semaphores**, belonging to the same group. The **change action** will only be **executed**, if **each group's semaphore** will not become negative; in that case, `change` waits without modifying a semaphore's value.

To realize the class `SemaphoreGroup`, we use an integer array (`values`) to hold the set of semaphores. (Trying to implement `SemaphoreGroup` by a set of `AdditiveSemaphore` objects could result in deadlocks.)

```
$ cat SemaphoreGroup.java
public class SemaphoreGroup {
        private int[] values;  // set of semaphores

        public SemaphoreGroup(int numberOfMembers) {
                if(numberOfMembers <= 0)
                        return;
                values = new int[numberOfMembers];
        }
…
```

The constructor's argument specifies the number of elements in that group.

We do not implement `down`- and `up`-operations, instead we realize **one** method `changeValues`.

```
...
        public synchronized void changeValues(int[] deltas) {
                if(deltas.length != values.length)
                        return;
                while(! canChange(deltas)) {
                        try {
                         wait();
                        } catch(InterruptedException e) {}
                }
                doChange(deltas);
                notifyAll();
        }
...
```

The parameter deltas defines the values for the changes of the semaphores: `deltas[i]` is the value to increase (if positive) or decrease (if negative) semaphore `values[i]`.

Private method `canChange` tests, whether all changes are possible (no `value[i]` may become negative).

```java
	private boolean canChange(int[] deltas) {
		for(int i = 0; i < values.length; i++)
			if(values[i] + deltas[i] < 0)
				return false;
		return true;
	}
```

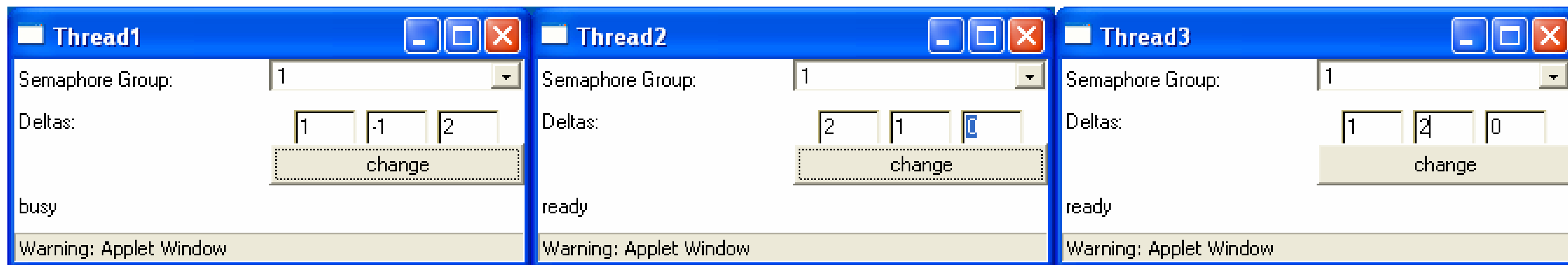
The changes are performed by the private method `doChange` after that all waiting threads are notified.

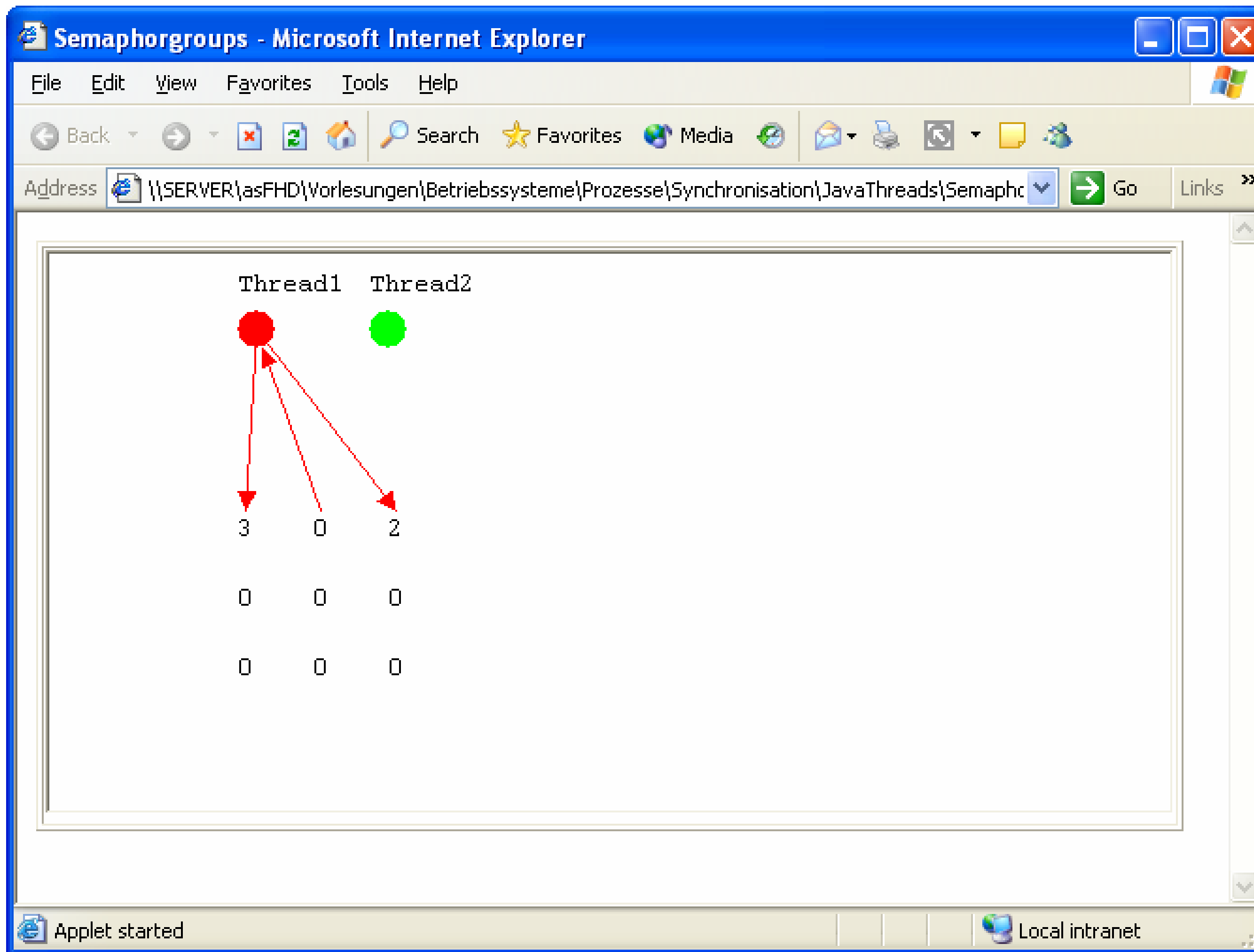
```java
	private void doChange(int[] deltas) {
		for(int i = 0; i < values.length; i++)
			values[i] = values[i] + deltas[i];
	}
```

To complete the class, we have a public method to get the number of elements within a semaphore group.

```java
	public int getNumberOfMembers() {
		return values.length;
	}
}
$
```

To visualize the behavior of semaphore groups, we consider an applet semgrp.html.


Thread1
Semaphore Group:
1
Deltas:
1
-1
2
change
busy
Warning: Applet Window
Thread2
Semaphore Group:
1
Deltas:
2
1
0
change
ready
Warning: Applet Window
Thread3
Semaphore Group:
1
Deltas:
1
2
0
change
ready
Warning: Applet Window


Semaphorgroups - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address \\SERVER\asFHD\Vorlesungen\Betriebssysteme\Prozesse\Synchronisation\JavaThreads\Semaphc
Go
Links
Thread1 Thread2
3 0 2
0 0 0
0 0 0
Applet started
Local intranet

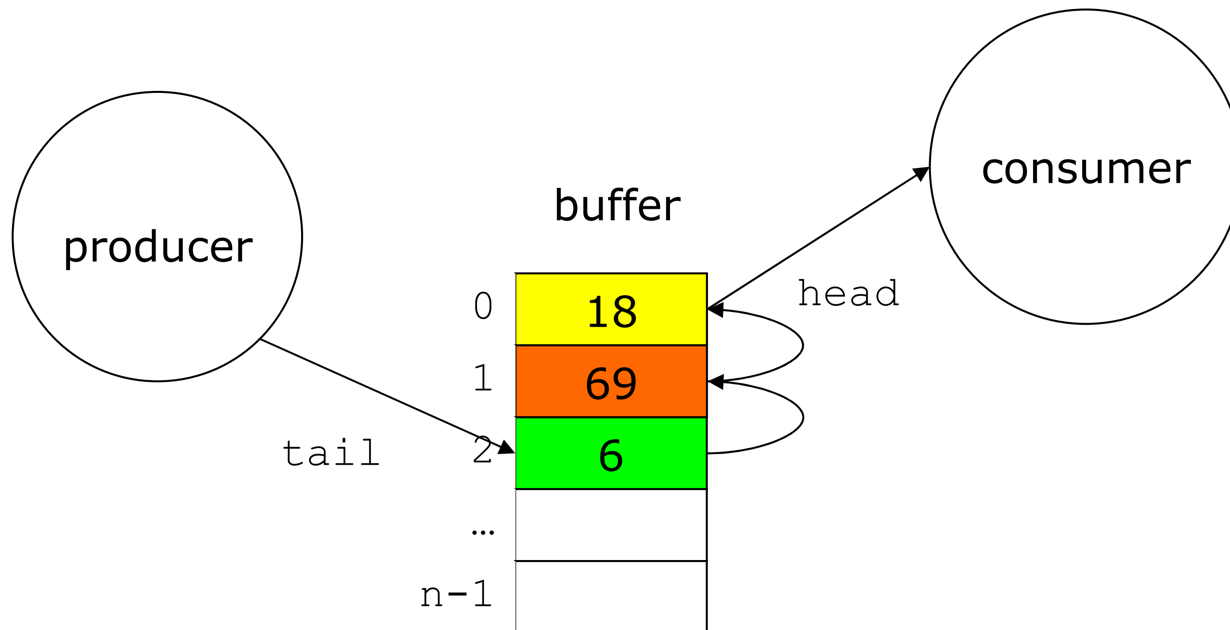
3. Message Queues

Semaphores are mechanisms to synchronize processes. To allow **processes** to communicate, operating systems provide **communication facilities**. This section introduces the concept of message queues, the next section covers pipes. Both concepts are realized within modern operating systems.

First, we discuss a generalization of the class **`buffer`**. This let processes transfer **data** of a **fixed length** (we use integer). After that we show, how data of **arbitrary length** can be transferred (**`MessageQueue`**, **`Pipe`**).

3.1. Buffer of N elements

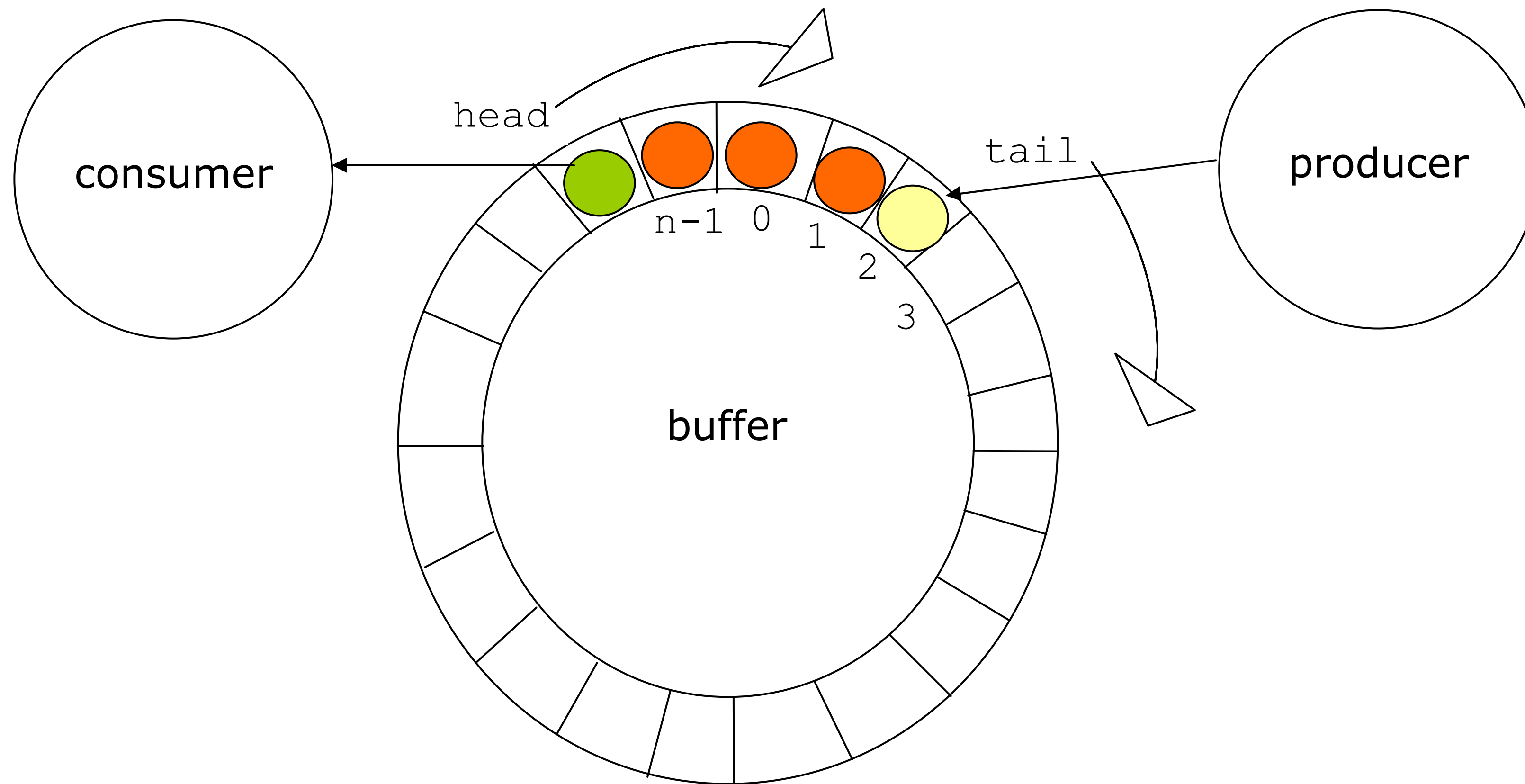
The first idea to organize a buffer of N elements as array is as follows:



The **producer** stores a value into the first empty field (tail).

The **consumer** takes always the value of field number 0 (head) and **reorganizes** the buffer (moving each element to its neighbor)

The **reorganization** of the buffer can be **avoided**, when we use the array in a **cyclic** manner:



The **head** element will be **taken**; a **new** element is stored at the **tail** position. After each operation the corresponding **position pointer** (head and tail) will be **incremented** cyclic.

This mechanism in mind let us come to the following implementation of `BufferN`.

The constructor initializes position pointers and creates the data array.

```
$ cat BufferN.java
public class BufferN {
        private int head;
        private int tail;
        private int numberOfElements;
        private int[] data;

        public BufferN(int n) {
                data = new int[n];
                head = 0;
                tail = 0;
                numberOfElements = 0;
        }
```

When the **buffer** has become **full** the put method is going to **wait**. The usage of a loop is necessary, because we have to use `notifyAll` to awake a task (as seen in the producer consumer example).

If there are some positions **free**, the value is stored at **tail position** and the position counter is incremented cyclic. After having increased the number of elements, we notify waiting threads.

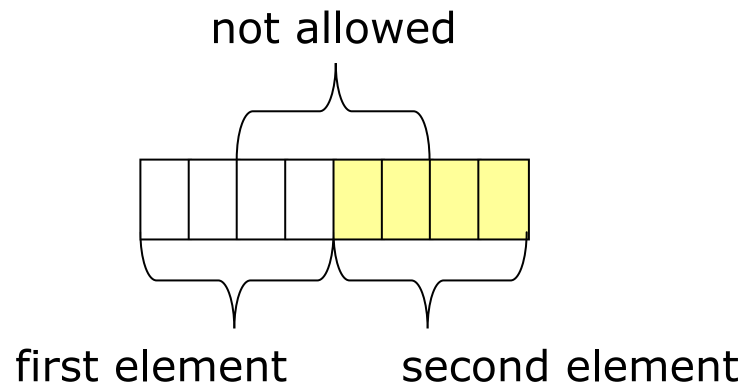
```java
public synchronized void put(int x) {
        while(numberOfElements == data.length) {  // buffer full
                try {
                        wait();
                } catch(InterruptedException e) {}
        }
        data[tail++] = x;
        if(tail == data.length)
                tail = 0;
        numberOfElements++;
        notifyAll();

}
```

Method get can be implemented analogically.

```
public synchronized int get() {
                while(numberOfElements == 0) {
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }
                int result = data[head++];
                if(head == data.length)
                        head = 0;
                numberOfElements--;
                notifyAll();
                return result;

        }

}
$
```

3.2. Message queue implementation

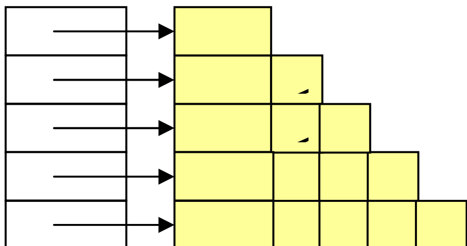
We now consider how to **interchange** data of **arbitrary length**, **preserving data boundaries**. That means, sending a byte array implies that the receiver gets exactly that byte array, not more, not less:



To achieve that, we use a **two dimensional array**. Remember, in Java a two dimensional arrays need not to be a matrix:

```
byte[][] arra2dim = new byte[5];          // array of references to a byte array
for (int i=0; i<array2dim.length; i++)
    array2dim[i] = new byte[2+i];         // the reference will points to an array
```

This code fragment is responsible for an array of the following form:



We copy the message to allow the sender to manipulate the original message after having send the message.

Using the mechanism of `bufferN` we found the solution:

```
$ cat MessageQueue.java
public class MessageQueue {
        private byte[][] msgQueue = null;
        private int qsize = 0;// size of message queue as number of entries
                                // (not number of bytes as in Unix)
        private int head = 0;
        private int tail = 0;

        public MessageQueue(int capacity) {
                if(capacity <= 0)
                        return;
                msgQueue = new byte[capacity][];
        }
```

```java
public synchronized void send(byte[] msg) {
    while(qsize == msgQueue.length) {        // full
        try {
            wait();
        } catch(InterruptedException e) {}
    }

    msgQueue[tail] = new byte[msg.length]; // copy message and store the copy
    for(int i = 0; i < msg.length; i++)
        msgQueue[tail][i] = msg[i];

    qsize++;
    tail++;
    if(tail == msgQueue.length)
        tail = 0;
    notifyAll();
}
```

```
        public synchronized byte[] receive() {
                while(qsize == 0) {
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }

                byte[] result = msgQueue[head];
                msgQueue[head] = null;
                qsize--;
                head++;
                if(head == msgQueue.length)
                        head = 0;
                notifyAll();
                return result;
        }
}
$
```

An applet demonstrates the behavior of class MessageQueue.

4. Pipes

Message queues preserve message boundaries. This kind of communication is named "**message oriented**".

Now we consider so called "**(data) stream oriented**" communication. That means, a **receiver** of a message can **not identify** the portions the message has been composed off.

An applet demonstrates the behavior of class Pipe.

A pipe has a defined size (we use a byte array of fixed size).

```
$ cat Pipe.java
public class Pipe {
        private byte[] buffer = null;
        private int bsize = 0;
        private int head = 0;
        private int tail = 0;

        public Pipe(int capacity) {
                if(capacity <= 0)
                        return;
                buffer = new byte[capacity];
        }
```

Put data into a pipe has to be realized as an **atomic operation**: if the message size is larger than available free space within the pipe, the sending thread must block until space will become available.

```
public synchronized void send(byte[] msg)         {
        if(msg.length <= buffer.length) {
                // sent as atomic operation
                while(msg.length > buffer.length - bsize) {
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }

                // copy message into buffer
                for(int i = 0; i < msg.length; i++) {
                        buffer[tail] = msg[i];
                        tail++;
                        if(tail == buffer.length)
                                tail = 0;
                }
                bsize += msg.length;
                notifyAll();
        }
```

If the **message length** is larger than the pipe's size, the sending **thread would hang**. Therefore, we implement the send operation to be able to **split a message** into **small portions**.

```
            else {
                  // send in portions
                  int offset = 0;
                  int stillToSend = msg.length;
                  while(stillToSend > 0) {
                        while(bsize == buffer.length) {
                              try {
                                    wait();
                              } catch(InterruptedException e) {}
                        }
                        int sendNow = buffer.length - bsize;
                        if(stillToSend < sendNow)
                              sendNow = stillToSend;
                        for(int i = 0; i < sendNow; i++) {
                              buffer[tail] = msg[offset];
                              tail++;
                              if(tail == buffer.length)
                                    tail = 0;
                              offset++;
                        }
                        bsize += sendNow;
                        stillToSend -= sendNow;
                        notifyAll();
                  }
            }
      }
```

Receiving a message has to **block**, if **no data** is available. The parameter of the receive method defines the **expected number of bytes** to receive. If **less data** is available than expected, the operation will **not** be **blocked**. In this case only the bytes available are received.

```
        public synchronized byte[] receive(int noBytes) {
                while(bsize == 0) {
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }
                if(noBytes > bsize)
                        noBytes = bsize;
                byte[] result = new byte[noBytes];
                for(int i = 0; i < noBytes; i++) {
                        result[i] = buffer[head];
                        head++;
                        if(head == buffer.length)
                                head = 0;
                }
                bsize -= noBytes;
                notifyAll();
                return result;
        }
}
$
```

5. Dining philosophers

A classical IPC problem is the “dining philosopher problem”. Since Dijkstra posed and solved this problem in 1965, everyone tries to show how wonderful his synchronization primitive can solve this problem. Thus, we have to show, how we can do it with Java build in primitives.

The problem can be stated as follows:

> **Five philosophers** are seated around a circular table. Each philosopher has a **plate** of Italian **spaghetti**. Italian spaghetti is so slippery that a philosopher needs **two forks** to eat it. Between each plate is a fork as shown next:

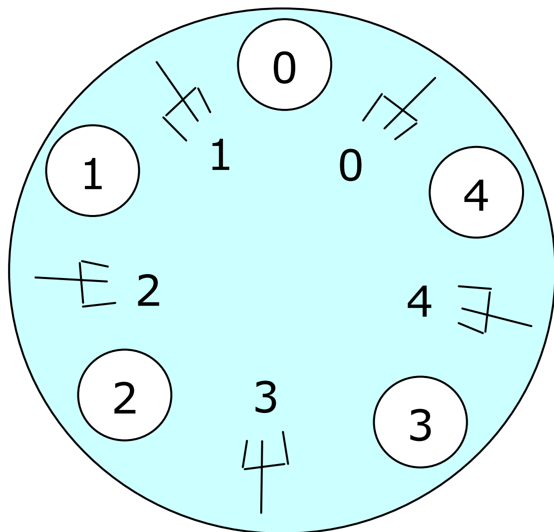


Plate i has
- on its left fork i and
- on its right fork i+1(cyclic add)

Philosophers i uses plate i

> The **life** of a philosopher consists of alternate periods of **eating** and **thinking**. When a philosopher gets hungry, he tries to acquire his left and right fork, one at a time, in either order. If successful in acquiring two forks, he eats for a while, then puts down the fork and continues to think.

The **key question** is: can you write a program for each philosopher that does what is supposed to do and never gets stuck (because of a deadlock situation)?

The obvious, but **wrong** solution (in C++) is:

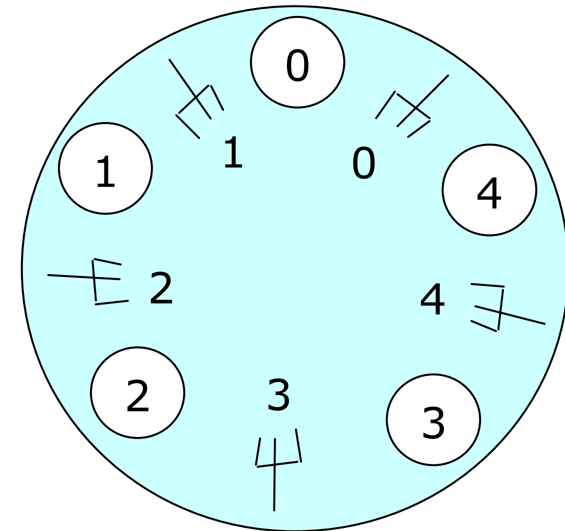
```
const int N=5;
philosophers(int i) {
    while (true) {
        think();
        takeFork(i);           // take left fork
        take_fork((i+1)%N);    // take right fork
        eat();
        putFork(i);            // put left fork back on the table
        putFork((i+1)%N);      // put right fork back on the table
    }
}
```

Function `takeFork` waits until the specified fork is available and then seizes it. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the process.

This solution fails if **all philosophers take simultaneously the left fork**. None will now be able to take their right fork, and there will be a deadlock – they all will die of hunger.

With the mechanisms of the course in mind, we know that we have to implement the process of taking a fork as **atomic** operation and find the following solution:

```
$ cat Philosophers.java
class Table {
        private boolean[] usedFork;

        public Table(int numberForks) {
                usedFork = new boolean[numberForks];
                for(int i = 0; i < usedFork.length; i++)
                        usedFork[i] = false;
        }

        private int left(int i) { return i; }

        private int right(int i) {
                if(i+1 < usedFork.length)
                        return i+1;
                else
                        return 0;
        }

        public synchronized void takeForks(int place) {
                while(usedFork[left(place)]|| usedFork[right(place)]) {
                        try {
                                wait();
                        } catch(InterruptedException e) {}
                }
                usedFork[left(place)] = true;
                usedFork[right(place)] = true;
        }

        public synchronized void putBackForks(int place)
        {
                usedFork[left(place)] = false;
                usedFork[right(place)] = false;
                notifyAll();
        }
} // table
```



```java
class Philosoph extends Thread {
        private Table Table;
        private int place;

        public Philosoph(Table Table, int place) {
                this.Table = Table;
                this.place = place;
                start();
        }

        public void run() {    // life of a philosopher
                while(true) {
                        thinking(place);
                        Table.takeForks(place);
                        eating(place);
                        Table.putBackForks(place);
                }
        }

        private void thinking(int place) {
                System.out.println("Philosoph " + place + " thinking.");
                try {
                        sleep((int)(Math.random() * 20000));
                } catch(InterruptedException e) {}
        }

        private void eating(int place) {
                System.out.println("Philosoph " + place + " starts eating.");
                try {
                        sleep((int)(Math.random() * 20000));
                } catch(InterruptedException e) {}
                System.out.println("Philosoph " + place + " finished eating.");
        }
}
```

```java
public class Philosophers {
	private static final int numberPhilisophers = 5;

	public static void main(String[] args) {
		Table Table = new Table(numberForks);
		for(int i = 0; i < numberPhilisophers; i++)
			new Philosoph(Table, i);
	}
}
$
```

```
$ java Philosophers
Philosoph 0 thinking.
Philosoph 1 thinking.
Philosoph 2 thinking.
Philosoph 3 thinking.
Philosoph 4 thinking.
Philosoph 3 starts eating.
Philosoph 3 finished eating.
Philosoph 3 thinking.
Philosoph 2 starts eating.
Philosoph 0 starts eating.
Philosoph 0 finished eating.
Philosoph 0 thinking.
Philosoph 4 starts eating.
Philosoph 2 finished eating.
Philosoph 2 thinking.
Philosoph 1 starts eating.
Philosoph 1 finished eating.
Philosoph 1 thinking.
```

An other solution is to use the concept of semaphore groups. We show it without any explanations.

```
$ cat PhilosophersSemGroup.java
class PhilosopherSemGroup extends Thread {
        private SemaphoreGroup sems;
        private int place;
        private int leftFork;
        private int rightFork;

        public PhilosopherSemGroup(SemaphoreGroup sems, int place) {
                this.sems = sems;
                this.place = place;
                leftFork = place;
                if(place+1 < sems.getNumberOfMembers())
                        rightFork = place+1;
                else
                        rightFork = 0;
                start();
        }

        public void run() {
                int[] deltas = new int[sems.getNumberOfMembers()];
                for(int i = 0; i < deltas.length; i++)
                        deltas[i] = 0;

                while(true) {
                        thinking(place);
                        deltas[leftFork] = -1;
                        deltas[rightFork] = -1;
                        sems.changeValues(deltas);
                        eating(place);
                        deltas[leftFork] = 1;
                        deltas[rightFork] = 1;
                        sems.changeValues(deltas);
                }
        }
```

```java
        private void thinking(int place) {
                System.out.println("Philosopher " + place
                                   + " is thinking.");
                try {
                        sleep((int)(Math.random() * 20000));
                } catch(InterruptedException e) {}
        }

        private void eating(int place) {
                System.out.println("Philosopher " + place
                                   + " starts eating.");
                try {
                        sleep((int)(Math.random() * 20000));
                } catch(InterruptedException e) {}
                System.out.println("Philosopher " + place
                                   + " finished eating.");
        }
}

public class PhilosophersSemGroup {
        private static final int N = 5;

        public static void main(String[] args) {
                SemaphoreGroup group = new SemaphoreGroup(N);
                int[] init = new int[N];
                for (int i = 0; i < init.length; i++)
                        init[i] = 1;
                group.changeValues(init);

                for(int i = 0; i < N; i++)
                        new PhilosopherSemGroup(group, i);
        }
}
```

The program can be animated by an applet.

Classroom exercise