Sistemas Distribuídos

Concurrency 1

António Rui Borges

Summary

- *Program vs. Process*
  - *Characterization of a multiprogrammed environment*
- *Processes vs. Threads*
  - *Characterization of a multithreaded environment*
- *Execution environment*
- *Threads in Java*
- *Suggested readings*

Program vs. Process

Generally speaking, a *program* can be defined as a sequence of instructions which describes the execution of a certain task in a computer. However, for this task to be *in fact* carried out, the corresponding program must be executed.

A program execution is called a *process*.

Representing an activity that is taking place, a *process* is characterized by

- the *addressing space* – the code and the current value of all its associated variables
- the *processor context* – the current value of all the processor internal registers
- the *I/O context* – all the data that are being transferred to the input and from the output devices
- the *state* of the execution.

Modeling the processes - 1

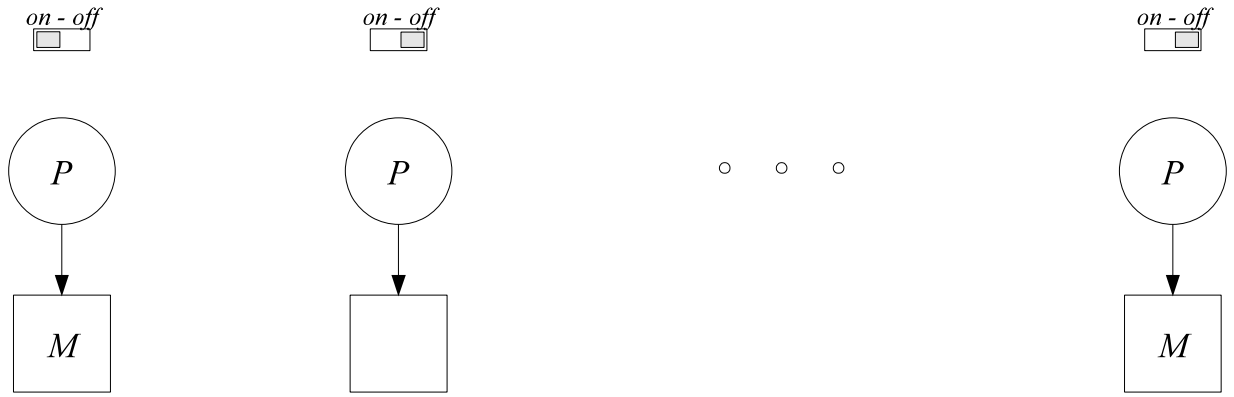
Multiprogramming, by creating an image of apparent simultaneity in the execution of different programs by the same processor, makes very difficult the perception of the activities that are taking place at the same time.

This image can be simplified if, instead of trying to follow the execution path undergone by the processor in its continuous meandering among processes, one supposes there are a set of virtual processors, one per process which concurrently coexists, and that the processes are run in parallel through the activation (*on*) and the deactivation (*off*) of the associated processors.

One further assumes in this model that

- process execution is not affected by the instant and the code location where the commutation takes place
- no restrictions are imposed to the total, or partial, execution time.

Modeling the processes - 2



- the *commutation* of the *process context* is simulated by the activation and the deactivation of the virtual processors and is controlled by its *state*
- in a *monoprocessor*, the number of active virtual processors at any given instant is one, at the maximum
- in a *multicore processor*, the number of active virtual processors at any given instant is equal to the number of processors in the core, at the maximum.

Process state diagram - 1

A process may be in different situations, called *states*, along its existence.

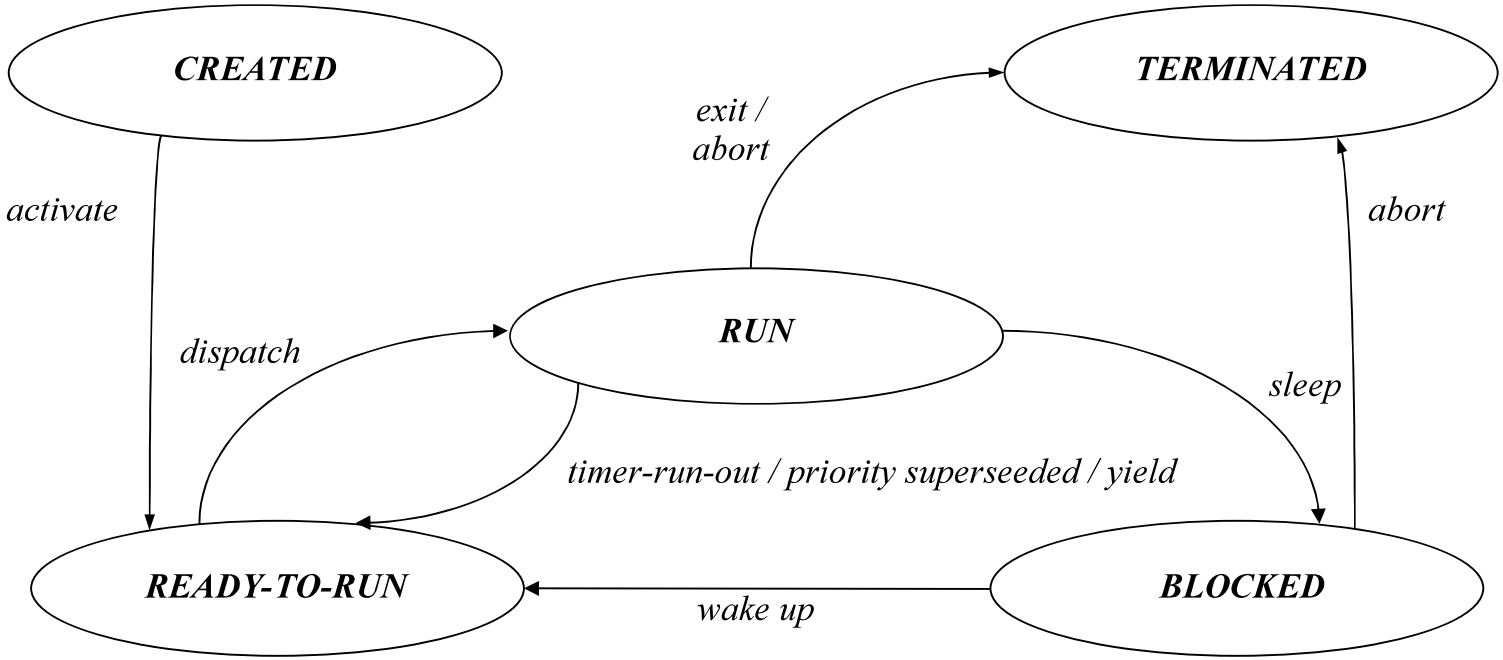
The most important states are the following

- *run* – when it holds the processor and is, therefore, in execution
- *ready-to-run* – when it waits for the assignment of the processor to start or resume execution
- *blocked* – when it is prevented to proceed until an external event occurs (access to a resource, completion of an input/output operation, etc).

State transitions are usually triggered by an external source, the operating system, but may be triggered by the process itself in some instances.

The part of the operating system which deals with [process] state transitions is called the *scheduler* (*processor scheduler*, in this case), and forms an integral portion of its nucleus, the *kernel*, which is responsible for exception handling and for scheduling the assignment of the processor and all other system resources to the processes.

Process state diagram - 2



Process state diagram - 3

activate – a process is created and placed in the *ready-to-run queue* waiting to be scheduled for execution

dispatch – one of the processes of the *ready-to-run queue* is selected by the scheduler for execution

timer-run-out – the process in execution exhausted the slot of processor time which was assigned to it (*preemptive scheduling*)

priority superseded – the process in execution looses the processor because the *ready-to-run queue* now contains a process of higher priority that requires the processor (*preemptive scheduling*)

yield – the process releases voluntarily the processor to allow other processes to be executed (*non-preemptive scheduling*)

sleep – the process is prevented to proceed and must wait for an external event to occur

wake up – the external event the process was waiting for has occurred

exit / abort – the process has terminated / is forced to terminate its execution and waits for the resources that were assigned to it to be released

Processes vs. Threads - 1

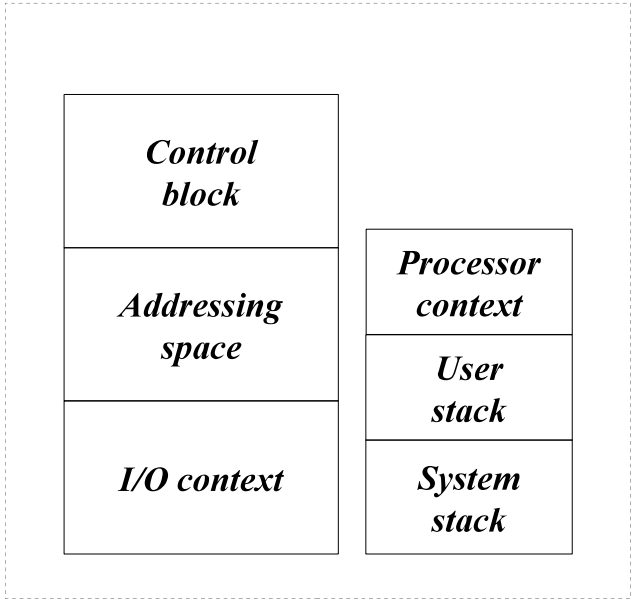
The concept of *process* embodies the following properties

- *resource ownership* – a private addressing space and a private set of communication channels with the input and output devices
- *thread of execution* – a *program counter* which points to the instruction that must be executed next, a set of *internal registers* which contain the current values of the variables being processed and a *stack* which keeps the history of execution (a *frame* for each routine that was called and has not yet returned).

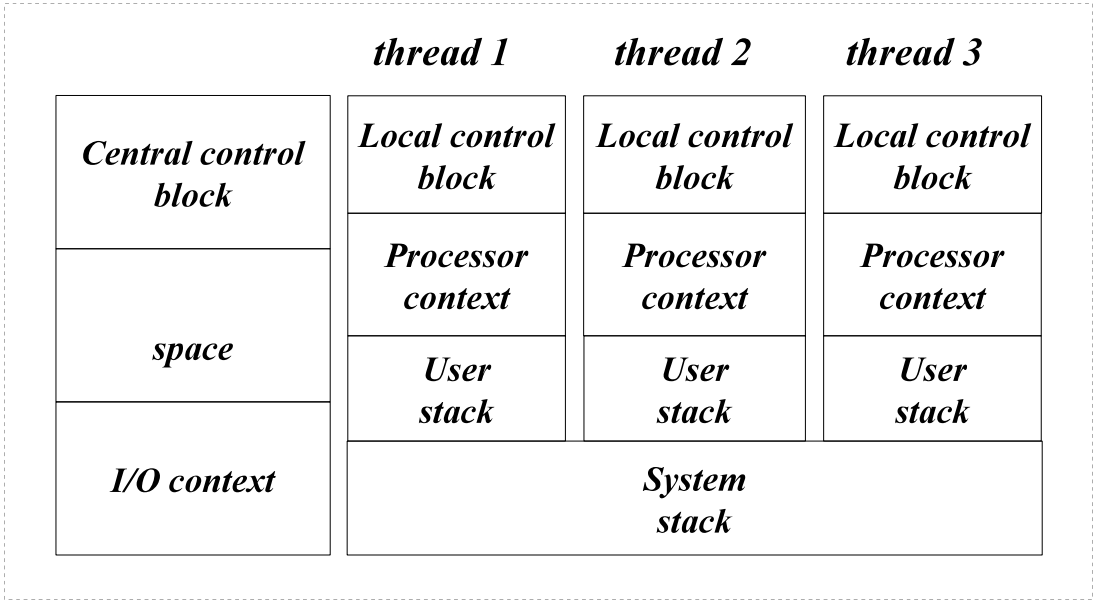
These properties, although taken together in a *process*, can be treated separately by the execution environment. When this happens, *processes* are envisaged as grouping a set of resources and *threads*, also known as *light weight processes*, represent runnable independent entities within the context of a single process.

Multithreading, then, means an environment where it is possible to create multiple *threads of execution* within the same process.

Processes vs. Threads - 2



Single threading

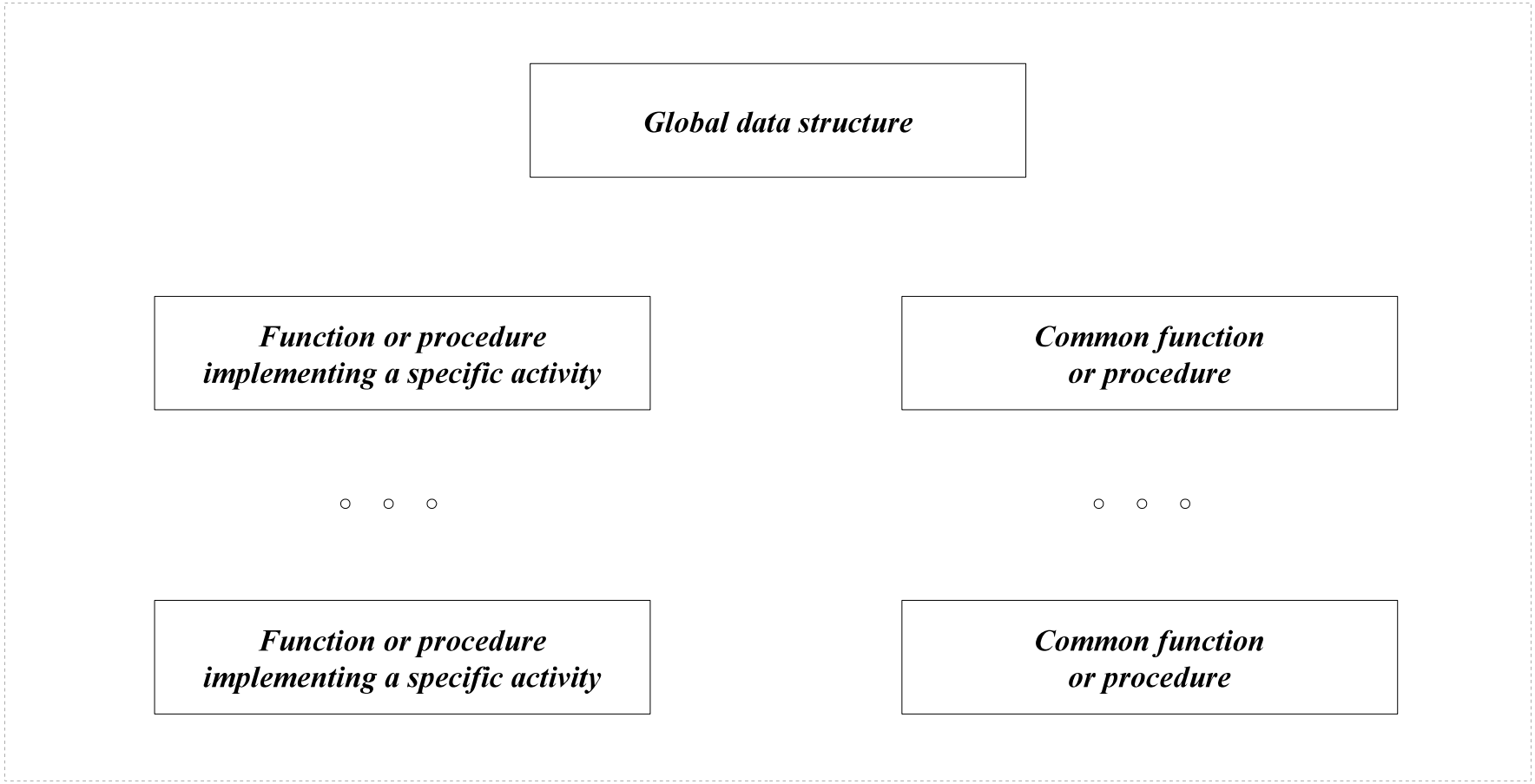


Multithreading

Advantages of a multithreaded environment

- *greater simplicity in solution decomposition and greater modularity in its implementation* – programs which involve multiple activities and service multiple requests are easier to design and implement in a concurrent perspective than in a pure sequential one
- *better management of computer system resources* – sharing the addressing space and the I/O context among the *threads* of an application results in decreasing the complexity of managing main memory occupation and access to the input / output devices
- *greater efficiency and speed of execution* – a solution decomposition based on *threads*, by opposition to one based on processes, requires less resources of the operating system, enabling that operations like process creation and termination and a context commutation to become less heavy and, thus, more efficient; furthermore, it becomes possible in symmetric multiprocessing to schedule for parallel execution multiple *threads* belonging to the same application, thus increasing the speed of execution

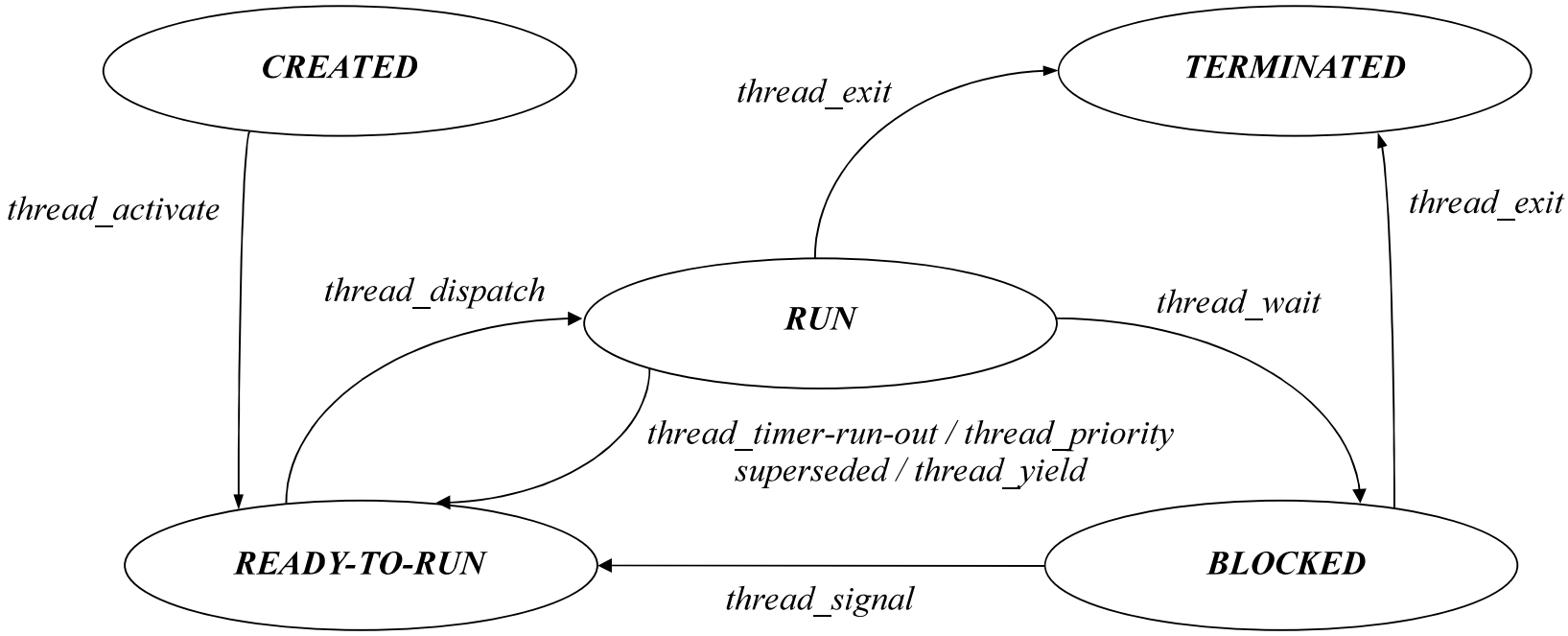
Organization of a multithreaded program - 1



Organization of a multithreaded program - 2

- each *thread* is typically associated with the execution of a *function or procedure implementing a specific activity*
- the *global data* structure forms an information sharing space, defined in terms of variables and communication channels with the input/output devices, to be accessed by the multiple *threads* that coexist at a given time for writing and for reading
- the *main program*, represented in the diagram by a *function or procedure implementing a specific activity*, constitutes the first *thread* to be created and, typically, the last *thread* to be concluded

Thread state diagram



Support to the implementation of a multithreaded environment

- *user level threads – threads* are implemented by a specific library at user level which brings support to the creation, management and scheduling of *threads* *without* kernel interference; this generates a very versatile and portable, but inefficient, implementation since, as the kernel perceives only processes, when a particular *thread* invokes a blocking *system call*, all the process is blocked, even if there were *threads* ready to be run
- *kernel level threads – threads* are implemented at kernel level by directly providing the operations for the creation, management and scheduling of *threads*; the implementation is operating system specific, but the blocking of a particular *thread* does not affect the dispatching of the remaining for execution and parallel execution in a multicore processor becomes possible

Execution environment - 1

Java virtual machine (JVM) constitutes the execution environment for a Java coded application. In principle, JVM runs on the top of the operating system of the hardware platform executing a Java program and establishes with it a very intimate connection. Access to information concerning the execution environment can be obtained through the invocation of methods on two reference data types of the Java base library, `java.lan :` and .

Among the information that is provided, the following should be noticed

- number of processors and memory size available to run the code
- references to the streams associated with standard input, standard output and standard error devices
- access to a modifiable set of definitions, called *properties*, which characterize the execution environment
- read-only access to the definitions of the variables of the operating system user interface, the *shell*, where the java command was executed – called in this context *environment*.

environment - 1

```
[ruib@ruib-laptop environment]$ java CollectEnvironmentData

Characterization of Java Virtual Machine (JVM)
N. of available processors = 8
Size of dynamic memory presently free (in bytes) = 248250352
Size of total dynamic memory (in bytes) = 249561088
Maximum size of available main memory of the hardware platform where Java
virtual machine is installed (in bytes) = 367840460

Properties of the execution environment
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = /opt/jdk1.8.0_241/jre/lib/amd64
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
path.separator = :
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
user.dir =
     /home/ruib/Teaching/SD/2020_2021/aulas teóricas/exemplos demonstrativos/
         threadBasics/environment
java.runtime.version = 1.8.0_241-b07
java.io.tmpdir = /tmp
os.name = Linux
sun.jnu.encoding = UTF-8
os.version = 5.10.22-100.fc32.x86_64
user.home = /home/ruib
                                        .    .    .
```

environment - 2

Variables of the execution environment

```
PATH   =   /opt/jdk1.8.0_241/bin:/opt/jdk1.8.0_241/jre/bin:/opt/mpich/bin:/home/
ruib/.local/bin:/home/ruib/bin:/usr/lib64/ccache:/usr/local/bin:/usr/bin:/
bin:/usr/local/sbin:/usr/sbin
LC_MEASUREMENT = pt_PT.UTF-8
LC_COLLATE = pt_PT.UTF-8
LOGNAME = ruib
PWD       =       /home/ruib/Teaching/SD/2020_2021/aulas       teóricas/exemplos
demonstrativos/threadBasics/environment
c=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;36:*.oga=00;36:*.opus=00;36:*.spx=00;36
:*.xspf=00;36:
XDG_SESSION_DESKTOP = KDE
SHLVL = 1
LC_MONETARY = pt_PT.UTF-8
DISPLAY = :0
LC_NUMERIC = pt_PT.UTF-8
HOME = /home/ruib
                                        .    .    .
```

Execution environment - 2

The execution environment also allows running commands directly on the underlying operating system user interface. Two reference data types of the Java base library, `java.lang`, are instrumental to fulfill this purpose: `ProcessBuilder` and `Process`.

Each `ProcessBuild` object manages process attributes such as the *shell* command to be executed and the setting of the working directory and of the streams associated with standard input, standard output and standard error devices. Multiple processes can be created in succession from the same object, thus sharing the same attribute configuration.

Each process that is created is an instance of the `Process` data type. Means are provided here to check if it is still running, wait for its termination, get its termination status, kill it and obtain references to the streams associated with its standard input, standard output and standard error devices.

runCommand

```
[ruib@ruib-laptop runCommand]$ java ListWorkDir
Listing current working directory

----------------------------------------
total 16
drwxrwxr-x. 2 ruib ruib 4096 Mar 17 12:09 .
drwxrwxr-x. 8 ruib ruib 4096 Feb 26  2018 ..
-rw-rw-r--. 1 ruib ruib 1298 Mar 17 12:09 ListWorkDir.class
-rw-rw-r--. 1 ruib ruib 1626 Feb 28  2017 ListWorkDir.java
----------------------------------------
exit status = 0
```

Threads in Java - 1

Being Java a concurrent programming language, *threads* are supported by the language itself. Conceptually, the creation of a *thread* presupposes two entities to exist in the Java virtual machine: an object representing an autonomous thread of execution, the *thread* itself, and a non-instantiated reference data type, or an object instantiated from it, which defines the method executed by the *thread*, its life cycle.

Java virtual machine manages the *multithreaded* environment according to the following rules

- every running program consists of at least one *thread* which is implicitly created when the virtual machine, after initializing the execution environment, calls the method `main` on the initial data type
- the remaining *threads* are explicitly created by the *thread* `main`, or by any *thread* created in succession from the *thread* `main`
- the program terminates when all the created *threads* have finished executing their associated method.

Threads in Java - 2

Java base library, `java.lang`, provides two reference data types, one using the constructor *interface* and the other using the constructor *class*, which are instrumental in building a *multithreaded* environment.

```
public interface Runnable
{
  public void run ();
}


public class Thread
{
    . . .
  public void run ()
  public void start ()
    . . .
}
```

Threads in Java - 3

Each autonomous thread of execution is an instantiation of the reference data type `Thread`. It defines two methods which are operationally relevant in this context

- `run` – which is called when the *thread* is put into execution (started) and which represents its life cycle
- `start` – which is called to *start* the *thread*.

It is not, however, strictly necessary to create new reference data types, derived from `Thread` and which *override* the method `run`, to ensure the execution of specific tasks. The same goal may alternatively be achieved by creating an independent reference data type which implements the interface `Runnable` and which, as a consequence, defines `run`.

The latter is usually reported as the preferred approach in Java related literature, but the former enables a Java solution to multiple inheritance which is quite useful in building servers that have a client service differentiation.

Threads in Java - 4

Thread creation (*approach 1*)

instantiation

```
. . .
MyThread thr = new MyThread ();

thr.start ();
. . .
```

putting into execution

overriding of the method `run` *which establishes the thread operativeness*

```
public class MyThread extends Thread
{
            . . .
  public void run ()
  {
       . . .
  }
}
```

reference data type which defines the thread functionality

Threads in Java - 5

Thread creation (*approach 2*)

instantiation

```
. . .
Thread thr = new Thread (new MyThread ());

thr.start ();
. . .
```

putting into execution

implementation of the method `run` *which establishes the thread operativeness*

```
public class MyThread implements Runnable
{
          . . .
  public void run ()
  {
       . . .
  }
}
```

reference data type which defines the thread functionality

Threads in Java - 6

A *thread* has the following attributes

- *name* – assigned name (by default, the execution environment generates a *string* of format `Thread-#`, where `#` is the creation number successively incremented from zero)
- *internal identifier* – number of type `long` which unique and is kept unchanged during the *thread* lifetime
- *group* – group the *thread* belongs to (all the *threads* of the same application belong by default to the same group, the group)
- *priority* – it may vary from 1 (*`MIN_PRIORITY`*) to 10 (*`MAX_PRIORITY`*), by default, the execution environment assigns the value of 5 (*`NORM_PRIORITY`*)
- *state* – *thread* current state
  - `NEW` (*CREATED*), after instantiation of a reference variable of data type `Thread`, or of a derived data type
  - `RUNNABLE` (*READY-TO-RUN* or *RUN*), when waits for execution or is in execution
  - `BLOCKED`, `WAITING` or `TIME_WAITING` (*BLOCKED*), when is blocked
  - `TERMINATED` (*TERMINATED*), after termination.

threadInfo

```
[ruib@ruib-laptop threadInfo]$ java CollectThreadData

Thread characterization

Name = main
Internal identifier = 1
Group = main
Priority = 5
Current state = RUNNABLE

Possible states:
   NEW
   RUNNABLE
   BLOCKED
   WAITING
   TIMED_WAITING
   TERMINATED
```

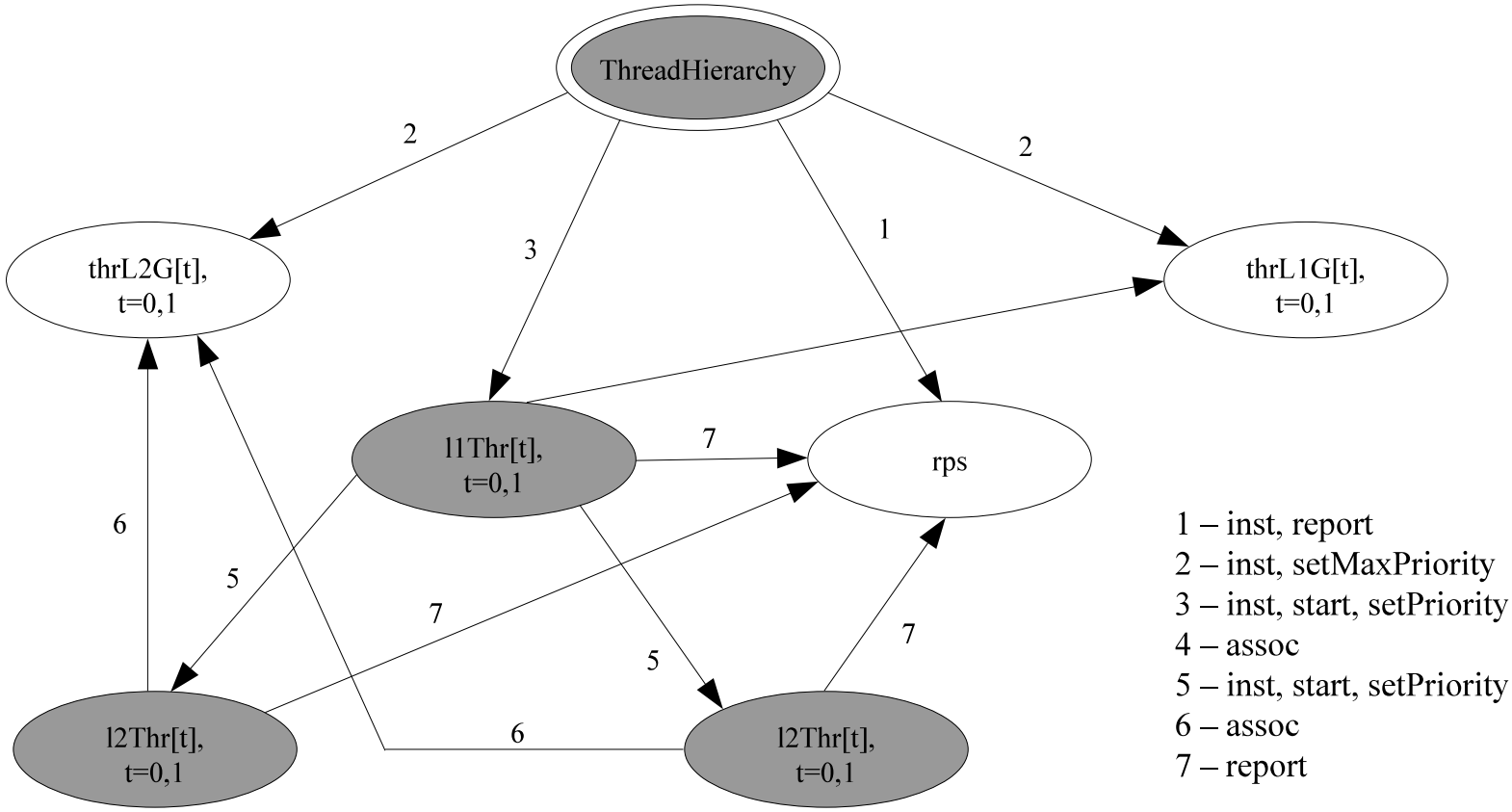
Threads in Java - 7

The partition in different groups of the *threads* which are instantiated in succession, makes possible to organize in a hierarchical and functional fashion an application, as well as taking advantage of the Java specificities in the definition of common properties and in the interaction with them. Java base library, `java.lang`, provides the reference data type `Thre` to fulfill this purpose.

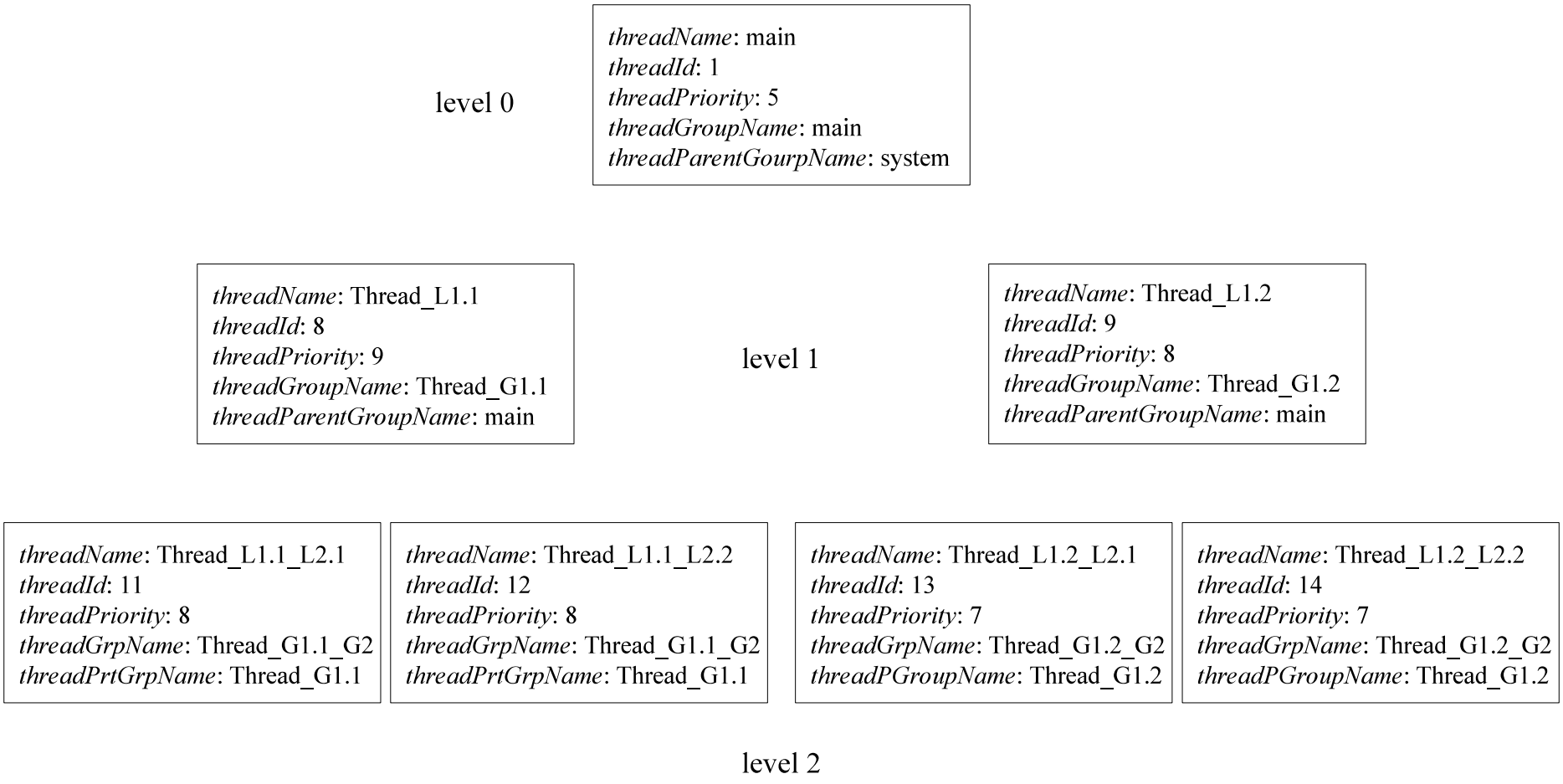
Thus, it turns out that it is possible

- to define at the very beginning the maximum priority and the property of being a *daemon* associated with a given group – all the *threads* later instantiated, and belonging to this group, will keep these properties
- to send an interrupt to *all* the *threads* belonging to the same group in an unique operation, instead of doing it separately to each member of the group.

hierarchy - 1


ThreadHierarchy
thrL2G[t], t=0,1
thrL1G[t], t=0,1
l1Thr[t], t=0,1
rps
l2Thr[t], t=0,1
l2Thr[t], t=0,1
2
2
3
1
7
6
5
7
5
7
6
1 – inst, report
2 – inst, setMaxPriority
3 – inst, start, setPriority
4 – assoc
5 – inst, start, setPriority
6 – assoc
7 – report

hierarchy - 2


level 0
threadName: main
threadId: 1
threadPriority: 5
threadGroupName: main
threadParentGourpName: system
threadName: Thread_L1.1
threadId: 8
threadPriority: 9
threadGroupName: Thread_G1.1
threadParentGroupName: main
level 1
threadName: Thread_L1.2
threadId: 9
threadPriority: 8
threadGroupName: Thread_G1.2
threadParentGroupName: main
threadName: Thread_L1.1_L2.1
threadId: 11
threadPriority: 8
threadGrpName: Thread_G1.1_G2
threadPrtGrpName: Thread_G1.1
threadName: Thread_L1.1_L2.2
threadId: 12
threadPriority: 8
threadGrpName: Thread_G1.1_G2
threadPrtGrpName: Thread_G1.1
threadName: Thread_L1.2_L2.1
threadId: 13
threadPriority: 7
threadGrpName: Thread_G1.2_G2
threadPGroupName: Thread_G1.2
threadName: Thread_L1.2_L2.2
threadId: 14
threadPriority: 7
threadGrpName: Thread_G1.2_G2
threadPGroupName: Thread_G1.2
level 2

hierarchy - 3

Printed values

```
[ruib@ruib-laptop hierarchy_1]$ java ThreadHierarchy1

Number of level 1 threads? 2
Number of level 2 threads per level 1 threads? 2

Thread name: main
Thread id: 1
Thread priority: 5
Name of the thread group: main
Name of the parent group of the thread group: system
N. of active threads in the thread group: 1
Name of active threads in the thread group: main
N. of active subgroups in the thread group: 0

Thread name: Thread_L1.1_L2.1
Thread id: 11
Thread priority: 8
Name of the thread group: Thread_G1.1_G2
Name of the parent group of the thread group: Thread_G1.1
N. of active threads in the thread group: 2
Name of active threads in the thread group: Thread_L1.1_L2.1 -
N. of active subgroups in the thread group: 0
                                    .   .   .
```

Threads in Java - 8

Java virtual machine supposes a policy of *non-preemptive scheduling* based on a system of static priorities with 10 levels

- the transitions between the state *RUN* and the state *READY-TO-RUN* are of type *thread_priority_superseded* and *thread_yield*
- the priority assigned to a *thread*, defined upon instantiation or modified before its creation, remains unaltered during its active life time.

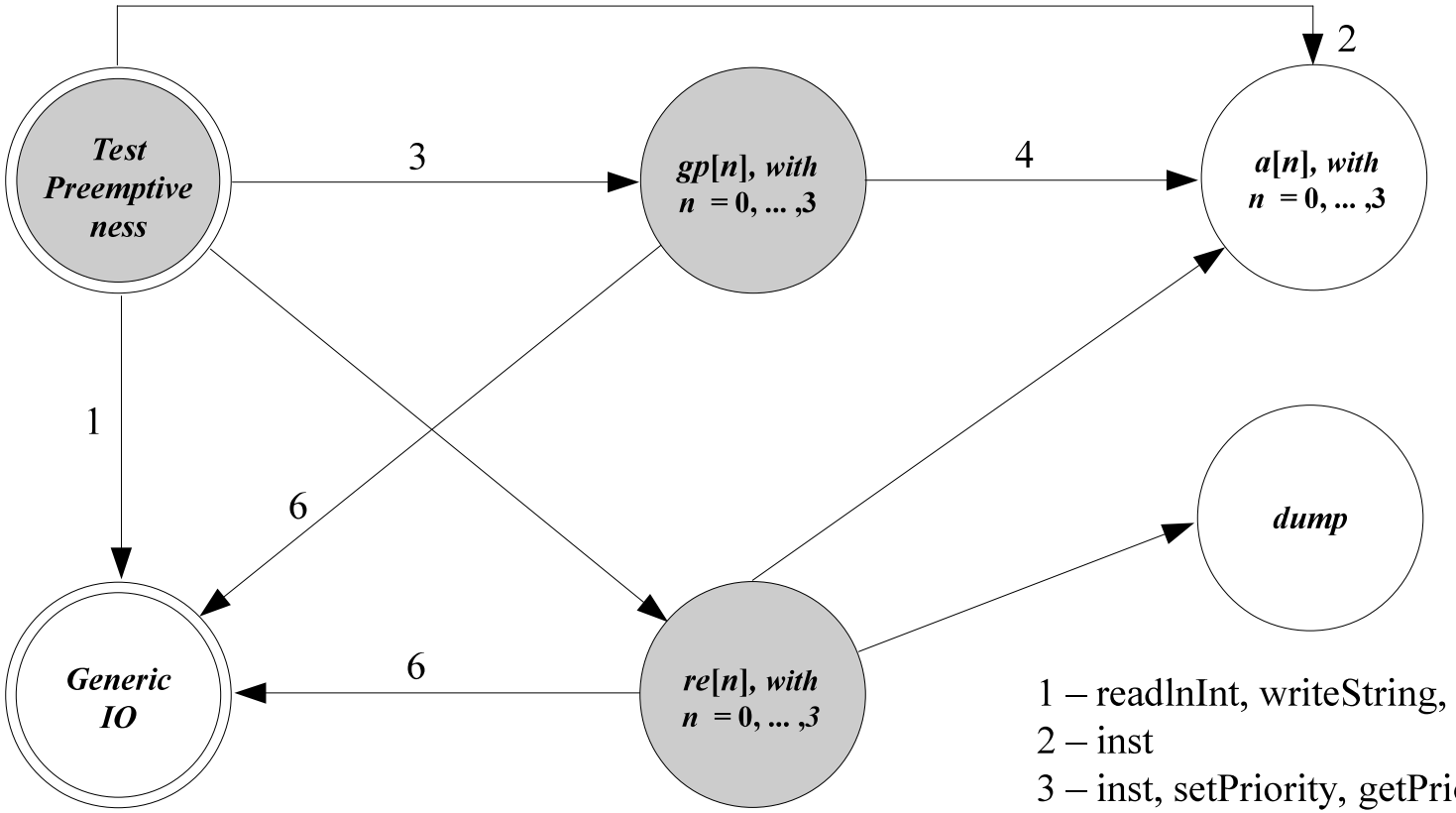
Java virtual machine, however, does not enforce strictly the *scheduling* policy. The implementation has a lot of freedom on how it puts it to work. This is particularly true when Java virtual machine is run on the top of a general purpose multitasking operating system!

In Linux, for instance, Java threads are *kernel* level threads, taking advantage of *multicore* processors to enhance parallel execution, and the prevailing *scheduling* policy is the local one.

testPreemptiveness - 1

- there are four sets, consisting each of a computation intensive thread, an I/O intensive thread and a shared access variable
- the computation intensive thread successively increments by one the shared access variable 10 million times
- the I/O intensive thread successively reads and prints the shared access variable until its value reaches 10 million
- thread priorities can be changed and *yield* may be introduced after each increment operation of the computation intensive thread is performed
- the number of reads and prints carried out by the I/O intensive thread is used as a figure of merit to estimate the relative execution speed of both threads

testPreemptiveness - 2



1 – readlnInt, writeString, writelnString
2 – inst
3 – inst, setPriority, getPriority, start
4 – write / read variable
5 – read variable
6 – writelnString
7 – inst, write, flush

testPreemptiveness - 3

Without thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 1
Priority level of I/O intensive threads? 10
Priority of computation intensive threads = 1
Priority of I/O intensive threads = 10
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of A = 59702
A = 10000000
N. of iterations in printing values of D = 123791
D = 10000000
N. of iterations in printing values of C = 111804
C = 10000000
N. of iterations in printing values of B = 102411
B = 10000000
```

testPreemptiveness - 4

Without thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 10
Priority level of I/O intensive threads? 1
Priority of computation intensive threads = 10
Priority of I/O intensive threads = 1
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of C = 62180
C = 10000000
B = 10000000
N. of iterations in printing values of B = 103934
N. of iterations in printing values of D = 91698
D = 10000000
A = 10000000
N. of iterations in printing values of A = 119485
```

testPreemptiveness - 5

With thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 1
Priority level of I/O intensive threads? 10
Priority of computation intensive threads = 1
Priority of I/O intensive threads = 10
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of C = 4834998
C = 10000000
N. of iterations in printing values of D = 5085208
D = 10000000
N. of iterations in printing values of A = 4931095
A = 10000000
N. of iterations in printing values of B = 5193997
B = 10000000
```

testPreemptiveness - 6

With thread yield

```
[ruib@ruib-laptop testPreemptiveness]$ java TestPreemptiveness

Priority level of computation intensive threads? 10
Priority level of I/O intensive threads? 1
Priority of computation intensive threads = 10
Priority of I/O intensive threads = 1
I have already started the I/O intensive threads!
I have already started the computation intensive threads!
I have finished my job!
N. of iterations in printing values of D = 5053297
D = 10000000
A = 10000000
N. of iterations in printing values of A = 5151587
N. of iterations in printing values of B = 5461388
B = 10000000
N. of iterations in printing values of C = 5392911
C = 10000000
```

Threads in Java - 9

One should notice that the field `a` of the reference data type `Variable` includes the modifier `volatile`. Its precise meaning is to inform the Java compiler that the *threads*, during their execution, must permanently observe a *consistent* value in the variable `a`.

Consistency means in this sense that access to the variable `a` should always take place in the exact manner prescribed by the code of each *thread*.

This information is crucial here because Java memory model allows the compiler, when generating the *bytecode* of a given reference data type, as well as Java virtual machine, when interpreting this *bytecode*, to perform code optimization which, being totally consistent in a *singlethreaded* environment, may produce a paradoxical execution in a *multithreaded* environment.

Threads in Java - 10

A Java *multithreaded* application ends in principle when all its constituent *threads* terminate. In complex applications, where the number of support *threads* is very large, dealing with exceptional situations that may require aborting the operations, can become rather strenuous and demand the introduction of specific code whose practical utility is questionable.

To simplify the problem, Java presents two alternatives

- *calling the method* `atus)` – which forcibly terminates the Java virtual machine, returning the communicated *status* of operation
- turning the instantiated *threads*, directly or indirectly, created from the *thread* `main` into *daemons* – the Java virtual machine terminates as soon as all the remaining *threads* have this property.

Threads in Java - 11

Nevertheless, the usual termination of a *multithreaded* application is done by making the first or principal *thread* waiting for the termination of all the *threads* that may have been created from it.

In Java, one has a similar situation. The reference data type `Thread` has a method called `join` which, as it is tradititional in concurrent programming, and in a object oriented perspective, blocks the calling *thread* until the referenced *thread* ends.

Suggested reading

- *Distributed Systems: Concepts and Design, 4th Edition, Coulouris*, Dollimore, Kindberg, Addison-Wesley
  - Chapter 6: *Operating systems support*
- *Distributed Systems: Principles and Paradigms, 2nd Edition,* Tanenbaum, van Steen, Pearson Education Inc.
  - Chapter 3: *Processes*
- *On-line* support documentation for Java program developing environment by Oracle (Java Platform Standard Edition 8)