Distributed Systems

Chapter 2

Change history

Date	Changes	Name
x.x.2022	Completion of the slides	Khader Karkar
29.03.2024	Revision of the slides 1 to 33	Bora Derici
31.03.2024	Revision of the slides 34 to 61	Bora Derici

2.0 Concurrency in Java

Concurrency* is implemented on operating systems through two different concepts:

Process or **Threads**.

In Java programming language, developers only have the option to create and use Threads. However this doesn't cause any major problems. Due to the object paradigm each thread has its own encapsulated data.

Concurrency: Multiple computations are happening at the same time.

2.1 Definition

A Thread [w] is a single, sequential processing segment inside a process [w]. Each thread is characterized by a **starting point**, a **sequential procedure** (subroutine) and an **ending point**.

For a thread to execute instructions, a process needs to be started. Each process gets a **control thread**. The differences between processes and threads can be seen in Table 2.1.

Table 2.1: Difference between Process and threads

Processes	Threads
Own address space [w]	Program Counter (PC) [w]
Global Variables [w]	Own stack
Opens files	Registers
Child Process [w]	Child thread
Virtual sharing of main memory [w]	Status
Timer	
Signal	
Synchronization variables	
Accounting information	

Functions, Subroutines, Procedures, Methods etc.

The term **function** will be comprehensively used for terms like subroutine , procedure , method , function etc. **Subroutine** [w], appears in various basic languages and refer to a “function call” with no return value.

In Pascal this is referred to as a **procedure**.

A function is a Subroutine/Procedure that have a return value. (in Basic , Pascal , C ,C++ etc.)

If a function is assigned to an object (in object oriented programming), it is called a **method**.

Multiple threads share resources within the process

When multiple threads run in parallel within a process, they must share the resources of the process.

For example, accessing the same global variables, opening files, etc. Therefore, it is possible for multiple threads to access a global variable simultaneously. It is often necessary that such access is synchronized so that no information is lost due to multiple data changes at the same time.

Most programming languages don’t support running multiple threads in parallel natively.

With the exception of Ada, C# and Java where threads are a native part of the language.

Other languages like C,C++, Fortran and modula-2, the threading functionality is provided by independent programming libraries. In Unix-Systems, the usage of the library POSIX-Thread [w] is widespread, even the Java-Thread was implemented with the assistance of POSIX on some systems

2.2 User-level Threads and Kernel-Level Threads

User-level thread	Kernel-level thread
User thread are implemented by users.	Kernel threads are implemented by kernel.
Kernel doesn't recognize user level threads. (leads to unfair scheduling since the kernel only see the process and not the threads' context)	Kernel threads are recognized by kernel.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
User level threads are designed as dependent threads.	Kernel level threads are designed as independent threads.
Example : Java thread, POSIX threads.	Example : Window Solaris.

Scheduling

Kernel-Level threads are scheduled by the kernel. The best Scheduling algorithms for User-Level Threads can be implemented within the application. However, In Multiprocessor systems, parallel execution of User-Level Threads in a Process can only be implemented if the Process' Program is running on multiple Kernel-Level Threads.

For examples Java VMs* run under Solaris, With multiple lightweight processes. Each Lightweight Process have it's own Kernel thread. These lightweight Processes act like an interface between user and kernel threads, and thus the parallelization of multiple Threads in Java on multiple Processors is Possible.

VMs: Virtual Machines Wikipedia

2.3 Threads on Hardware Level

Threads can also be implemented directly at the hardware level. The advantage is that context switching and also creation are very fast. Threads at the hardware level were implemented, for example, by transputers and used by the distributed operating system Helios.

With its Hyperthreading technology, Intel now offers the option of running multiple threads in parallel on one processor.

2.4 Thread and Process states

2.4.1 Process state

A process may occupy a variety of states , the kernel or operating system may not necessarily recognize these distinctive states but they are a useful abstraction to understand the concept of processes.

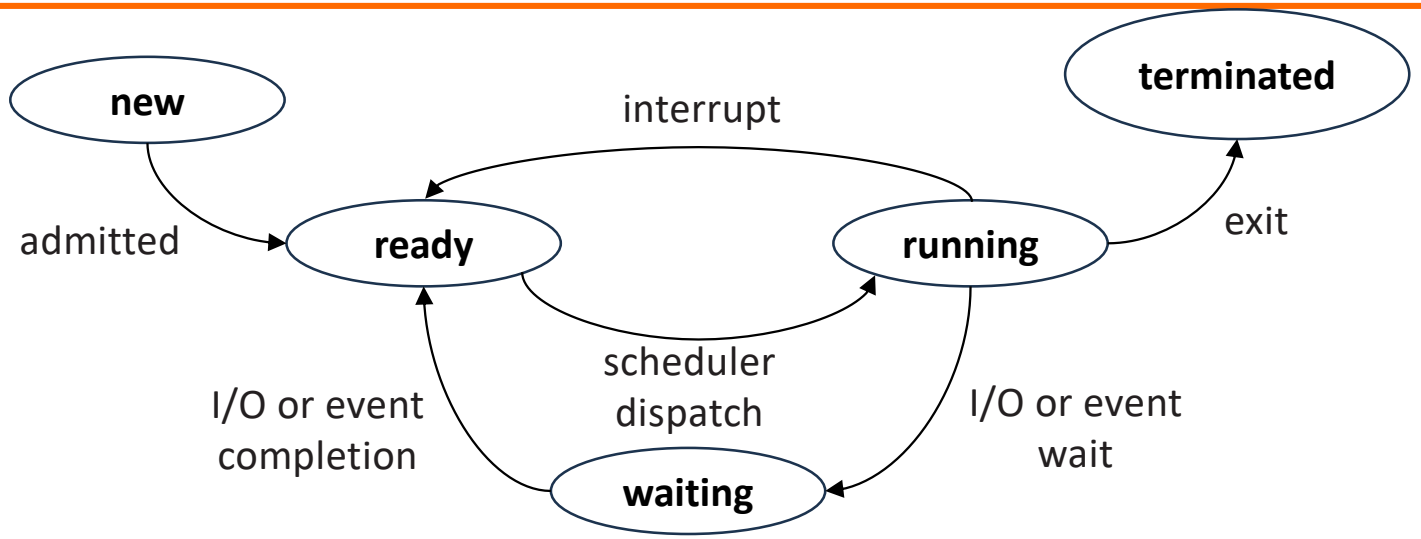
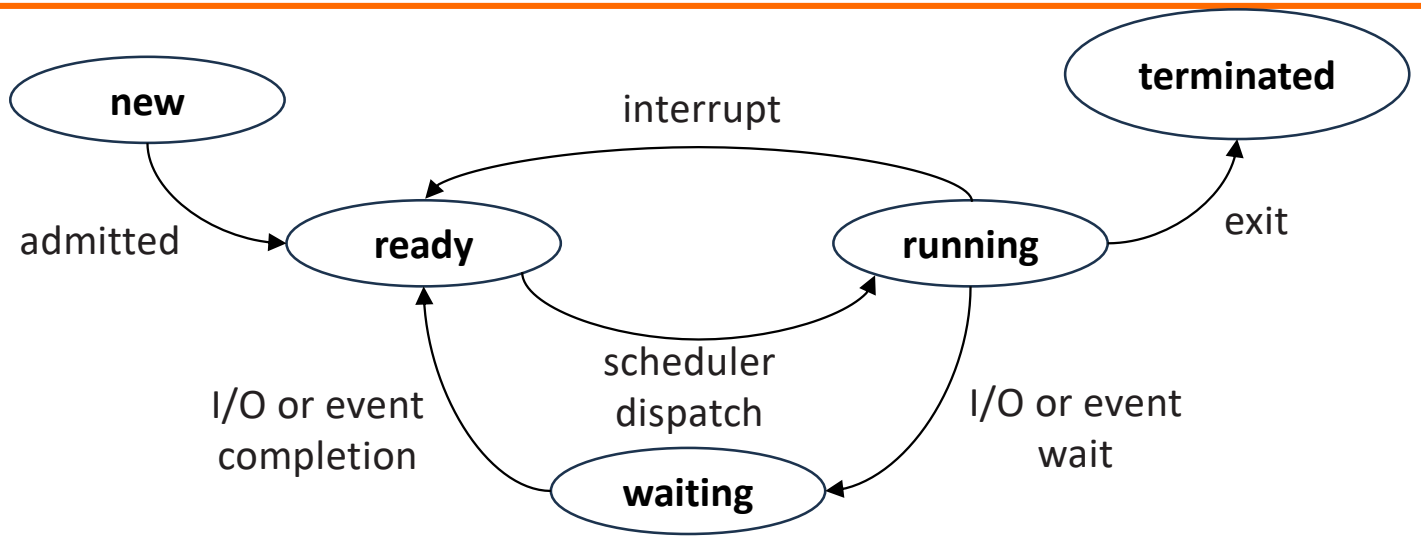


Figure 2.1: Process state diagram

State	Meaning
New	The Process is being created
Running	Instructions are being carried out (executed)
Waiting	The Process is waiting for an event to occur (E.g.: I/O Operation , receiving a signal , etc.…)
Ready	The Process is waiting to be assigned to a processor
Terminated	The Process have Completed Execution

Table 2.2: Overview of process states



Once the process has been created, it transitions from the new state to the **ready** state. There, the process is generally chained into a queue and is then allocated a free processor as soon as all other processes that entered the queue before it have been or are currently being processed by a processor. If it is interrupted in its execution, it is directly chained back into the ready queue. In the case of waiting an input-output operation (I-O) or other events, the process is chained into a separate queue, that of the waiting state.
From there it returns as soon as the event or I/O is complete. When the process finishes its execution, it changes from the running state to the **terminated** state.

2.4.2 Thread States

State	Meaning
New	The Thread is willing to be computed (the Thread is waiting it's execution)
Running	The Thread's instruction are being executed
Blocked	The Thread is Waiting for an event(example : I/O operation)
Terminated	The Thread Has completed execution

Table 2.3: Overview of general thread states

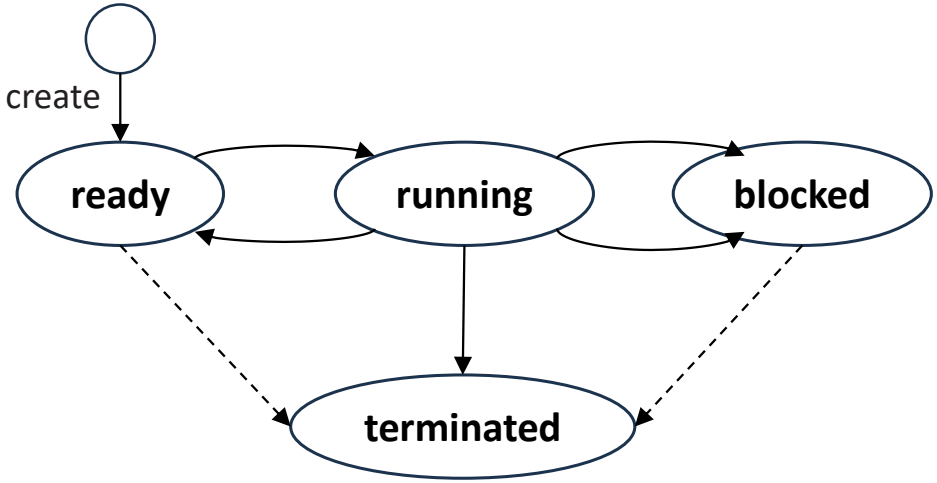
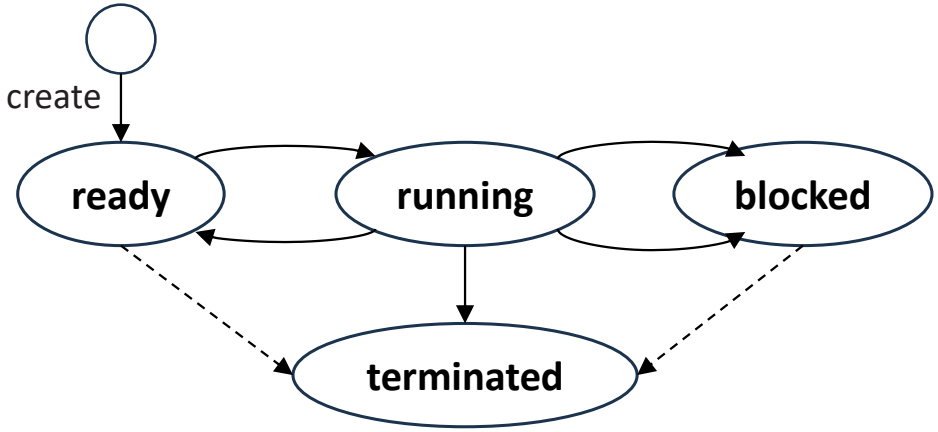


Figure 2.2: General Thread States



After its creation, a thread is in the **ready** state. When scheduled to run by the scheduler, it transitions to the **running** state.

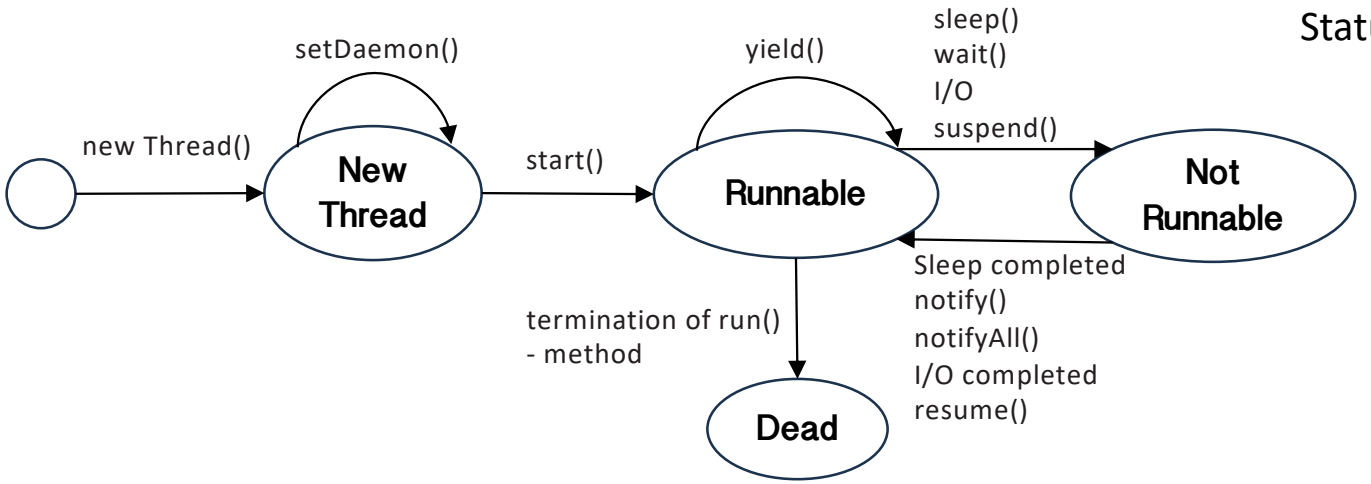
When, For example: The thread is waiting for an input, it changes to the **blocked** state.

When a thread has completed its task, it enters the **terminated** state.

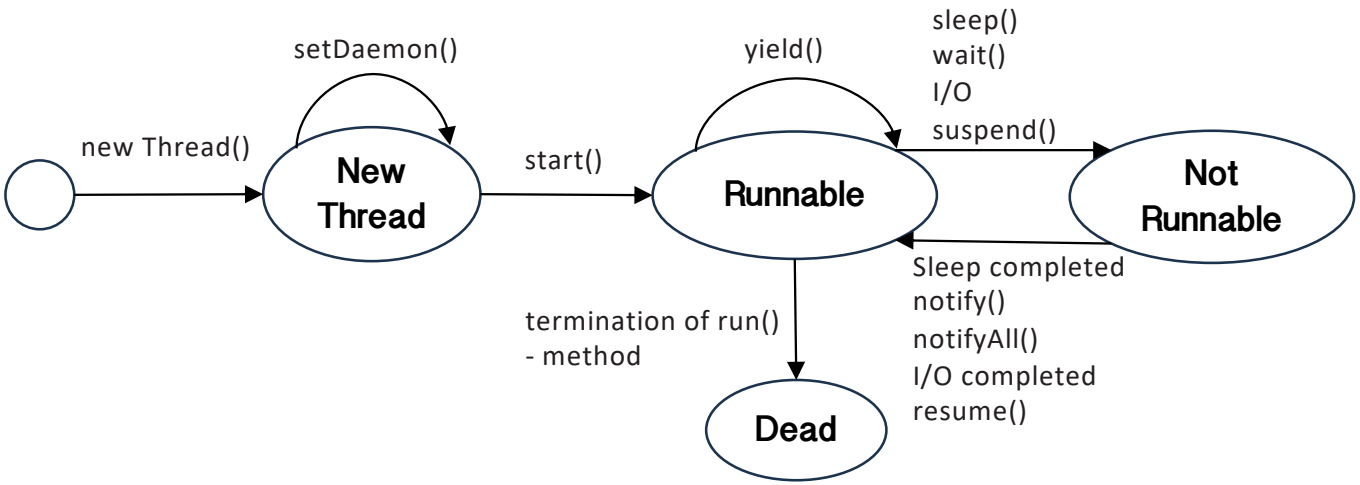
However, since threads are terminated when their process ends, a direct transition from the blocked or ready state to the **terminated** state is also possible.

2.4.3 Java Threads

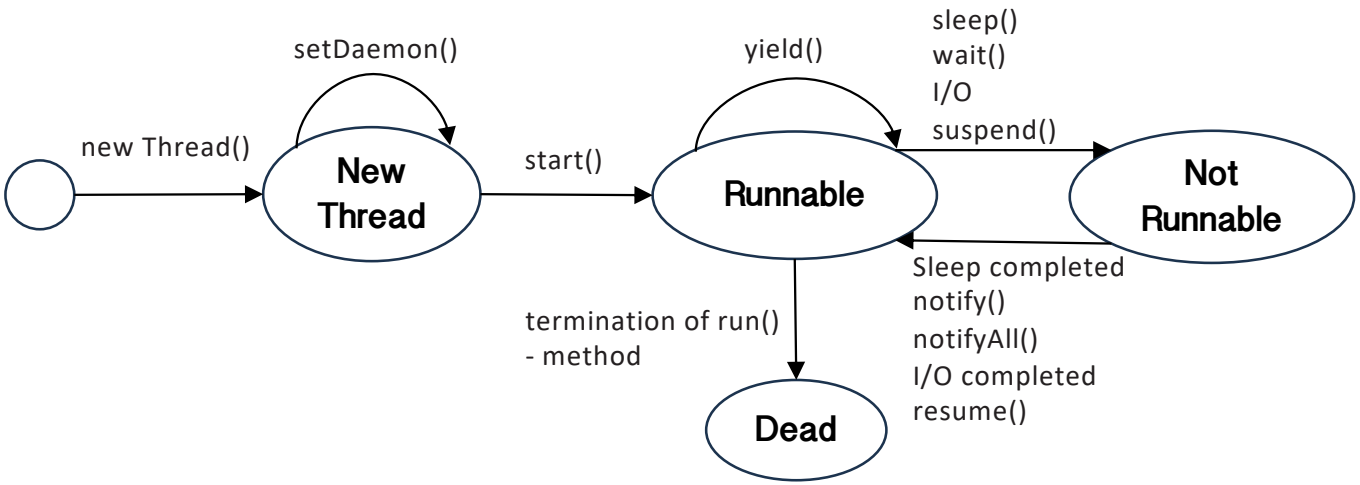
Figure 2.3: Java Threads' Status-Diagram



Almost every Java program requires multiple threads. For example, as soon as a Swing interface, a server or an applet comes into play, the program consists of several threads. Therefore, worth taking a closer look at threads in Java. The Java programming language is based on the general thread states. As Figure 2.3 shows, the states defined in Java differ only slightly from the general ones.



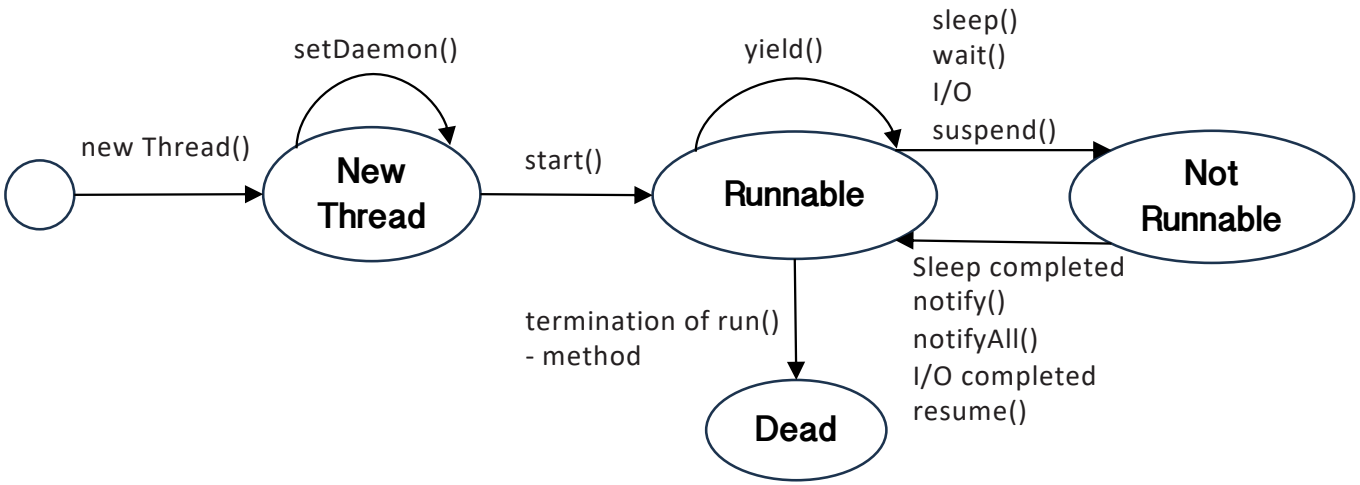
State	Meaning
New Thread	The thread have just been created, in order to be executed in parallel, the method start() must be called
Runnable	The thread is executable, or is already in execution
Not Runnable	The thread is not willing to execute or is inexecutable
Dead	The thread had completed the execution of run()-method



setDaemon()

In the Java programming language, a Thread object is in the **New Thread** state after it is created.

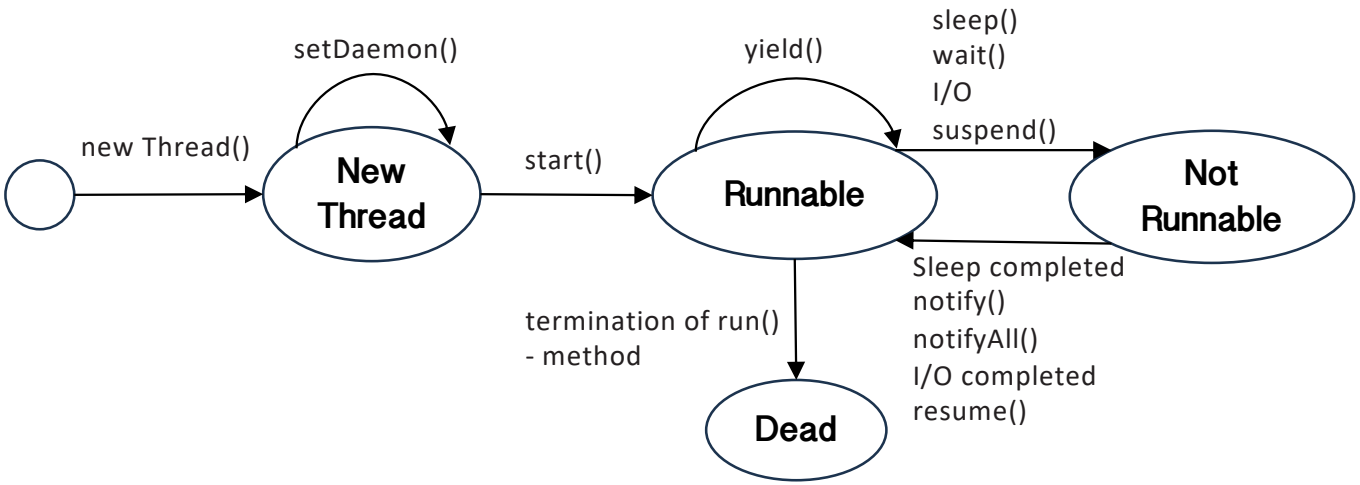
While the object is in this state, the setDaemon() method can be used to determine whether the thread should become a daemon or a user thread. A daemon thread is a: thread created to support user threads. As soon as all user threads are terminated, the JVM terminates the process and thus also the daemon threads. For example, the garbage collector is a daemon thread.



start()

Calling the start( ) method spawns a new thread that executes the Thread object's run() method.

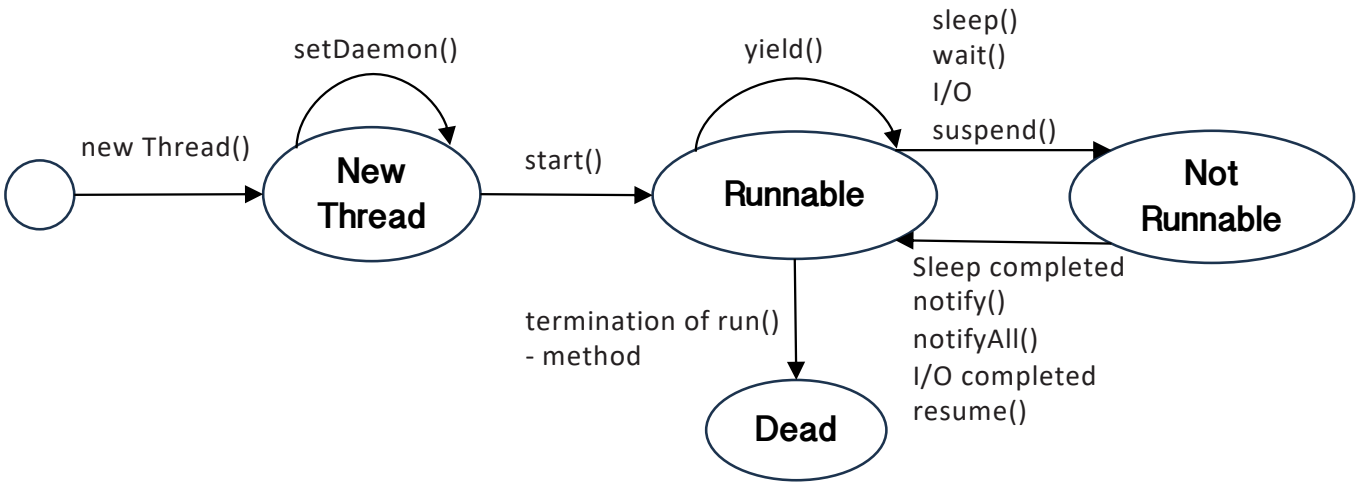
The thread is placed in the Runnable state by calling the start( ) method.



yield()

In the Runnable state, a thread can give away its time slice using yield().

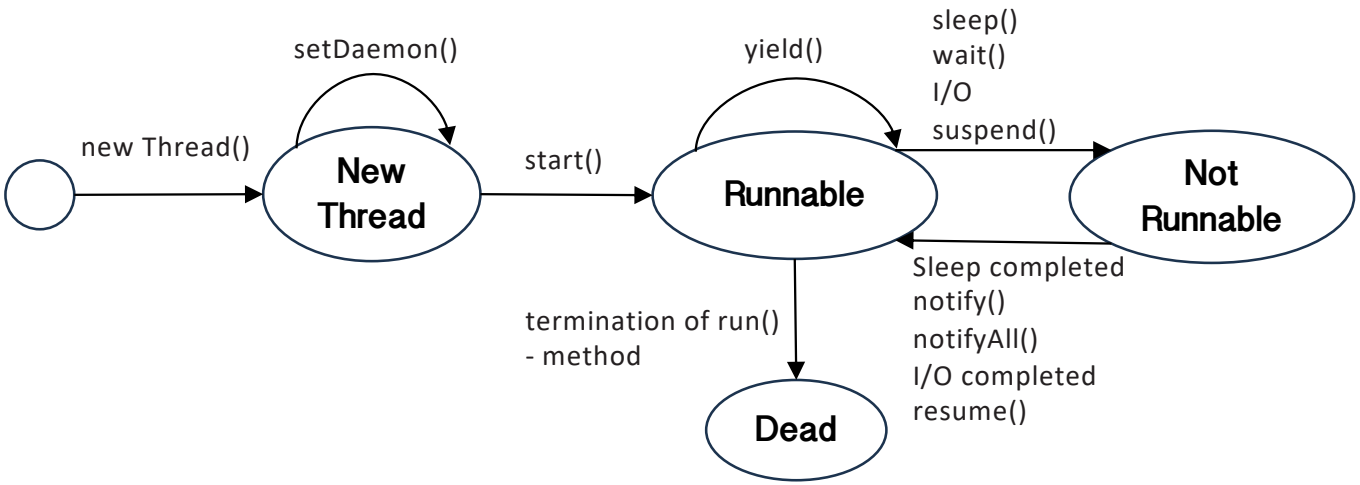
Before the thread is determined for execution again by the scheduler, an attempt is first made to entrust another thread with the execution.



wait(), sleep(), suspend()

The thread enters the **Not Runnable** state when it is waiting for a monitor condition.

1. using wait().
2. when it has been put to sleep for a specific time using sleep().
3. when it has been explicitly put into the Not Runnable state using suspend()
4. or at blocking input and output.

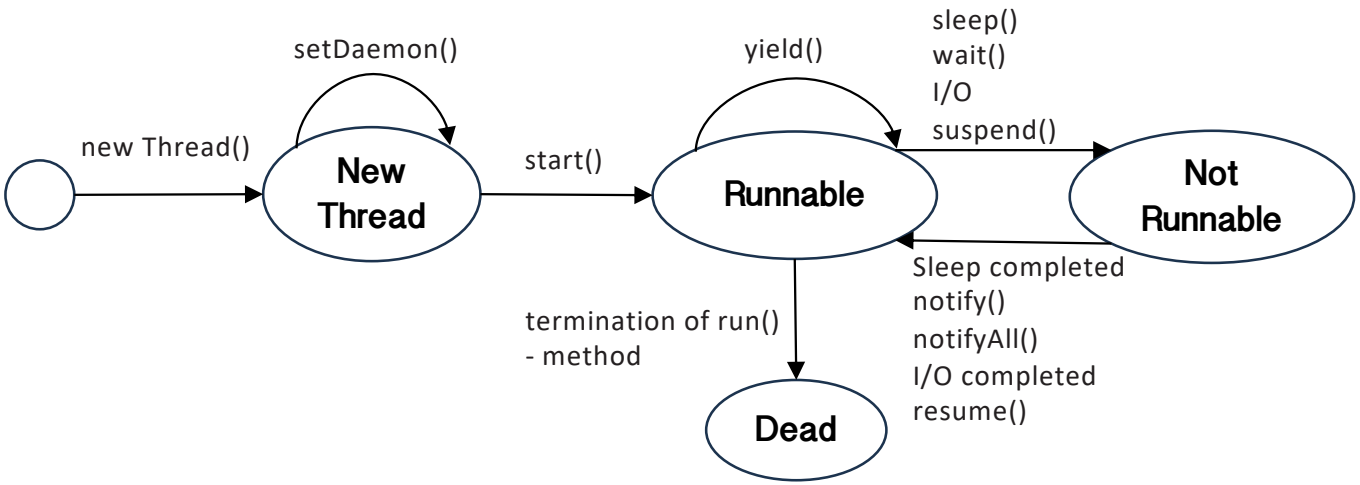


notify(), notifyAll(), resume()

when the input and output has been completed the thread returns accordingly , it is woken up from the monitor

1. when calling notify() or notifyAll(),
2. when the timer of sleep() has expired
3. or it is explicitly switched back to the monitor by resume() ,

state Runnable is set.



Ending run()-Method()

The thread ends it's execution when the run( ) method has executed the last statement and transitions to the **Dead run() state**.

Thread Levels

Since Java is platform-independent, Java threads are implemented above other individual implementations.

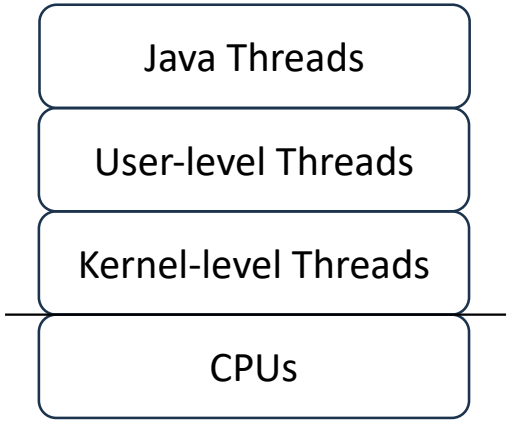


Figure 2.4: threads' levels on top of the CPU.

Note on thread mapping.*

** (for explanation purposes only)*
Source: Thread models and virtual processors - IBM Documentation

- Different Operating systems **maps** different user-level threads differently , there are multiple standard models that operating systems can follow:
  - M:1 model
  - M:N model
  - 1:1 model
- **In the M:1 model**, all user threads are mapped to one kernel thread;
  - all user threads run on one VP(virtual processor). The mapping is handled by a library scheduler.
  - All user-threads programming facilities are completely handled by the library.
  - This model can be used on any system, especially on traditional single-threaded systems.

- **In the 1:1 model**, each user thread is mapped to one kernel thread;
  - each user thread runs on one VP.
  - Most of the user threads programming facilities are directly handled by the kernel threads. This model is the default model.
- **In the M:N model,** all user threads are mapped to a pool of kernel threads;
  - all user threads run on a pool of virtual processors.
  - A user thread may be bound to a specific VP, as in the 1:1 model. All unbound user threads share the remaining VPs. This is the most efficient and most complex thread model;
  - the user threads programming facilities are shared between the threads library and the kernel threads.

Java Threads mapping

Java threads are mapped to either kernel-level threads or user-level threads. Under Windows NT, threads created by the user are treated directly as kernel-level threads.

For example, Under Solaris, Java threads are mapped 1:1 to Solaris threads (user-level threads). These are then mapped 1:1 or m:n to lightweight processes (LWPs) and these in turn are mapped 1:1 to kernel threads. More in section 2.6

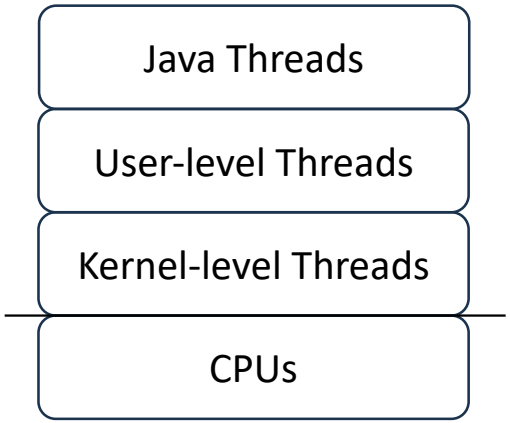


Figure 2.4: threads' levels on top of the CPU.

2.5 Mapping Java Threads with POSIX threads

Since Java threads are implemented in various UNIX operating systems with the help of the POSIX thread library, the mapping of the Java thread methods to the POSIX thread functions should be considered here as an example. The prerequisite for this mapping is that the POSIX library has at least the same functionality provides such as Java threads.

POSIX Threads	Java Threads
pthread_create(… . func. args)	Thread(Runnable target)
pthread mutex mutex	
pthread_mutex_init(mutex)	
pthread mutex lock(mutex)	synchronized(object){
pthread_mutex_unlock(mutex)	}
pthread_cond_init()	
pthread_cond_wait(c. m)	obj.wait()
pthread_cond_timedwait(c. m. t)	obj.wait(t)
pthread_cond_signal(c)	obj.notify()
pthread_cond_broadcast(c)	obj.notifyall()

Table 2.5 provides a brief (partial) overview of the most important POSIX thread statements and functions under the C programming language and their Java equivalents.

In the pthread_create(...) function, the function that is to be executed as a new thread is specified via func. In Java, this is possible using the constructor of the Thread(Runnable target) class. The target object contains the run( ) method that is to be executed as a new control thread.

Every object in Java is created with a lock (mutex lock). This lock is taken when entering a synchronized block. If another thread requests a lock on the same object, that thread is blocked until the lock is released. The locks are used to guarantee mutual exclusion to a code segment. Each synchronized block is implemented using the mutex variable of the object specified in the parentheses of the synchronized statement. Using synchronized to synchronize entire methods is equivalent to using a synchronized(this) block that spans the entire method, as shown in the following program:

```
public class Example{
   private int result;
   ...
   public synchronized void add(int a){
      result = result + a;
   }}
```

```
public class Example{
   private int result;
   ...
   public void add(int a){
      synchronized (this){
         result = result + a;
      }
   }}
```

Program 2.1 : Equivalent use of the synchronized statement

The initialization is done automatically in Java when the object is created. More about synchronization using mutex variables, semaphores and monitors as well as critical areas in Chapter 3: Synchronization.

With the POSIX threads, the mutex variable of type pthread_mutex must first be declared and then initialized with pthread_mutex_init(...). Calling pthread_mutex_lock(...) ensures that any subsequent calls to this function on the same mutex variable will block the calling thread. The mutex variable is released again with pthread_mutex_unlock(...).

In Java, this is realized by the synchronized block. The opening parenthesis can be compared to requesting the lock, the closing parenthesis to release it.

The blocking has the advantage that you don't accidentally forget to release a mutex variable and thus provoke a deadlock, since the lock is automatically released again at the end of the block. The downside, however, is that a lock cannot be released from anywhere else in the program.

POSIX threads can wait for a specific condition variable using the pthread_cond_wait(c.m) function. With this call, both the condition variable c and the mutex variable m are specified, which must be released so that another thread can enter the critical area. The pthread_cond_timedwait(c.m.t) function can also be used to specify a maximum time t that the thread should wait for a "wake-up signal".

A blocked thread that is waiting for the condition variable c is woken up via pthread_cond_signal(c). The function pthread_cond_broadcast(c) wakes up all blocked threads that are waiting for the corresponding condition variable [IEEEI003.1/20031].

In Java, the signaling is implemented using the methods wait(),notify() and notifyAll(). These methods may only be used within a synchronized block. wait() causes the mutex variable to be released and the calling thread to be blocked. The pendant to pthread_cond_timedwait(c. m. t) is realized with wait(t). A thread that is waiting at the monitor is woken up again by notify(). notifyAll() wakes up all threads that are waiting at the monitor.

The other methods of the Java threads can also be mapped to the POSIX library. More information on the POSIX library can be found in [SUN/MPG 2002] and [IEEEI003.1/2003].

2.6 Scheduling threads on Solaris 9

In order to make the management of threads by the operating system more understandable, the scheduling on a specific operating system (Solaris 9 from Sun Microsystems) is described in this chapter. This chapter also looks at how Java threads are mapped to the native threads.

2.6.1 Thread Libraries

Two different thread libraries are available under Solaris 9, which implement two different threading models [SUN/TPC 2004]. The "Tl" library, the standard library for all predecessors of Solaris 9, implements an m:n threading model. m:n means that m native threads are mapped to n LWPs(Lightweight processes) (see Figure 2.5).

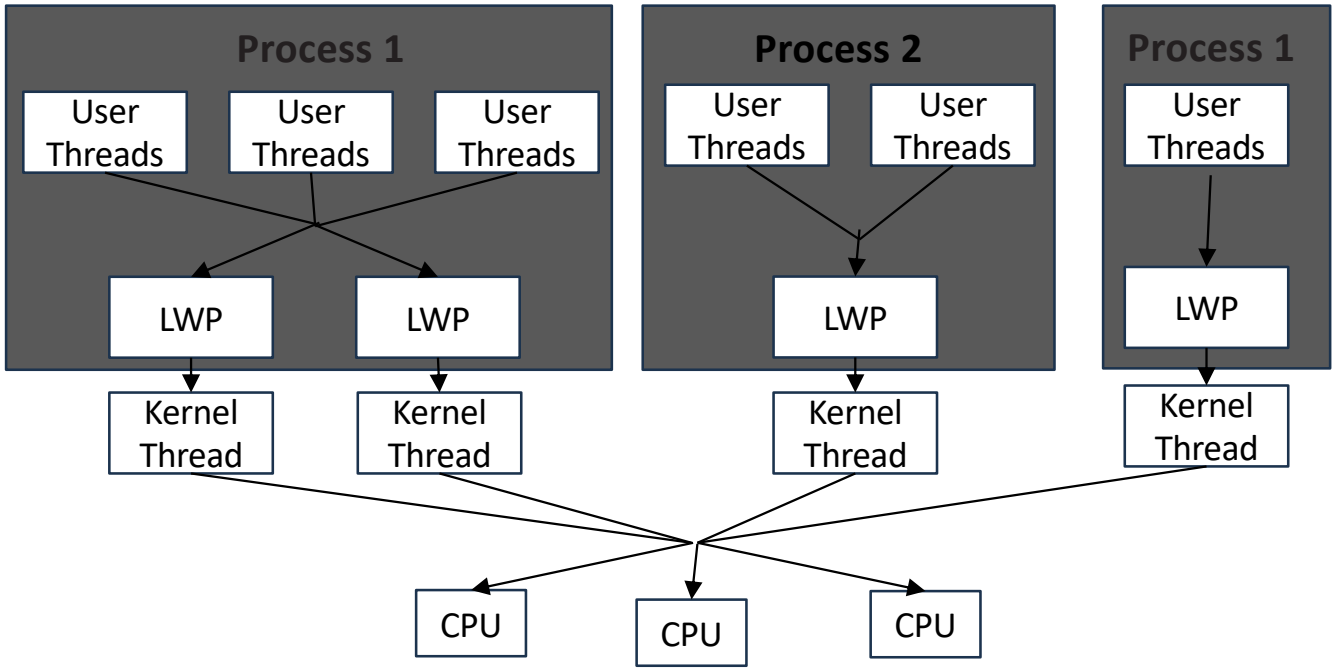
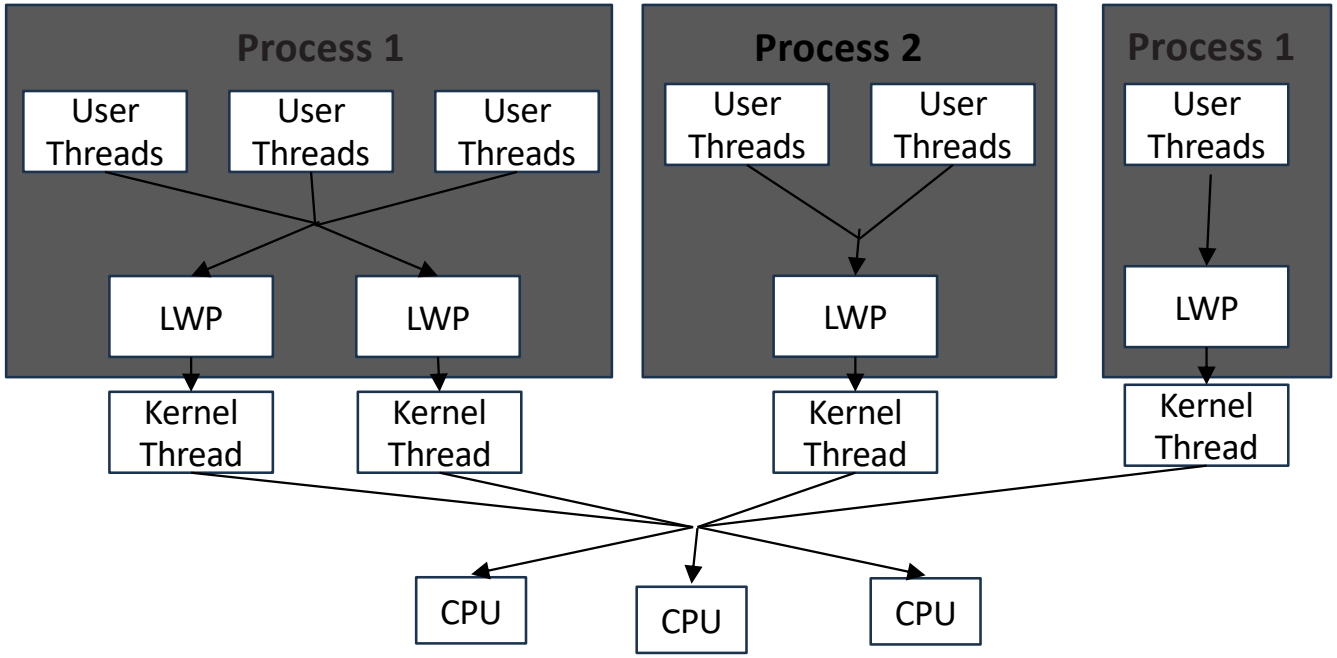


Figure 2.5: m:n threading model in Solaris



Each LWP is in turn assigned a kernel thread [Sun/Threading]. The kernel threads can then be executed directly by a CPU. A program (e.g. the JVM) consists of several LWPs. The user threads can then be distributed to the various LWPs of the program.

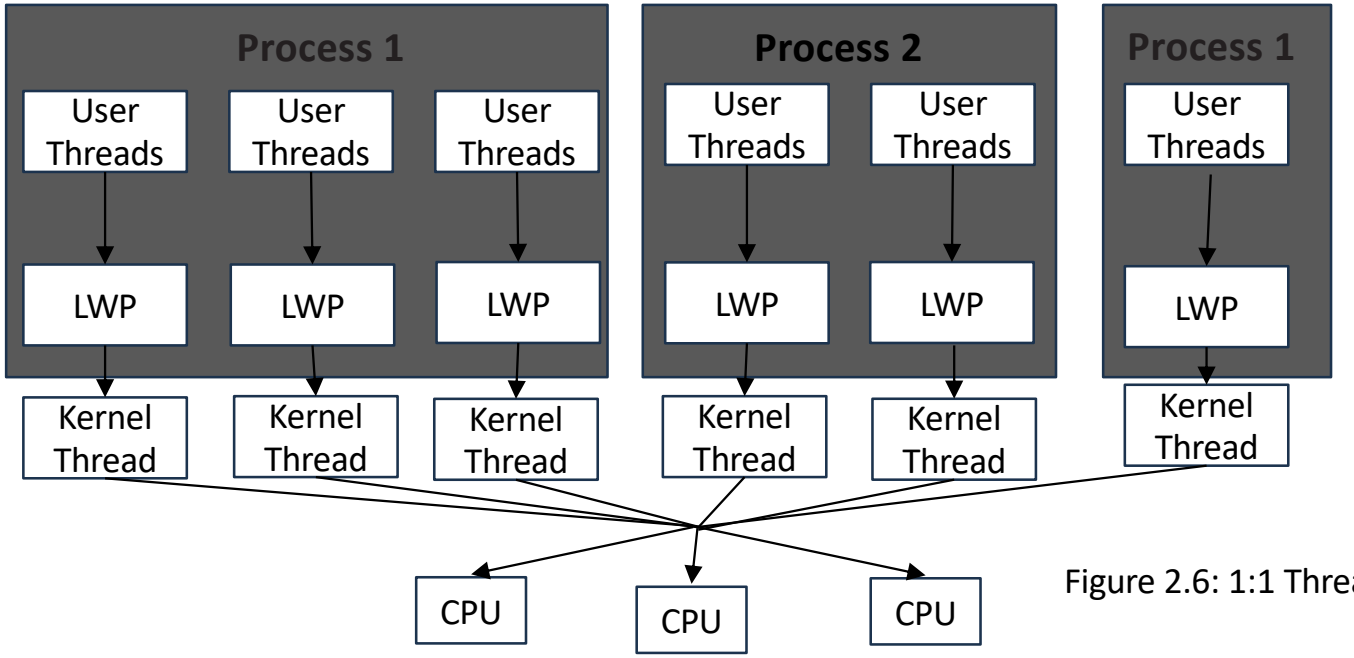
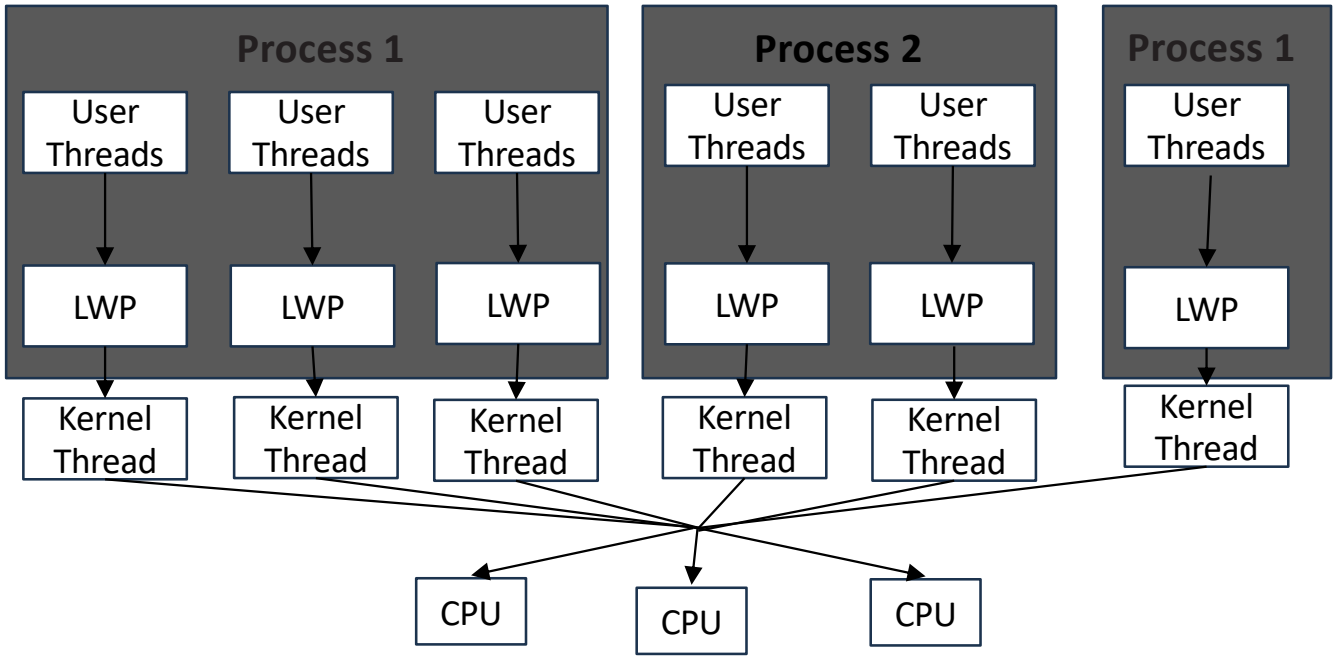


Figure 2.6: 1:1 Threading Model on Solaris

The "T2" thread library, the standard Solaris 9 thread library, implements a **1:1 threading model**, i.e. an LWP is created for each user thread. Each LWP is still mapped to a kernel thread. During its entire lifetime, the thread does not change the LWP.



This scheduling is more robust and simpler [Sun/TPC 2004]. For each Java thread, a Solaris thread (user thread) is created. Since each user thread either runs on exactly one LWP and thus kernel thread (1:1 threading model) or on several different LWPs, each of which is assigned a kernel thread (m:n threading model), user threads, and thus also Java threads be parallelized within a process on a multi-processor system.

2.6.2 Creation of Lightweight Process

Since real parallelization of Java threads is possible under Solaris, the question arises as to how the JVM performs the parallelization. The 1:1 threading model means that the Java threads are assigned to LWPs by the thread library. Therefore, no additional work is required during implementation. With the m:n threading model, the programmer has the option of including the libthread.so library in his program via the linker during the compilation process. As soon as the program starts, a background process called dynamiclwps is created. This process is the "Dynamic LWP Scheduler". It subscribes unassigned user threads to free LWPs. If an LWP is not needed for about 5 minutes, it terminates [Gbodimowo 2001]. The exact procedure is implemented via door mechanisms and signals [Mauro 2001].

A Java Virtual Machine could, for example, implement the mapping of Java threads to LWPs using one of the two methods. In the JVM specification [Lindholm 1999] there is no explicit provision for the mapping of Java threads to kernel-level threads.

2.7 Using Java Threads

Now that we know how the operating system deals with threads, let's look at how Java uses threads.

First of all, a Java thread has two very important methods: start() and run().

The start( ) method handles parallelization, i.e. it is responsible for actually starting a new thread. The new thread then calls the run( ) method.

The run( ) method tells the thread what instructions to run in parallel. Thus, the run() method represents a new execution thread.

2.7.1 Starting Threads

There are several options for specifying the content of the run( ) method. Inheritance is the simplest option here. A class needs to be designed that inherits from the Thread class and overrides its run() method. An object of this class is triggered by Thread's original start( ) method and then executes its own run( ) method in parallel (see Program 2.2).

In this example, another thread is started from the main thread. Both threads increment the value of variable i. By calling the static method Thread.currentThread(), a reference to the currently running thread can be returned and a distinction can thus be made between the two running threads. In order to be able to assign the output of the two threads, their names are also output in addition to the result i.

```
public class Main{
   public static int i = 0;
   public static void main(String[] args){
      MyThread m = new MyThread();          // Create Thread
      m.setName("MyThread");
      m.start();                            //start thread
      for(int j = 0; j < 1000000000; j++) i++;//increment i
      System.out.print(Thread.currentThread().getName());
      System.out.println(" i: " + i);
   }
}
public class MyThread extends Thread{
   //These methods will be executed in parallel
   public void run(){
      for(int j = 0; j < 1000000000; j++) Main.i++; //increment i
      System.out.print(Thread.currentThread().getName());
      System.out.println(" i: " + Main.i);
   }}
```

Program 2.2: Realization of concurrency through inheritance

However, the disadvantage of inheritance is that the class that inherits from the Thread class cannot inherit from any other class. Therefore, a second way of using threads was provided, namely through the Runnable interface. The Runnable interface consists of the run() method. Therefore, the run() method must be implemented by the class that wraps the interface. An object of this class is then passed to the constructor when a thread is created. Once the thread's start() method is called, a thread is actually started in parallel. This executes the run() method of the Thread class. This in turn calls the run() method of the passed Runnable object (see Program 2.3).

```
public class Thread implements Runnable{
   ...
   /* What will be run. */
   private Runnable target;
   ...
   public void run(){
      if(target != null) target.run();
   }
   ...
}
```

Program 2.3: Sample from java.lang.Thread

Using the Runnable object, the 2.2 program looks like this:

This ability to run threads has one major advantage.

It is still possible for the class that implements the Runnable Interface to inherit from another class.

When inheriting directly from the Thread class, this is not possible.

```
public class Main{
   public static int i = 0;
   public static void main(String[] args){
      MyThread m = new MyThread(); // Create Runnable object
      Thread t = new Thread(m); //Creating Thread
      t.setName("MyThread");
      t.start();                    //starting thread
      for(int j = 0; j < 1000000000; j++) i++;
      System.out.print(Thread.currentThread().getName());
      System.out.println(" i: " + i);
   }
}
public class MyThread implements Runnable{
   //These methods will be executed in parallel
   public void run(){
      for(int j = 0; j < 1000000000; j++) Main.i++;
      System.out.print(Thread.currentThread().getName());
      System.out.println(" i: " + Main.i);
   }}
```

Program 2.4: Realization of concurrency through a Runnable Object

As a third option, the initialization of the thread can be transferred to the class that implements the runnable interface. This solution is the clearest and has the advantage that the class - as with the inheritance solution - knows its own thread (see program 2.5). The only disadvantage (or also advantage) is that starting the thread happens in the same class. So it is no longer possible (or necessary) to instantiate a thread and start it **later**.

With this solution, the main program only has to create an object of the class that implements the runnable interface. This class passes the thread a reference to itself and then starts the thread.

```
public class Main{
   public static int i = 0;
   public static void main(String[] args){
      MyThread m = new MyThread();       //Creating Runnable-Object
      for(int j = 0; j < 1000000000; j++) i++;
      System.out.print(Thread.currentThread().getName());
      System.out.println(" i: " + i);
   }
}
public class MyThread implements Runnable{
   //Reference to thread that executes the run() method of this class
   Thread t;
   public MyThread(){
      t = new Thread(this);      //Creating Thread
      t.setName("MyThread"); //Giving a name to the thread
      t.start();                 //Starting thread
   }
   public void run(){
      for(int j = 0; j < 1000000000; j++) Main.i++;
      System.out.print(t.getName());
      System.out.println(" i: " + Main.i);
   }}
```

Program 2.5: Realization of concurrency by implicit thread instantiation

The fourth and last possibility is realized via an (anonymous) inner class. This procedure has the advantage that the number of Java files in the program does not increase unnecessarily and is used when the class is only required at one point in the program.

Another advantage of this possibility is that the variables of the outer class can be accessed from the inner class.

```
public class Main{
   public static int i = 0;
   public static void main(String[] args){
      Thread t = new Thread(){
         //These methods will be executed in parallel
         public void run(){
            for(int j = 0; j < 1000000000; j++) i++; //increment i
          System.out.print(Thread.currentThread().getName());
            System.out.println(" i: " + i);
         }
      };
      t.setName("MyThread"); //starting thread
      t.start();
      for(int j = 0; j < 1000000000; j++) i++; //increment i
      System.out.print(Thread.currentThread().getName());
      System.out.println(" i: " + i);
   }}
```

Program 2.6: Realization of concurrency through an inner class

Regardless of which of the four suggested solutions you consider, the expected result should always be the same. However, if you start a single program of the four proposed solutions several times, you will usually get different results. For example, a run through the program produced the following result:

MyThread i: 1364082967

main i: 1371634760

Looking at the different results, it also becomes clear why synchronization mechanisms are needed. In the above program, one would actually expect that one of the two execution threads with i == 2000000000 ends the program. However, since the variable i is accessed at the same time, the value of i can be read by both threads. As a result, both threads have their own copy of the variable they want to increment. If both threads now increment and write back the variable, an increment has been lost. This becomes clear in the following example.

There are two threads t_1 and t_2. The integer variable i starts with the value 6. Table 2.6 shows a possible scheduling.

Instructions	Resulting Value
t_1.read(i)	t_1.i == 6
t_2.read(i)	t_2.i == 6
t_1.inc(i)	t_1.i == 7
t_1.inc(i)	t_2.i == 7
t_1.write(i)	i == 7
t_1.write(i)	i == 7

Table 2.6: Possible scheduling of two threads

If two increments are made, the value 8 should actually come out as a result. However, without synchronization, the result depends on the order of execution. For more information, see Chapter 3: Synchronization.

2.7.2 Threads termination

If threads are to be terminated prematurely, each thread provides the method stop(). However, problems arise here: objects can be in an inconsistent state after termination, locks may remain in use.

Experience has shown that when a thread is the sole holder of a lock and no other thread is in the lock's queue, calling stop() will release that lock. As soon as another thread is in the queue, the lock is not released. This leads to the problem that the further course of the program cannot be predicted, or that a deadlock can result if the lock is not released.

Therefore, a safe thread termination method is presented. However, this implementation must be carried out by the application programmer.

With a longer run() method, it is possible to use:

if(!runnable)
break;

To Exit the

while (runnable)

Loop, unless the if statement is used inside another loop.

```
public class StoppableThread implements Runnable{
   Thread t;
   boolean runnable = true;
   public StoppableThread(){
      t = new Thread(this);
      t.setName("StoppableThread");
      t.start();
   }
   public void run(){
      while(runnable){
         //do work
      }
   }
   public void setRunnable(boolean runnable){
      this.runnable = runnable;
   }
}
```

Program 2.7: Terminable thread

2.7.3 Waiting for threads

The join() method allows a thread to wait for another thread to finish. The thread calling join() is blocked until the thread referenced through the join() method has completed its execution.

In this example, the main thread is waiting for the end of the thread it started. The thread started counting up to 1,000,000,000. Since the scheduler does not necessarily switch to the new thread immediately after starting the new thread, the main thread simulates work through **thread.sleep(100).**

Therefore, the newly started thread is also entrusted with execution.

If this line is omitted, the value of i before calling the join( ) method would almost always be 0, since it is very unlikely that there would be a context switch exactly between starting the thread and printing the text.

The output after calling the join() method shows that the restarted thread has completed the for loop.

```
public static void main(String[] args){
   Thread t = new Thread(){
      public void run(){
         for(int j = 0; j < 1000000000; j++) i++; //Increment i
      }
   };
   t.start(); //Start thread
   try{
      Thread.sleep(100); //Simulate work
      System.out.println("Before join: i = " + i);
      t.join();      //Wait till the end of thread t
   }catch (InterruptedException e){
      System.out.println("join failed");
   }
   System.out.println("After join: i = " + i);
}
```

Program 2.8: join()-method

2.8 Query for Thread Information

Since several threads are usually used in the implementation of servers, it is interesting to obtain as much information as possible about the number of threads, their states or the memory allocation of the program, not least to track down possible errors or bottlenecks in the server.

As of J2SE 5, Sun Microsystems provides **Managed Beans (MXBeans)** for the areas just mentioned. These are located in the java.lang.management package. An implementation of a specific MXBean can be requested through a ManagementFactory.

2.8.1 Query properties of running threads

The ThreadMXBean can be used to generate a list of all running threads. The information about each thread is stored in a ThreadInfo object:

```
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadInfo;
import java.lang.management.ThreadMXBean;
…
ThreadMXBean tb =ManagementFactory.getThreadMXBean();
ThreadInfo[] threadInfo;
threadInfo = tb.getThreadInfo(tb.getAllThreadIds());
```

Program 2.9: querying the thread status

The `getAllThreadIds()` method returns an array of all thread IDs. When this array is passed to the `getThreadInfo()` method, the appropriate `ThreadInfo` object is requested for each passed ID. `ThreadInfo` objects provide information such as the thread name, the thread status, the thread ID, the time a thread has been blocked, and so on.

You can query the status, the name and the lock on which a thread is blocked as follows:

```
String name = threadInfo[j].getThreadName();
String state =threadInfo[j].getThreadState().toString();
String lockName = threadInfo[j].getLockName();
```

Program 2.10: Query thread information

However, the getAllThreadIds() method only returns the IDs of the threads and no reference. This is desirable for designing distributed applications (e.g. through RMI) since references to threads started on other machines are useless. However, if there is a desire to obtain references to all threads - e.g. for the development of an administration or debugging tool - then this is possible using the static method enumerate() of the ThreadGroup class. This method makes it possible to query the references of all threads in a thread group. Since thread groups are hierarchical, it is possible to get to the root of the thread group by calling the getParent() method (repeatedly). You can get the ThreadGroup of a thread by calling its getThreadGroup() method. If the enumerate() method is executed on the root of all thread groups, you get a thread array that contains the references to all threads.

```
Thread[] t = new Thread[...];
//get the ThreadGroup of the currently active thread
ThreadGroup tg = Thread.currentThread().getThreadGroup();

//get the root of all thread groups
while(tg.getParent() != null){
   tg = tg.getParent();
      }
//save reference to all threads
int no = tg.enumerate(t);
```

Program 2.11: Query all thread references

2.8.2 Memory Usage

The memory consumption is of great importance for the implementation of highly parallel servers. If a large number of threads are already running, the question arises as to whether more threads should be created.

As program 2.12 shows, the memory can be queried using the new **MemoryMXBean**

```
import java.lang.management.ManagementFactory;
import java.lang.management.MemoryMXBean;
import java.lang.management.MemoryUsage;
…
MemoryMXBean mb = ManagementFactory.getMemoryMXBean();
...
MemoryUsage hmu = mb.getHeapMemoryUsage();
MemoryUsage nhmu = mb.getNonHeapMemoryUsage():
…
```

As you can see, both heap and non-heap memory can be queried. The heap memory is the memory in which the application runs and requests memory for new objects. Non-heap memory is memory used by the JVM itself

The following memory information can be queried with the two methods getHeapMemoryUsage() and getNonHeapMemoryUsage():

Memory Information	Description
initial heap size	Size of the heap when starting the JVM
used heap size	Memory currently in use
committed heap size	Memory allocated to the application
maximum heap size	Maximum available memory for the application If this memory size is exceeded, a java.lang.OutOfMemoryError occurs
initial non heap size	Memory allocated to the JVM when the JVM is started
used non heap size	Memory currently used by the JVM
committed non heap size	Memory allocated to the JVM
maximum non heap size	Maximum memory made available to the JVM

Table 2.7: Types of heap memory

The initial size and the maximum size of the heap memory can be specified when starting the JVM with the options -Xms and -Xmx.

2.8.3 System Architecture and Operating System

If servers are used on heterogeneous systems, it makes sense to make decisions about the type of server based on the system architecture, the number of processors, the operating system name and the operating system version. This information can be queried by the OperatingSystemMXBean.

```
OperatingSystemMXBean ob = ManagementFactory.getOperatingSystemMXBean();
...
String arch = ob.getArch();
int no = ob.getAvailableProcessors();
String name = ob.getName();
String ver = ob.getVersion();
...
```

Program 2.13: Querying the system architecture and operating system information

After explaining how threads work and how they are used in the Java programming language, one question remains unanswered: How can threads be synchronized with each other? The next chapter deals with this question.