Informatics and Mathematical Modelling
Technical University of Denmark
Building 321
DK–2800 Lyngby
Denmark

HHL 28–08–2008

Hans Henrik Løvengreen:

CONCURRENT PROGRAMMING PRACTICE 1

Processes and Threads

Version 1.5

In this note we show how the process concept appears in different languages or can be implemented by use of program libraries.

Contents

1 Introduction

The most central notion of concurrent program is probably the notion of processes. There are may ways to define this notion, but here we shall use the following general characteristics:

> A *process* is the behaviour of an distinguished program part to be executed independently. By behaviour we understand all potential executions of the program part.

Since the program part exactly is a description of how the process is going to proceed, we usually talk about the program part as "the process".

In practice, a process will be defined by a sequential program part as a sequence of actions (statements) that are to be executed one after the other. In some languages (e.g. Occam) any statement may be a process, but typically a process is related to an abstraction concept and thus takes the form of a procedure, a function, or an object.

In a concurrent program language, it must at least at the outermost program level be possible to delimit two or more processes to be run concurrently, i.e. potentially overlapping in time. In most languages, it is also possible to let a process fork into subprocesses or to activate a new process from a running one.

In this note, we shall see how to create (and terminate) concurrent processes in some current languages and operating systems.

As a common example it is shown how to make a program that creates three processes/threads and awaits their termination.

All of the languages and systems shown do, of course, also provide means of synchronization and communication betwen processes, but these means will be dealt with in other parts of this series. Also, it is not discussed how the execution of concurrent processes may be influenced by the programmer, e.g. by assigning priorities to the processes.

It has been tried to write the code as it would appear in the actual language. Some places, but not always, it has been tried to make the code robust towards exceptions etc. Finally, it should be noted that only a few a the examples have been checked syntactically not to mention being tested.

2 Threads versus processes

In many languages and operating systems *processes* and *threads* are separate notions. In the early development of operating systems the *multiprogramming technique* was introduced to switch the the execution among different user programs. Hereby the user programs were executed concurrently with each program corresponding to a process. A central task of the operating system became to protect the user programs against each other since they could not be assumed to cooperate—rather on the contrary! Apart from representing an independent activity, the process concept was extended to encompass also control of resources such as memory and files. As a consequence, the process concept has become "heavy" within operating systems.

Mean-while, for control applications, a number of simple *multiprogramming kernels* or *multitasking kernels* implementing concurrent processes were developed. For a control application, one may assume that the processes cooperate, and therefore such kernels do not spend much effort (i.e. time) protecting processes against each other.

In recent years, the two worlds have come closer: The usefulness of having a user program consists of several concurrent activities has been recognized. On the other hand, control applications demands more traditional operating systems facilities (file systems, user interfaces, etc.).

As a result, the two worlds are now amalgamating such than in operating systems we now see two levels of concurrent entities:

Process (operating system notion)
: A process is an encapsulation an activity, typically the execution of a user program (application). The process is associated with a number of resources (memory, files) which thereby become protected from other processes. Processes exist concurrently. The activity of a process is carried by one or more threads.

Thread
: A thread is an independent sequential activity within a process. A process may contain several concurrent threads, sharing the resources of the process. Threads within a process are not protected against each other by the operating system. Normally, all thread may act externally on behalf of the process. Threads are also known as "light-weight processes".

From the above, we see that a thread is the notion that corresponds best to our general understanding of a process. This gives rise to an unfortunate language confusion that we have to live with for some time.

2.1 Thread implementation

Threads may be implemented in principally two ways:

- Within the framework of a single process, one may provide a *thread library* that implements a mini-kernel switching the execution among threads. From the viewpoint of the operating system, the process looks like a single sequential program. This is often called *user-level threads*.

- Threads may be introduced as a built-in operating system notion. Hereby, it becomes the task of the operating system to switch execution among all threads in all processes. Often we say that the operating system has *native threads*.

Old Unix-systems usually use user-level threads where the mini-kernel is given by a program library. Also some implementations of Java still use user-level threads (e.g. the *green threads* library).

Modern operating systems usually have native threads. This is the case for OS/2, Windows 95/98/NT/XP/Vista and Mac OS X.

Some operating system use multilevel scheduling in which a number of native threads are used by a thread library to implement user threads. For instance, the Solaris operating system uses *light-weight processes* (native threads) to implement user-level threads.

It may be noticed that Linux has sticked to a single concept: processes, but has allowed these to share resources such that they act as "threads".

3 Pthreads

Pthreads (POSIX threads) is a prescription for the introduction of light-weight processes (*threads*) within the POSIX standardization work for Unix systems. The Pthreads standard was fixed in 1995 as the POSIX.1003.1c standard.

The standard describes a number of system calls that (more or less) must be present in order for an implementation to comply with the standard.

Any implementation must provide *threads* and synchronization of these. As described above, threads are light-weight processes that share an address space. A traditional Unix process may comprise one or more threads.

An elementary presentation of Pthreads may be found in [NBF96]. As a thorough introductions to multi-threaded programming using Pthreads, [KSS96] and [But97] are recommended.

3.1 Thread creation

The code of a thread must be provided by a function P with the C type `void*` P`(void* arg)`. I.e. the thread gets an argument which is a pointer to "something" and like-wise it may return a pointer to "something". It is up to the program to utilize this.

The thread is created and started dynamically by calling

`pthread_create(`t`,`$attr$`,`P`,`arg`)`

where t is (the address of) a *thread handle variable* where an identification of the new thread is put, P is the function that defines the thread, *attr* is a (pointer to) user-definable attributes such as stack size etc., and *arg* is (the address of) an argument that will be passed to P. If one is satisfied with default attribute values, *attr* may be set to `NULL`.

Any call of `pthread_create` generates a unique *thread handle* that must be stored in a variable of the type `pthread_t`. This value may be used to identify the thread in different calls, e.g. to await the termination of a particular thread.

3.2 Termination

A thread terminates when the function P is ended with `return` or calls `pthread_exit`. The return value (the pointer to something) is kept until some other thread retrieves it (see below).

A thread request another thread to terminate by calling `pthread_cancel`. Cancellation in Pthreads is complex and will not be dealt with here. See [But97].

To await the termination of a thread with handle t the following is used:

`pthread_join(`t`,`r`)`

where r is the (address of) a variable where the "result" of the thread can be put. (If `NULL` is used for r, the result is ignored.)

3.3 Example

A program that starts three instances of a function `P` and awaits their termination could take the following form:

```
#include <pthread.h>

void P(void *arg) {
   ...
}

int main(...) {
  pthread_t p1, p2, p3;

  pthread_create(&p1, NULL, P, "Huey");
  pthread_create(&p2, NULL, P, "Dewey");
  pthread_create(&p3, NULL, P, "Louie");

  ...

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);
  pthread_join(p3, NULL);

}
```

4 Java

Java is a simple, object-oriented programming language that was originally developed by Sun Microsystems for programming consumer electronics. Its (original) implementation in the form of translation into a machine-independent byte-code (J-code) to be interpreted by an abstract machine (Java Virtual Machine) made it suited for download and execution by a Web-browser and fitted with a number of useful standard libraries, the language has attracted enormous attention since 1996.

From a concurrent programming view, Java is interesting because it from its very beginning has had an integrated process concept in the form of *threads*. Threads share resources within a Java program, including shared data represented by shared objects. A Java program is initially executed by a main-thread executing the programs `main`-routine.

An introductory description of threads and their synchronization in Java is found in [OW97]. More advanced techniques are covered in [LB00]. The book [Lea97] is concerned with design of concurrent programs in Java.

4.1 Thread creation

Threads are created through *thread objects* belonging to the standard class `Thread`. Each thread object is assigned a "virtual processor" (the thread proper) that can be accessed and controlled through the operations of the thread object. E.g. the execution of a thread belonging to a thread object t is started by calling the start operation of the object: t.start(). Since a thread is uniquely defined through the thread object it is associated with, we often identify the two and talk about the thread t.

The code that describes the thread behaviour is determined through a parameterless procedure named run. Since a procedure cannot be declared by itself in Java, it must be defined in a separate class, and an object of this class must be provided when the thread object is created.

Declaration and start of a thread thus take the following form:

```
class MyThreadCode implements Runnable {

  public void run() {
    ...                // Code of thread
  }
}

Runnable c = new MyThreadCode();
Thread t = new Thread(c);
t.start();
```

Here "implements Runnable" indicates that objects of the class have a run() procedure. First, a code object c is created and passed as a parameter when creating the thread object t. Now the thread object remembers that the code is to be found in the c object such that the call of t.start() starts up a thread that begins executing c.run().

Instead of creating a separate code object, it can be combined with the thread object. This is accomplished by making a specialization[1] of the class Thread with the run-procedure wanted. Declaration and start then become a little simpler:

```
class MyThread extends Thread {

  public void run() {
    ...                // Code of thread
  }
}

Thread t = new MyThread();
t.start();
```

As the thread object is not passed any external code object it will by default use itself as code object such that it becomes t.run() that is being executed as result of t.start().

Since one usually does not need a separate code object, the simpler form will be used in the sequel.

4.2 Termination

A Java thread terminates when its run procedure ends. To await the termination of a thread t, the operation t.join() is provided. This operations may return with an Interrupted-exception (see below) why calls of join must appear within an exception handler.

[1] Really just an overwriting of the Thread-class' given, empty run()-procedure.

A Java program terminates when all user threads[2] have terminated. Thus, the program needs not terminate when the main thread reaches the end of the `main`-routine. Any thread, however, may terminate the program prematurely by calling `System.exit(...)`.

An original operation t`.stop()` for forcing termination of a thread t has now become "deprecated" since it is difficult to use safely. If a thread needs to be stopped, it may instead be *interrupted* by setting a flag it may react to (see section 4.5).

4.3 Examples

It is not possible to pass arguments directly to `run`, but one may declare and set values in the class to which `run` belongs.

Below we see a program that starts three concurrent threads of the same type but with different arguments and awaits their termination:

```
public class C {

  class P extends Thread {
    String name;

    public P(String n) {
      name = n;
    }

    public void run() {
      ...
    }

  public static main(String[] argv) {
    Thread p1 = new P("Huey");
    Thread p2 = new P("Dewey");
    Thread p3 = new P("Louie");

    p1.start();   p2.start();  p3.start();

    ...

    try {p1.join(); } catch (Exception e) {}
    try {p2.join(); } catch (Exception e) {}
    try {p3.join(); } catch (Exception e) {}
  }
}
```

The private variables of a thread can be declared as local variables in `run`. If a thread needs to access shared objects, these must be accessed via global variables or, like parameters, be passed to the object that `run` is associated with. If e.g. two threads `t1` and `t2` are going to use a shared object `x`, it can be implemented in the following way:

[2] A *user thread* is a thread that is not marked as a "daemon" by calling t`.setDaemon(true)`

```
class Shared { ...}

class MyThread extends Thread {

  private Shared s;

  public MyThread (Shared x) {
    s = x;
  }

  void run() {
   ... s.op() ...
  }
}

Shared x = new Shared();           // Create shared object

Thread t1 = new MyThread(x);
Thread t2 = new MyThread(x);
t1.start();  t2.start();
```

Autostart

If it is desired that a thread starts immediately when created, it can be done (as the last thing) in the constructor:

```
class MyThread extends Thread {

  public MyThread () {
    ...
    this.start();
  }

  void run() {
   ...
  }
}

Thread t1 = new MyThread(x);   // Starts itself
Thread t2 = new MyThread(x);   // Starts itself
```

4.4 Thread cancellation

Often in multi-threaded GUI-based applications, activities carried out by dynamically started threads (e.g. file-download) are to be cancelled. Rather than forcing a thread to die using the deprecated *t*`.stop()` method, the thread should be requested to terminate itself using the operations:

t`.interrupt()`	Sets the interrupt mark of the thread t
`Thread.interrupted()`	Tests/resets the interrupt mark of the current thread.

The *interrupt mark* of a thread is a flag that the thread may poll using the `interrupted()` method when convenient (and safe) to do so. Hereby the thread may terminate itself in a more structured way than possible if it was forced to terminate. If the thread is waiting when the mark is set, it will be reactivated with an `InterruptedException`. Similarly, if the flag is set when the threads arrives at a operation that may block the thread, the `InterruptedException` is thrown (and the flag reset).

4.5 Other thread operations

A few other useful thread operations:

`Thread.sleep(`n`)`	Waits for (at least) n milliseconds.
`Thread.currentThread()`	Returns the Thread object of the calling thread

There are also the operations t`.suspend()` and t`.resume()` that may be used to temporarily cease the activity of a thread. However, like the `stop` operation, `suspend` and `resume` have proven so error-prone that their use is no longer recommended.

Thread groups

Within a single Java program, all threads are normally equal. It is, however, possible to gather threads into *thread groups* and give different groups different privilege. One may, for instance, prevent threads of one group to kill threads in another group using `stop`. The thread group concept may thus be seen as a generalization of the process concept of operating systems and it is used, e.g. when executing unreliable *applets* downloaded from the web. For an introduction to thread groups, see [OW97].

5 Win32

Win32 is the common application programming interface (API) to *Windows 95/98* and *Windows NT*.

Win32 amongst other things provide a notion of threads to the application programmer. As above, threads are light-weight processes that share an address space.

The use of Win32 is shown in C, but the operations are also available from e.g. Delphi.

The thread part of Win32 is treated in depth in [Ric95].

5.1 Thread creation

The code of a thread is defined by a function P of the form:

```
DWORD WINAPI P (LPVOID arg) {
  ...
  return(...)
}
```

where DWORD, WINAPI, and LPVOID are predefined Windows types. The procedure receives an argument which is a pointer to "something".

A thread is created by calling

CreateThread(NULL, 0, P, *arg*, *go*, NULL)

where P is the function defining the code, *arg* is (the address of) an argument to be passed to the thread, and *go* is a constant (0 or CREATE_SUSPENDED) that determines whether the thread is to be started immediately, or is initially suspended (see below). The other arguments determines security properties, memory allocation etc. The arguments shown select the (reasonable) default settings.

The result of a CreateThread call is a reference of the type HANDLE that identifies a *kernel thread object* that represents the thread. As in Java, all operation on a thread are performed through its thread object.

5.2 Termination

A thread terminates by when the function returns or explicitly calls ExitThread(r) where r then becomes the result of the thread.

It is possible to force the termination of another thread by calling TerminateThread(t,r) whereby t's return value is set to r. The thread t, however, is given no chance to clean up.

Termination of a thread is awaited by the general operation WaitForSingleObject(o,*timeout*) that awaits that a system object o is "ready". For a thread object this means that the termination is awaited.

When a thread has terminated, its result may be obtained with the GetExitCodeThread operation.

5.3 Example

Using the Win32 system calls, the common example becomes:

```
DWORD WINAPI  P(LPVOID arg) {
   ...
}

int main(...) {
  HANDLE p1, p2, p3;

  p1 = CreateThread(NULL, 0, P, "Huey" , 0, NULL);
  p2 = CreateThread(NULL, 0, P, "Dewey", 0, NULL);
  p3 = CreateThread(NULL, 0, P, "Louie", 0, NULL);

  ...

  WaitforSingleObject(p1,0);
  WaitforSingleObject(p1,0);
  WaitforSingleObject(p1,0);

}
```

Suspension

As in Java, threads may *supend* and *resume* each other, but contrary to Java, it is registered how many times a thread has been suspended simultaneously.

Suspension of threads is deadlock-prone and should be used carefully.

6 .NET and C#

The so-called .NET platform is an execution environment developed by MicroSoft corporation. Like the JVM (Java virtual machine), its major component is a virtual machine executing a byte-code instruction language. The virtual machine is known as CLR (Common Language Runtime) and the byte-code as CIL (Common Intermediate Language).

Whereas the Java byte-code and the JVM was designed specifically as a target for the Java high-level language, CIL has been developed to be the target code of many different programming languages and parts written in different languages can be combined into a single program. On the other hand, the object-oriented language C# was developed to address all the functionalities of the .NET platform and is considered the language of choice for most .NET applications. C#has many similarities with Java.

Another major component of the platform is the .NET Framework Class Library providing lot of functionality, dealing especially with interfacing to networks, web-servers and databases.

Major parts of .NET platform specification have been standardized within the ECMA and ISO organizations [Ecm02a, Ecm02b] and based on this, alternative implementations for non-MicroSoft platforms have been developed, see e.g. [mon].

As in Java, concurrency is an inherent notion within the .NET framework as well as the C# language. Since especially the concurrency part of .NET seems strongly inspired by Java, we here chose present the .NET thread notions by the way they appear in C#, compared to Java.

6.1 Thread creation and termination

A C#program comprises a number of concurrent threads executing within a shared execution context.

The built-in classes for C#that support concurrency are found in the *namespace* `System.Threading` *namespace*[3]

In Java, the code to be executed by a thread is defined by a method called run() within some object and a reference to this object is passed to the thread-object. In C# a new notion of delegates is used in a similar way.

In general a *delegate* is a dedicated object that refers to a method in some other object. The actual reference is bound when the delegate object is created.[4] Each threads is represented by an instance of the `Thread` class. When the thread object is created, it is passed a reference to a delegate instance of the type `ThreadStart` that again points to a parameterless method in some

[3] A namespace corresponds to a Java package

[4] Compatible delegates may actually be combined, but that would make little sense for thread bodies.

object. By calling the `Start()` method on the thread object, a virtual processor is allocated and starts executing the method pointed to by the delegate.

A typical creation of a thread in C#could look like:

```
class MyThreadCode  {

  public void WorkToBeDone() {
    ...                // Code of thread
  }
}

MyThreadCode c = new MyThreadCode({parameters});
Thread t = new Thread(new ThreadStart(c.WorkToBeDone));
t.Start();
```

Here `new ThreadStart(c.WorkToBeDone)` creates a delegate object pointing to `WorkToBeDone` in object `c`. The effect is that the thread `t` will start executing the `WorkToBeDone` concurrently with the main thread.

6.2 Thread control

The life cycle of a C# thread resembles that of a Java thread. Initially, the thread is *unstarted*. When started by the `Start()` method, it becomes *running* (no distinction is made between being ready to execute or actually executing). From the running state, it may enter the a *waiting* state, e.g. by calling `Sleep()` or `Join()`. When the threads reaches the end of its thread body, it enters the *stopped* state.

A thread may be *suspended* and *resumed*. However, as in Java, this should only be used for scheduling experiments and not as a synchronization method.

As in Java, a thread may be cancelled in two ways. The `Abort()` method will stop a thread immediately, throwing an `ThreadAbortException`. This should be used with utmost care in order to clean up the effects of the thread. Similar to Java, a more controlled way of stopping a thread is by calling `Interrupt()`. If the thread is in the waiting state, an `InterruptedExecption` will be thrown. Otherwise, an interrupt mark is set and the exception will be throw whenever the thread performs an operation that may wait.

Threads also have operations for setting thread priorities and operations that may control low-level access to shared variables.

The mechanism specific for synchronization are covered in [Løv04].

6.3 Other thread related notions

The need for isolating parts of a program due to security or robustness issues (like running an applet within a browser) was in Java addressed by the notion of thread groups. In .NET a similar notion is that of *application domains*. Usually a program runs in a single application domain, but may create new ones and load these with new parts of the program (using a unit of code known as *assemblies*, typically a single file). Various protection parameters may be set up for each application domain.

7 Ada95

Ada95 is a revision of the language Ada that was originally defined in 1980 by the US Department of Defense for use within the US Military Systems.

Ada has an integrated process notion called *tasks*. An Ada-program may create a number of tasks to be executed concurrently and share the resources of the program. Thus, tasks correspond to threads in the other languages.

The good description and discussion of Ada's concurrent programming facilities is found in [BW95].

7.1 Task creation and termination

As opposed to the languages described above, creation and termination of Ada tasks is linked to the structure of the program. Task types in Ada are declared by providing a specification and a body. A task specification defines the services the task provides to others (see [BW95] or [And00]). The body defines the code to be executed.

Instances of a task type T are declared as variables of type T. For any block consisting of declarations and block body, the tasks declared will be created as the declarations are elaborated. When the block body is reached (`begin`), all created tasks are started and execute concurrently with the block body. When the block body has ended (`end`) the termination of all created tasks is awaited before the block terminates.

7.2 Example

And Ada program that creates three concurrent tasks of the type `T` and awaits their termination may have the following form:

```
procedure Prog is

  task type T is
    entry E(...
  end T;

  task body T is
    ...
  begin
    ...
  end T;


  t1, t2, t3: T;          // Here t1, t2, and t3 are created

begin                     // Here t1, t2, and t3 are started

  ...

end;                      // Awaits the termination of t1, t2, and t3
```

It is not possible to pass parameters to Ada tasks before they are started. Instead a task could read (accept) parameters as the first thing in its body.

If only a single instance of a task type is needed, the `type` keyword may be dropped in the specification resulting in the creation of exactly one instance.

It is also possible to create tasks dynamically using pointers to tasks. This works almost as in Pthreads/Java in that a call of `new` T dynamically creates an instance of task type T, starts the task, and returns a unique reference to the task.

8 Erlang

Erlang is a programming langauge developed by the Ericsson company for use in their telecommunication equipment. The language has been designed with emphasis on simplicity, concurrency and robustness. In recent years, the language has received renewed interest as a canditate for scalable computing.

Erlang is basically a dynamically typed *functional language* enhanced with *concurrency* and *message passing*. This implies that activities are programmed by functions that typically call themselves recursively in order to provide an indefinite behaviour. Functions may be executed (or strictly evaluated) as an independent activity by *spawning* a new process to do the evaluation. Each spawned processes generates a unique *process identifier* which works as a handle for directing messages to this process (se other parts of these notes) as well as for setting up an exit-handler for the process.

Erlang processes are comparable to threads, but are typically even more leight-weight. They are implemented as user-level threads but using several underlying native threads, modern multiprocessors may be utilized. Also, a system of Erlang processes may be distributed over several computation nodes enabling simple scaling of programs.

The identifiers of processes may be stored in a kind of *registry* such that they may be looked up by name. [This shared datastructure is somewhat in conflict with the rest of the philosophy though.]

The language itself has extended with several libraries including *Open Telecom Platform (OTP)* which provides interfacing to networks, databases etc.

Information about Erlang can be found at `erlang.org`. An introduction to programming in Erlang can be found in [Arm07].

Many of the ideas of Erlang can also be found in the recent **Scala** langauge which combines object orientation with functional programming. For information about Scala, see `scala-lang.org`.

8.1 Example

Below we show an Erlang program which creates three subprocesses and await their termination.

```
-module(donald).
-export([main/0]).

p(Name,Uncle) ->
    ...
    Uncle ! {self(),'DONE'}.

main() ->
    Me = self(),
    A = spawn(fun () -> p("Hewey", Me) end),
    B = spawn(fun () -> p("Dewey", Me) end),
    C = spawn(fun () -> p("Louie", Me) end),

    receive {A,'DONE'} -> {} end,
    receive {B,'DONE'} -> {} end,
    receive {C,'DONE'} -> {} end.
```

Here, three instances of the function P are spawned as processes which will run concurrently with the main funtion. Erlang has no direct means of awaiting process termination[5], so here this is accomplished by explicit sending of termination messages from the sub-processes. The details of the message passing will be covered in other parts of these notes.

9 Other languages

There are a number of other languages for which processes/threads are integrated concepts. Among the historically most prominent ones, we mention:

Concurrent Pascal was the Danish operating systems pioneer Per Brinch Hansen's proposal for at language with integrated concurrency notions. The communication is based upon monitors. Long time before DOS/Windows he demonstrated how a single-user operating systems could be written almost entirely in Concurrent Pascal.

Concurrent Pascal has been in industrial use, but now it is only of historical interest.

Modula-3 was developed by the former company Digital and Olivetti for the purpose of structured systems programming (i.e. as an alternative to C). The syntax is (as in Modula-2) Pascal-like. The language has a notion of threads and syntactically supports a Ptheads-like monitor concept. The language has been in industrial use, but lost to the emergence of Pthreads and Java.

Concurrent ML is an ML library that extends the functional language ML with a thread concept. Threads communicate through synchronous communication.

References

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[5]Erlang does have a notion of *links* which can be used to generate termination messages automatically upon process exit. See the Erlang documentation for details

[Arm07] *Programming Erlang — Software for a Concurrent World*. The Pragmatic Bookshelf, 2007.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

[BW95] Alan Burns and Andy Wellings. *Concurrency in Ada95*. Cambridge University Press, 1995.

[Ecm02a] Ecma. *$C^{\#}$ Language Specification*, 2002. ECMA-334.
`http://www.ecma-international.org/publications/standards/Ecma-334.htm`.

[Ecm02b] Ecma. *Common Language Infrastructure (CLI)*, 2002. ECMA-335.
`http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[KSS96] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads*. Sunsoft Press, 1996.

[LB00] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Java Threads*. The Sun Microsystems Press, 2000.

[Lea97] Dough Lea. *Concurrent Programming in Java: Design Principles and Patterns (2nd ed.)*. Addison-Wesley, 1997.

[Løv04] Hans Henrik Løvengreen. Synchronization mechanisms. Course notes, IMM, DTU, 2004. Version 1.2.

[mon] The mono project. URL: `www.mono-project.com`.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Ptreads Programming*. O'Reilly, 1996.

[OW97] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, 1997.

[Ric95] Jeffrey Richter. *Advanced Windows*. Microsoft Press, 1995.