Mathematics and Computer Science
Technical University of Denmark
Building 303B
DK–2800 Lyngby
Denmark

HHL 02–09–2024

Hans Henrik Løvengreen:

CONCURRENT PROGRAMMING PRACTICE

Processes, Threads and Tasks

Version 2.0 (preliminary v.3)

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

[In this preliminary version, thread pool options have been eloborated for Java only.]

Contents

1 Introduction

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

> A *process* is a distinguished independent activity.

Processes are assumed to take place over time and different processes are *concurrent* if they may overlap in time.

In a computer system, a process is generally associated with the activity of executing a program or part of a progam. We usually just talk about the program part as "the process" and think of it as being executed by its own *abstract processor*[1].

This general, abstract model of concurrent activities may be illutrated as in Figure 1. Here the rounded boxes denote the delineated program parts, the *X*-box indicates that processes may share state and the arrows indicate that processes may communicate.

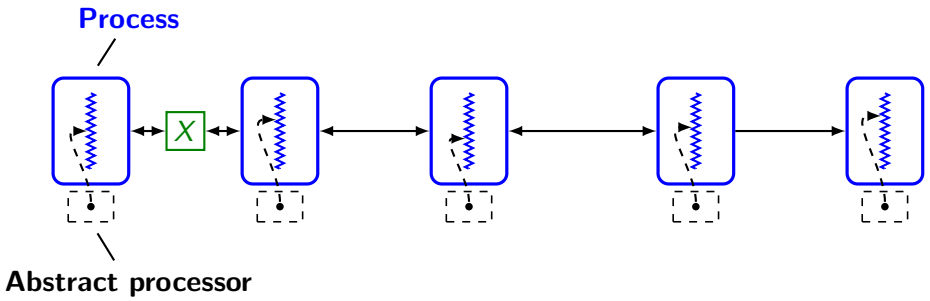

Figure 1: Concurrent processes

In this note, we shall see how this abstract model of concurrent execution is implemented in an number of current languages and operating systems.

In a concurrent system, it must at least at the outermost level be possible to establish two or more concurrent activities. In some programming languages, this may be done by statically declaring a number of activities before the programs starts. Typically, however, in most current languages a *main activity* may *fork* into sub-activities or to *spawn* (create) new activitivies dynamically.

As a common example it is shown how to make a program that creates three concurrent sub-activities and awaits their termination.

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

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 resilint to exceptions etc. Finally, it should be noted that not all examples have been checked syntactically not to mention being run.

[1] Could also be called a *logical* or *virtual processor*, but these terms are used at the hardware level.

2 Threads versus processes

In many languages and operating systems *processes* and *threads* are separate notions. Below, we will elaborate on this.

In the early development of operating systems the *multiprogramming technique* was introduced to switch the the execution among different sequential user programs in order to utilize the system resources better. While one program would be waiting for i/o, another program could be executed on the CPU. By this techniques, the user programs were executed largely independently and overlapping in time. Hence the execution of a program would correspond to the general notion of a process and the simultaneous execution of multiple programs would correspond to concurrent processes. Thus, a "program in execution" became known as a "process", and it became a central task of the operating system to schedule the CPU (and later multiple CPUs/cores) among the processes.

With several user programs running simultaneously, another important task of the operating system became to protect the programs against each other since they could not be assumed to cooperate—rather on the contrary! Apart from representing an independent activity, the process notion was extended to encompass control of resources such as memory and files. As a consequence, within operating system terminology, the process concept has also become an administrative entity and hence a "heavy" notion.

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

Over the 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 have demanded more traditional operating systems facilities (file systems, user interfaces, etc.).

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

Process (operating system notion)
: A process is administrative unit encapsulating an activity, typically the execution of a user program (application, app). 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. Processes may interact by means of operating systems provided *inter process communication (IPC)* mechanisms.

Thread
: A thread is an independent sequential activity within the context of an operating system 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. Especially, threads share the (virtual) address space of the process and hence may efficiently share data with each other. Normally, all threads may act externally on behalf of the process.

From the above, we see that "thread" is the notion that corresponds best to the general understanding of a process. This gives rise to an unfortunate language confusion that we have to live with and be aware of.

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. Threads implemented this way are 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. Threads implemented by the operating systems are often called *operating systems threads* (*OS threads*), *native threads* or *kernel threads*.

Older Unix-systems used user-level threads where the mini-kernel was given by a program library. Also some early implementations of Java used user-level threads (e.g. the *green threads* library).

With OS/2 being one of the first non-academic operating systems supporting OS threads and shortly followed by Windows 95, OS threads are now standard in modern operating systems. OS threads enable the operating system to assign several physical processing entities (processors, cores) to work simultaneously on activities within the same process.

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

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

For a more thorough discussion of processes and threads, you may consult a standard operating systems text such as [TB24].

By today's standards, the term *thread* is usually understood to represent an **OS thread**.

A characteristics of OS threads is that operations that must await certain conditions to be true (e.g. availability of input data) will *block* the execution of the thread until the conditions are satisfied.

2.2 Beyond threads

One concurrency paradigm is to allocate a dedicated thread for each larger piece of work to be executed. E.g. a server may apply a *thread-per-request* model for request made by independent clients.

Although threads are lighter than processes, both user-level and OS threads require some resources. In particular, due to the sequential nature of thread code, each thread needs a *stack* on which to store the *activation records* (parameters and return address) and *local variables* of currently active procedure/method calls. In most systems, the worst-case stack size must be fully allocated before the thread is started, and many run-time systems by default allocate 1 MB of memory for each thread stack. This limits the numbers of such threads to the order of thousands on current desktop computers.

Certain applications, eg. high-end servers and telecommunication systems, may demand a much higher number of concurrent activities. E.g. a web-server may have to service ten-thousands of overlapping requests, waiting for contents to be fetched from databases or other servers. For such systems a simple *thread-per-request* model does not scale well.

Therefore, during the last decade, some systems and language run-time systems have introduced an even finer and more light-weight notion of concurrent activity. This has basically been accomplished in two ways:

Tasks and threadpools
: Separating the work from the concurrent entity executing it.

Light-weight concurrency
: Providing more light-weight thread notions.

These approaches are elaborated below.

2.2.1 Tasks and threadpools

Rather than creating a thread for each small piece of concurrent activity (like a database query), the actual work may be separated from the entity executing it.

This idea is present in the notion of *thread pools* where a limited number of *worker threads* repeatedly execute a number of smaller work items usually known as *tasks*[2].

Typically, a task is defined as a *closure* (i.e. a function with a context) represented by a *task object*. Such tasks may be *submitted* to (executed on, started on) a thread pool forming a *task queue*. Each thread of the thread pool will repeatedly pick a task from the task queue and execute it by calling the associated closure function.

The number of worker threads may be statically determined upon thread pool creation or dynamically adjusted according to various parameters.

There may be many options for controlling the task queue as well as the thread pool. E.g. threads may be created by the thread pool machinery on demand up to a certain limit. Or the task queue may be of limited size. Also the ordering of the queue may be controllable.

For generality, we in this note take the abstract view that a thread pool may internally decide on the queue organization as well the actual number of worker threads; and further that any number of tasks may be submitted to the pool. Therefore, any two tasks submitted to the thread pool may be executed concurrently. Thus tasks should be seen as a *fine-grained concurrency notion*.

Often, the effect or result of a task is needed by other activities. In some thread pool implementations, the result of a task is represented by a special object, typically known as a *future* (or *promise*) returned when submitting a task. The result may then be awaited by operations typically named *get*() or *join*(). In other implementations, the idea of a future result is built into the task notion itself.

[2]Beware that some languages (e.g. Ada) use the term *task* in the same sense as a thread. Also, some operating systems (e.g. Linux) internally use task to denote a kernel thread.

Also the future mechanism (whether explicit or built-in) may provide means for *cancelling* the task. If the task is still in the task queue, this amounts to removing it, remembering its cancelled status. If the task has been selected for execution, the implementation may try let the execution recognize the cancellation wish, e.g. by setting a flag to be interrogated by the task code or by using a language based mechanism like interrupts.

The notions of tasks and thread pools have been included as libraries in many languages, e.g. Java and C#.

Structured concurrency

Often a computation may naturally be described as a number simple tasks that must be executed in a certain order where a sub-task may be dependent on the result of one (or more) predecessor tasks. One example is a nested *fork-join* pattern. To support such *structured concurrency*, more advanced thread pools may allow *continuation tasks* to be attached to existing tasks. Continuation tasks will typically be given by (lambda) functions which will become eligible for execution when the predecessor task has finished with a result which is then passed as argument to the function.

To ease the usage of continuation tasks, the C# language has introduced a special `async/await` syntax making the chaining of tasks become similar to normal, synchronous, sequential composition. This syntax has later been incorporated into other languages.

2.2.2 Light-weight concurrency

Rather than separating work items from execution entities some languages have recently introduced notions where work items are considered concurrent entities by themselves. E.g. a function may be defined as a "mini-process" to be executed concurrently.

Examples of such notions are *go-routines* in Go, *co-routines* in Kotlin and *virtual threads* in Java.

Although these notions are often implemented by tasks executed on thread-pools, the programmer needs not be aware of this, but may merely think of these as small concurrent processes in the general sense.

2.3 Example

As we have seen, the notions of processes and threads are used in different senses at various levels of the system. Also the term "thread" is now also applied at the hardware level. In Figure 2, we give an example illustrating the various notions.

The figure shows two programs being executed on a hyper-threaded computer consisting of one core with two hardware threads (aka. hyper-threads or logical processors) each capable of executing its own independent sequence of instructions, but sharing certain execution units of the core, especially the arithmetic/logical unit (ALU) with each-other.

Each program is executed by an operating system process. The processes may communicate via inter-process communication (IPC) (e.g. pipes). In the left process three (program) threads have

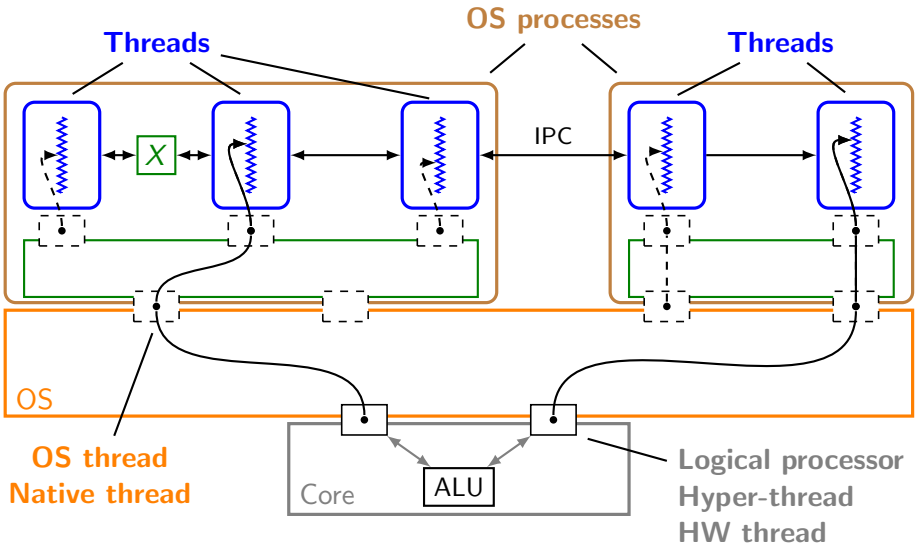

Figure 2: Process and thread notions

been created while the right process comprises two threads. In the right process, the program threads are implemented one-to-one by OS threads, whereas in the left process, the program are being scheduled on two OS threads using a run-time library.

The four operating system threads are being scheduled by switching the executions of the two logical processor among the four OS threads. In the situation shown, the two logical processors execute threads in different processes.

As seen, term "thread" should be used carefully. In these notes, the term will always refer to the thread notion of the given programming language. In most cases, this will be implemented one-to-one by an operating system thread. If not, this will be emphasized.

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 an object-oriented language originally developmed by Sun Microsystems for programming consumer electronics. The language, however, has since gained widespread use as a general programming language.

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]. Advanced usage of Java concurrency is covered extensively in [GPB+06].

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[3].

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[4] of the class `Thread` with the `run`-procedure wanted. Declaration and start then become a little simpler:

[3] As of Java 1.8, a lambda expression may be used for the `Runnable` object

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

```
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.

Thread implemenation

Whereas the first Java Virtual Machines provided a user-space thread library (known as *green threads*), it is now understood that a Java thread is associated with an underlying native thread.

4.2 Termination

A Java thread terminates (dies) 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.

A Java program terminates when all user threads[5] 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 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).

Thread life-cycle

A Java `Thread` goes through a number for states (or phases) during its existence. When created, the thread is non-terminated and non-alive. When started, it becomes *alive*.

During the alive phase, the thread may be *ready* to execute, *running* or *blocked*.

Once terminated (by returning from `run()`), the thread is no longer alive and will remain so as a thread cannot be restarted.

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

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:

```
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();
```

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.

4.6 Thread Pools

Since version 5 of the Java platform, it has offered various forms of thread pools.

In Java, an `Executor` is any class with an `execute(`r`)` method, where r is a `Runnable` object.

An `ExecutorService` is an extension of `Executor` corresponding to a standard thread pool notion. Tasks for execution on the task pool are provided by the basic types `Runnable` with a method `run()` not returning any result or `Callable<V>` with a method `call()` returning a value of type `V`. Objects of these types may be provided by lambda expressions.

Tasks are submitted to a thread pool *tp* using:

tp`.submit(`t`)`	Submits a `Runnable` r for background execution
tp`.submit(`c`)`	Submits a `Callable<V>` c for background execution

In both cases, the submission returns a `Future` object associated with the task and collecting its result. A `Future` object f has operations:

`f.get()`	Awaits and returns the result of the task
`f.get(`t, u`)`	Do. but with time-out after t units u of time
`cancel()`	Attempts to cancel the task

The `cancel()` method makes an attempt of cancelling the task, but if the task has been dispatched for execution by a worker thread, this may not be feasible.

Java provides no default or built-in thread pool. Hence Java programs must create and use thread pools explicitly. For this, the class `ThreadPoolExecutor` may be used. It is a very general thread pool implementation controlled by a number of parameters. Eg. a call of the constructor `ThreadPoolExecutor(`n, m, t, u, q`)` creates a thread pool where:

n	default number of threads (known as *core threads*)
m	maximum number of threads
t, u	idle threads are reclaimed after t units u of time
q	task queue to use

The class `Executors` provides a number of static factory methods for standard thread pools. E.g.

```
Executors.newFixedThreadPool(n)
Executors.newCachedThreadPool()
Executors.newSingleThreadExecutor()
```

A thread pool created by `newFixedThreadPool(`n`)` allocates a fixed number n of worker threads for the execution. A `newCachedThreadPool()` allocates threads as needed by tasks, but let them linger for while when done, waiting for new tasks.

An executor created by `newSingleThreadExecutor()` allocates one (and only one) worker thread. This, however, does not correspond to a thread pool proper, but rather resembles the notion of a *server* which consistently maintains its own state.

Shutdown

A task pool tp provides methods for shutting it down, in case it is no longer needed:

tp`.shutdown()`	Reject further tasks
tp`.shutdownNow()`	Do. and tries to cancel task under execution
tp`.awaitTermination(`t, u`)`	Awaits that all task have completed (with timeout)

However, in many use cases, thread pools are supposed to last until the program is closed (where the shutdown methods may be used).

Scheduled Execution

In addition to immedate task execution, a *scheduled thread pool* allows task to be submitted for execution at a later time or to be repeatedly submitted for execution. E.g. a scheduled thread pool *stp* created by `Executors.newScheduledThreadPool(`n`)` provides methods like:

stp`.schedule(`r, t, u`)`	Schedules the runnable r to enter the task queue after t units u of time
stp`.scheduleAtFixedRate(`r, t, p, u`)`	Schedules the runnable r to enter the task queue after t units u of time and then repeatedly every p units u of time

Fork/Join Pool

In the standard `ExecutorService` tasks are submitted to the thread pool by external threads for immediate or scheduled execution. Hence, if one tasks needs the result of another task, the result will have to be awaited by an external thread (e.g. the main thread) which will then create a task using this result and submit that to the thread pool.

Many computations naturally follow a *fork/join* pattern, where a computation splits up into sub-computations whose results are again combined. This splitting may be done recursively. Such computations do not fit the basic thread pool pattern well. Although a task may create sub-tasks, awaiting these with the `get()` method blocks the worker thread and effectively oppose the idea of using thread pools.

Since version 7, however, the Java platform has provided a framework supporting fork/join computations. It comprises a specialized thread pool called a `ForkJoinPool` used by special tasks of the type `ForkJoinTask`. These tasks provide some dedicated execution methods:

t`.compute()`	The computation function of the task
t`.fork()`	Submits the task to the current fork/join pool
t`.join()`	Awaits and returns the result of the task

Typically fork/join tasks are recursive: In the `compute()` method, sub-tasks are created and submitted using sub_i`.fork()` and then their results awaited using sub_i`.join()`.

Most fork/join computations are run on a system provided default fork/join pool obtained with `ForkJoinPool.commonPool()`. This pool uses *work stealing queues* — one for each worker thread. When a (fork/join) task creates new tasks submitted to thread pool using `fork()`, these are inserted into the queue of the current thread using a last-in-first-out principle. This way the thread will work on tasks which are closely related leading to improved performance. If a task queue is exhausted, tasks from be "stolen" from the queues of other threads.

If a task needs the result of a forked sub-task t, it will call t`.join()`[6]. If the sub-task has not yet been dispatched for execution, the calling worker thread may execute it itself. This way results from sub-tasks may be efficiently transferred to parent tasks without the need of external threads.

[6]Notice that this method is similar to, but different from, the `join()` method on threads.

4.7 Virtual Threads

As described in Section 2.2, thread pools were to tackle situations where the thread-per-request model would not scale well.

Rather than adopting the `async/await` syntax for coping with massively concurrent systems, the Java language has an ongoing project, called Loom, which aims at defining a much more light-weight concurrency notion known as *virtual threads*.

A virtual thread should be thought of as a standard Java thread, but is not implemented one-to-one by a native thread. Instead, the thread uses a light-weight implementation which allows for millions of threads to be created. Thus, with virtual threads, the thread-per-request model becomes feasible and efficient.

Currently, virtual threads are only available as an early-access pre-release build of Java 19. For futher details, see [Loo].

5 .NET and C#

The .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 chosen to present the .NET thread notions by the way they appear in C#, compared to Java.

5.1 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*[7]

[7] A namespace corresponds to a Java package

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.[8] 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 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.

5.2 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 [Løv16].

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

5.3 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.

6 C^{++}

Althouth C^{++} has been standardized since 1998, concurrency has not been part of this. Instead, users of C^{++} have relied on platform specific libraries like Pthreads for concurrent programming or used cross-platform libraries like Boost or Qt.

With the C^{++}-11 standard of 2011, the language has become concurrency aware in a platform independent way.

A thorough presentation of the concurrency related aspects of the C^{++}-11 standard is presented in [Wil12] including a large number of application examples.

6.1 C^{++} Threads

Threads in C^{++} are created and controlled via objects of the `std::thread` class found in the `<thread>` library. A thread is dynamically created by declaring a thread object, passing a *callable* item in the constructor. The callable item may be a named function, a *lambda function* or an object with a callable operator. Parameters to the callable may be passed as well. The effect is that a thread is *immediately started* executing the code of the callable item and this thread becomes *owned* by the thread object.

Indirectly the thread that declared the thread object becomes the owner of newly started thread. The owner may await termination of the new thread by calling `t.join()` on the thread object. Once joined, however, the association between the thread object and the thread disappears and subsequent calls of `t.join()` will result in run-time errors.

Beware that although the thread notion superfically may resemble that of other languages like Java and C$^{\#}$, the C^{++} threading model is more complex and there are a number of subtle differences that one should observe. Eg. that a thread may be joined only once. Consult [Wil12] for a comprehensive presentation of these issues.

6.2 Example

Our standard example may be implemented by:

```
#include <string>
#include <threads>

void p(string &name) {
```

```
   ...
}

int main(...) {
  std::thread p1 (p, "Huey");
  std::thread p2 (p, "Dewey");
  std::thread p3 (p, "Louie");

  p1.join();
  p2.join();
  p3.join();
}
```

7 Python

Python is a popular interpreted programming language. It is characterized by having a very dynamic type system and a large number of libraries. The language is well documented at its web site [Pyt].

The concurrency notions have evolved over many years. The current framework for concurrent execution is the module `threading` which provides a notion of threads, closely following the Java thread model.

Thus, as in Java, a thread may be created by extending the `Thread` class and overwriting a `run()` with the code to be executed concurrently. The concurrent execution of an instance of this class is then started by calling a `start()` method on the thread object. Also it is possible to *join* with a thread, ie. awaiting its termination.

The standard implementation of Python, CPython, maps Python threads one-to-one with underlying operating system threads. However, due to the interpreted execution, access to the common interpreter data structures (including object reference counts) are confined to a *global critical region* controlled by the so-called *global interpreter lock (GIL)*. This implies that the execution cannot exploit an underlying multiprocessor (multicore) architecture.

In order for a CPU bound Python program to run faster on a parallel architecture, other libraries must be used, eg. the `multiprocessing` module which enables Python execution to take place using several operating system processes. The interface to the `Process` class has carefully been defined to mimic that of the `Thread` class.

7.1 Example

The standard example follows the Java structure, but using a simpler syntax:

```
import threading

class P (threading.Thread):

    def __init__(self, name):
        threading.Thread.__init__(self)
```

```
        self.name = name

    def run(self):
      ...


p1 = P("Huey!")
p2 = P("Dewey!")
p3 = P("Louie!")

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

p1.join()
p2.join()
p3.join()
```

8 Erlang

Erlang is a programming language 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 candidate 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 (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 (see other parts of these notes) as well as for setting up an exit-handler for the process. This closely follows the general *actor model* of concurrency.

Erlang processes are comparable to threads, but are typically even more light-weight. They are implemented at user-level by scheduling one or more underlying operating system threads. 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 data structure 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].

8.1 Example

Below we show an Erlang program which creates three sub-processes 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.
```

In this example, three instances of the function P are spawned as processes which will run concurrently with the main function. Erlang has no direct means of awaiting process termination[9], 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 Scala

Scala is new language on the scene offering a combination of functional and object oriented programming.

With roots in the Java world, it provides a syntactically simple, extensible language which may be compiled to the Java Virtual Machine and seamlessly run together with Java libraries.

The core Scala language does not define a built-in notion of concurrency. It is possible to access the underlying Java thread model with shared state, but this is considered error-prone. However, it is possible to make library-based language extensions and several of such ones have been developed for concurrent programming (see below).

Information about the Scala language can be found at `scala-lang.org`. For an introduction, covering the Actor library, see [OSV08]

9.1 The Scala Actors Library

The Scala Actors library is an attempt of providing the notion of communicating actors as a concurrency programming model. The library is heavily inspired by the built-in Erlang concurrency notions.

As in Erlang, sub-processes (called *actors*) are spawned and their identity stored. This may then be used to send messages to private mailbox of the actor. The actor may selectively receive and handle messages from its mailbox.

[9] 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

There are several notions of message passing available. These, however, are covered in a separate note.

Normally actors are implemented one-to-one by underlying OS threads. However, for certain constrained actor behaviour (known as *reactions*), they may be scheduled efficiently at user-level using a limited number of underlying OS threads.

The standard implementation of actors does not allow communication to extend a single Java Virtual Machine. However, a recent library called *Akka* addresses the issues of actors communicating transparently over a network of distributed JVMs.

9.2 Example

Below we show a Scala program which creates three processes and await their termination.

```
import scala.actors.Actor

case class Done(a: Actor)

class Nephew(name: String, uncle: Actor)  extends Actor {
            def act() = {
                ...
                uncle ! Done(this)
        }
}

class Uncle extends Actor {
        def act() = {
            val A = new Nephew("Hewey",this)
            val B = new Nephew("Dewey",this)
            val C = new Nephew("Louie",this)
            A.start();  B.start();  C.start()

            receive {case Done(A) =>}
            receive {case Done(B) =>}
            receive {case Done(C) =>}
        }
}

object Donald {
    def main(args: Array[String])= {
            val donald = new Uncle().start
    }
}
```

Here each sub-process is represented by the `act()` method of an `Actor` class. Much like in Java, this is instantiated and started in order to run the process concurrently.

As in Erlang, there is less need for joining with a started process, since it should not modify shared state. However, as in the Erlang example, we may emulate a join by waiting for a dedicated termination message from the sub-process. Here, the message is of the form `Done(`*pid*`)` where *pid* is the identity of the sub-process. This message types must be declared as a (case) class. Finally, a global `Donald` object (corresponding to a static class) acts as the program entry actvating the first actor.

10 Rust

Rust a recent language which tries to address memory and concurrency issues such as dangling pointers and unprotected concurrent access to shared data through its type system.

The languages combines functional and imperative paradigms with elements of object-oriented programming using a c-like concrete syntax. Is is a compiled language aiming at being the langauge of choice for efficient systems programming (replacing C/C++).

Rust is well documented on the language home page [Rus] although the language itself it still in a stabilizing phase. The initial development of the language was done under the auspices of the Mozilla project.

For concurrency, Rust offers a simple version of traditional *threads*. A piece of sequential Rust code in the form of a parameterless function may be executed by its own thread by passing it to the `spawn` function of the `thread` package. The result of this is a handle to the *result* the thread may yield. This is like a *future* which may be awaited and retrieved by the method *t*`.join()`.

Unlike Java, a Rust program terminates whenever the main thread terminates. Currently there are no means of cancelling a thread.

The interesting part about Rust concurrency is the way data shared among concurrent threads is handled. Using a sophisticated type system, the language tries to avoid common issues like race conditions. This aspect, however, is covered in the synchronization part of these notes [Løv16].

10.1 Example

Our usual example may be expressed by the following Rust program:

```
use std::thread;

fn p(name: &str) {
    ...
}

fn main() {
    let p1 = thread::spawn(|| p("Huey"));
    let p2 = thread::spawn(|| p("Dewey"));
    let p3 = thread::spawn(|| p("Louie"));

    p1.join().unwrap();
    p2.join().unwrap();
    p3.join().unwrap();
}
```

A few langauge notes:

The `||` in the spawn function is not something indicating parallel execution. It is part of the (dubious) notation for *lambda expressions* (closures) in Rust which generally takes the form `|`*params*`|` *expression* only here, the function has no parameters.

The `unwrap()` method call on the (void) result of join ensures that if the thread ended in a fatal situation (known as *panic*), this will make the main thread panic too. If left out, a compiler warning about unused results will appear.

11 Go

Go is a newer language which attempts to provide limited set of simple notions sufficient efficient systems programming.

The language has high focus on concurrency introducing its own notion of a light-weight thread, called a *goroutine* (a pun on the old term *co-routine*). Goroutines are meant to interact by message passing although shared state is also feasible.

The Go language is documented on the home page [Go]. Even though the development is anchored at Google, the language is anchored at the Google company.

Any function can be started as a concurrent goroutine by prefixing its call with the keyword `go`. Also anonyous function expressions (lambda-expressions) may be started concurrently this way.

Conceptually each goroutine is a light-weight concurrency unit executing independently of each other. Goroutines are typically implemented by tasks scheduled on an pool of OS threads, but the programmer would not be aware of this.

Go does not provide handles to started goroutines, so there is no way to join with a goroutine waiting for its termination. Instead an explicit termination communication protocol may be established or a synchronization mechanism like the provided *WaitGroup* may be used.

11.1 Example

Our usual example may be expressed by the following Go program:

```
package main

import "fmt"
import "sync"

var wg sync.WaitGroup

func p(name string) {
  defer wg.Done()

  for i := 0; i < 10; i++ {
    fmt.Printf("I am %s\", name)
  }
}

func main() {
  wg.Add(3);

  go p("Huey")
  go p("Dewey")
  go p("Louie")

  wg.Wait()
  fmt.Println("Enough, thanks!")
}
```

Here the `go` actions spawns new go-routines concurrently executing the `p` instances. To wait for their termination, a `WaitGroup` instance `wg` initialized to 3 is used. The `defer` construct exeutes the `Done()` operation when the `p` function ends, thereby decrementing the WaitGroup counter by 1. When the counter reaches 0, the main program may pass the `wg.Wait()` operation, terminating the program properly.

12 Other languages

There are numerous 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.

Ada was originally defined in 1980 by the US Department of Defense for use within the US military systems. Later several revsions of the language have appeared (Ada-95, Ada-12, Ada-15).

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].

References

[Arm07] Joe Armstrong. *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`.

[Go] The go programming langauge. `www.golang.org`.

[GPB+06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Dough Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

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

[Loo] The Java Loom project. `https://openjdk.org/jeps/425`.

[Løv16] Hans Henrik Løvengreen. Synchronization mechanisms. Course notes, DTU Compute, 2016. Version 1.5.

[Mon] The mono project. `www.mono-project.com`.

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

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Inc., 2008.

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

[Pyt] Python programming langauge — official website. `www.pythyon.org`.

[Rus] The rust programming langauge. `www.rust-lang.org`.

[TB24] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems (5th ed.)*. Pearson, 2024.

[Wil12] Anthony Williams. *C++ Concurrency in Action. Practical Multithreading*. Manning Publications Co., 2012.