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

HHL 28–09–2006

Hans Henrik Løvengreen:

CONCURRENT PROGRAMMING PRACTICE 2

Synchronization Mechanisms

Version 1.3

In this note we present a number of mechanisms for synchronizing concurrent processes/threads as they appear in current langauages and program libraries. Especially we show how the monitor concept appears or can be implemented.

Contents

1 Introduction

In concurrent programs there will practically always be a need for synchronization of the processes. By synchronization we generally understand any restriction of the execution of the concurrent processes relative to one another.

Especially, synchronization is necessary when resources are shared by the concurrent processes. Typically these resources are shared variables, but may also be other internal or external resources.

All langauges, program libraries, and operating systems that provide concurrency also have to offer some means of synchronization. Especially, it must be possible to establish the three basic synchronization forms:

> **Mutual Exclusion**. It must be possible to establish criticial regions, each consisting of a number of critical sections executed under mutual exclusion.
>
> **Conditional Synchronization**. It must be possible to let processes await the fulfilment of (perhaps complex) conditions established by other processes.
>
> **True Synchronization**. It must be possible to let two or more processes meet. This is also known as *barrier synchronization*.

Often only a single *synchronization primitive* (eg. semaphores) is offered. This will then have to be used for implementation of all three synchronization forms.

As a structured solution for the use of shared variables, Hoare and Brinch Hansen at the start of the 1970'ies proposed the monitor concept.

A *monitor* is a program component that combines *data abstraction*, *atomicity* and *synchronization*. A typical form is a class with implicit mutual exclusion among the operations of the class and with an associated wait queue mechanism.

In the beginning of the 1980'ies the academic interest changed to communicating processes, while practical use of shared variables continued to be handled by simple semaphore-like mechanisms. Thus, during the 1980'ies, the monitor concept was almost forgotten. This changed in the beginning of the 1990'ies, where the concept was revived by being chosen as the main synchronization mechnism in a number of widespread langauges and libraries such as *Phtreads*, *Ada*, *Java* and *C#*.

In this note we are going to illustrate how different synchronization problems can be solved using the synchronization mechanisms of a number of different languages and systems. Especially we show how monitors appear or can be implemented.

It has been attempted to write the code is it would appear in the given system/language. For library functions, C++ is usually used to emphasize data abstraction. Some places, but not always, it has been tried to make the code robust towards exceptions etc. Not every piece of program has been syntactically checked not to mention being tested!

For an introduction to the various languages and creation of processes and threads, see part I of these notes [Løv04].

2 Specification of synchronization problems

Below, we describe a number of synchronization problems in the form of "abstract monitors" where the operations are presented as one or more conditional atomic actions on the monitor state.

2.1 General semaphore

Dijkstra's classical general semaphore is assumed known:

> **monitor** *Sem*;
>
> **var** S : *integer* := 0;
>
> **procedure** P :
> $\langle\, S > 0 \rightarrow S := S - 1 \,\rangle$;
>
> **procedure** V :
> $\langle\, S := S + 1 \,\rangle$;
>
> **end**;

By changing the counter S to a data queue, the semaphore is generalized to a buffer.

2.2 Barrier synchronization (N-synchronization)

This mechanisms is supposed to let N processes meet ($N > 0$). We allow more than N processes to call the mechanism, thus processes should be let pass in groups of N. Also, the mechanism should be prepared for processes released to call the mechanism again.

> **monitor** *Meet*;
>
> **var** OK : *boolean* := *false*;
> K : *integer* := 0;
>
> **procedure** *Sync* :
> $\langle\, \neg OK \rightarrow (K, OK) := (K + 1, K = N - 1) \,\rangle$;
> $\langle\, OK \rightarrow (K, OK) := (K - 1, K > 0) \,\rangle$
>
> **end**;

This specification is rather operational. The first atomic statement lets only the first N calls through. After this, OK will remain true until all N processes have left *Sync* again. Then then next group of N processes allowed to enter etc.

2.3 Reader/writer synchronization

A number of processes read or write some shared data. These *readers* and *writers* are assumed to call *StartRead*/*EndRead* respectively *StartWrite*/*EndWrite* around their read/write-operations. It must be ensured that no writers are active at the same time as other writers or readers.

We here chose to specify a solution that gives priority to writers. The number of active readers are counted in R, W indicates whether a writer is active, and P denotes the number of pending writers.

monitor *ReadWrite*;

  var W : *boolean* := *false*;
    R, P : *integer* := 0;

  procedure *StartRead* :
    $\langle \neg W \wedge P = 0 \rightarrow R := R + 1 \rangle$

  procedure *EndRead* :
    $\langle R := R - 1 \rangle$

  procedure *StartWrite* :
    $\langle P := P + 1 \rangle$
    $\langle R = 0 \wedge \neg W \rightarrow (P, W) := (P - 1, \mathit{true}) \rangle$

  procedure *EndWrite* :
    $\langle W := \mathit{false} \rangle$

end;

3 Hoare Monitors

Hoare's original queue-semantics is is characterized by the fact that a process that is woken up by a signal immediately takes over the right to execute in the monitor and that the signalling process has preference to reenter the monitor. In [And00], this is the *signal and urgent wait* semantics. This semantics gives a very precise control of is is executing in the monitor. The Hoare semantics is present in a few academic langauges like Pascal-Plus and Emerald. In [hoa] you can find a Java package that implements Hoare-monitors.

3.1 General Semaphore

It is exploited that a process that has been woken up reenters the monitor before new ones:

monitor *Sem*;

  var S : *integer*;
    c : *condition*;

  procedure P;
  begin

```
    if S = 0 then wait(c);
    S := S − 1
  end;

  procedure V;
  begin
    S := S + 1;
    signal(c)
  end;

begin
  S := 0;
end;
```

3.2 N-synchronization

Again we exploit the fact that during a cascade wakeup, no new processes can enter the monitor.

```
monitor Meet;

  var K : integer;
      c : condition;                    – Common waiting queue

  procedure Sync;
  begin
    K := K + 1;
    if K < N then wait(c);
    K := K − 1;
    signal(c)
  end;

begin
  K := 0
end;
```

4 Pthreads

Pthreads (POSIX threads) is a prescription for introduction of leight-weight processes (*theads*) under the POSIX standardization work for Unix systems.

As described in Part I, Pthreads allows C routines to be started as concurrent threads.

For synchronization, threads may use a monitor-like critical region (of the type `pthread_mutex_t`) to which condition queues (of the type `pthread_mutex_t`) may be associated. Regions and condition queues must be initialized before they are used. The queue semantics is basically *signal-and-continue*, but with some deviations from the standard semantics described in [And00].

Exclusive access to a critical region m is obtained by calling `pthread_mutex_lock(`m`)` at entry and `pthread_mutex_unlock(`m`)` at exit. There are also versions with timeout. A critical region is owned by the thread that has locked it and *cannot* be released by another thread. Further, a thread *cannot* acquire a region that it has already acquired.

By calling `pthread_cond_wait(`c`,`m`)`, a thread starts waiting on condition queue c atomically with realeasing the region m. By calling `pthread_cond_signal(`c`)`, *at least* one of the waiting threads will be woken up and implicitly start to try to reenter the critical region. By calling `pthread_cond_broadcast(`c`)`, all waiting threads on c will be woken up. These threads participate *on equal terms* with external threads regarding (re)entry to the critial region.

There is no possibility of asking how many threads are waiting on a particular queue, but these can be recorded exlicitly explicitly in monitor variables.

It may be noticed that condition queues in Pthreads are not in a fixed association with a particular monitor, but rather attached to a monitor when the wait takes place. The enables the use of condition queues for dynamically allocated waiting places in cases where one needs detailed control of the the waiting processes. Examples can be found in [KSS96]

An idiosynchrasy in Pthreads is the notion of *spurious wakeups*. In order to enable certain low-level multiprocessor implmentations of the operations, `Phtread_cond_signal(`c`)` is defined to wake up one *or more* threads on c. Even worse, most authors interpret the Pthreads standard such that spurious wakeups may occur *at any time*, even when no thread signals the condition queue. Thus, when designing a monitor, one has to cater for these spurious wakeups on condition queues. This, for instance, makes the programming of N-synchronization rather complicated.

An introduction to Pthreads can be found in [NBF96]. For a very thorough presentation of all aspects of programming with thread libraries, [KSS96] can be recommended. Another good book mentioning a many good programming principles is [But97].

4.1 General Semaphore

Since a waiting thread may be woken up at any time, the `P` operation must check its condition after wait.

```
#include <pthread.h>

class Sem {

  pthread_mutex_t mutex;
  pthread_cond_t queue;

  int S;

public:

  Sem() {
    S = 0;
    pthread_mutex_init(&mutex, null);
    pthread_cond_init(&queue, null);
  }

  void P() {
    pthread_mutex_lock(&mutex);
```

```
    while (S == 0) pthread_cond_wait(&queue,&mutex);
    S-;
    pthread_mutex_unlock(&mutex);
  }

  void V() {
    pthread_mutex_lock(&mutex);
    S++;
    pthread_cond_signal(&queue);
    pthread_mutex_unlock(&mutex);
  }

}
```

Also, a destructor could be added to close down the `mutex` and `queue` in a proper manner.

It should be mentioned that Pthreads goes hand-in-hand with *POSIX Semaphores* that renders it superflous to construct them as monitors.

4.2 N-synchronization

Due to the problem of spurious wakeups, each thread must check after each wait whether the synchronization is about to take place. Hereby, it becomes critical that new threads may enter the monitor during wakeups of the waiting threads. To cope with this, we introduce a "pre-queue" where new, external threads are gathered while a synchronization is taking place. A synchronization is started by setting a variable `L` to indicate how many threads are still to get out the monitor. When all N threads are out of the way, the threads on the pre queue can be allowed to proceed.

```
#include <pthread.h>

class Meet {

  int K, L;

  pthread_mutex_t mutex;
  pthread_cond_t pre,ok;

public:

  Meet() {
    K = 0;
    L = 0;
    pthread_mutex_init(&mutex, null);
    pthread_cond_init(&ok, null);
    pthread_cond_init(&pre, null);
  }

  void Sync() {
    pthread_mutex_lock(&mutex);
    while (L > 0) pthread_cond_wait(&pre,&mutex);
    K++;
```

```
        if (K < N)
          while (L == 0) pthread_cond_wait(&ok,&mutex);
        else {
          K = 0;
          L = N;
          pthread_cond_broadcast(&ok);
        }
        L--;
        if (L == 0) pthread_cond_broadcast(&pre);
        pthread_mutex_unlock(&mutex);
      }

    }
```

Alternatively, one may also make a solution where the processes wait on two alternating queues, see [KSS96].

5 Java

In Java, threads are synchronized by syntactically supported critical regions. Every Java object implicitly has an associated *critical region*. A critical section belonging to the region of an object o is established using the `synchronized` construct:

`synchronized (`*o*`) {` critical section `}`

By declaring an operation (method) of a class as `synchronized`, a call of this operation for a given object is automatically executed as a critical section of the region associated with the object. If all the operations of a class are declared as `synchronized`, the objects of the class hereby become monitors.

For conditional waiting within the critical sections, each object's critical region has *one and only one* anonymous *wait queue* (wait set). A thread enters the wait queue by calling the `wait()`-operation of the object[1]. As usual for condition queues, the critical region is released atomically with the entrance to the wait queue.

Threads that wait on an objects wait queue can be woken up by call of the object's `notify()` and `notifyAll()` methods. A call of `notify()` will wake up exactly one arbitrary thread on the wait queue, if any. A call of `notifyAll()` will wake up all threads on the queue. The notifying thread continues without releasing the critical region. The woken threads will have to enter the critical region again before continuing after the `wait()`. These threads participate *on equal terms* with new threads in the race of getting access to the critical region.

The notify operations are *only* to be used within critical sections belonging to an object's critical region.

[1] During the wait, the thread may be "interrupted" by another process resulting in an `InterruptedException`. Therefore this exception must be handled somewhere around the wait. If interruption is not used, it is often caught immediately around the wait and is ignored.

It is also possible to wait with timeout by the call `wait(`*millisec*`)`. A thread that waits using timeout is not informed whether is was woken due to a notify operation or to timeout. Therefore, it has to determine by itself whether a timeout has occurred or not.

It is not possible to ask how many threads are on the wait queue, nor whether it is empty or not. If this information is needed, it has to be recorded in monitor variables.

As of Java 1.5, it is explicitly defined that waits may suffer from *spurious wakeups* as in Pthreads. Therefore, after a wait, a Java thread cannot be sure whether it was woken due to a call of `notify`/`notifyAll`, a timeout (if the wait is timed) or a spurious wakeup. Therefore, the **waiting condition should always be rechecked** in Java monitores.

Note that most Java threading literature (including [And00]), has not yet been adapted to this semantics.

5.1 General Semaphore

Even though spurious wakeups cannot occur in Java, a woken thread may still be overtaken by a new, external thread. Thus, after the wait in the `P`-operation, it is necessary to check if the semaphore value is still positive.

```
class Sem {

  int S;

  public Sem() {
    S = 0;
  }

  public synchronized void P() {
    while (S == 0) try {wait();} catch (InterruptedException e) {}
    S--;
  }

  public void synchronized V() {
    S++;
    notify();
  }
}
```

5.2 Barrier Synchronization

An attempt to utilize the `notifyAll()` operation:

```
class Meet {

  int K;

  public Meet() {
    K = 0;
  }
```

```
  public synchronized void Sync() {
    K++;
    if (K < N)
      try {wait();} catch (InterruptedException e) {}
    else {
      K = 0;
      notifyAll();
  }

}
```

does **not** work. A spurious wakeup may take place at any time and make a tread leave the barrier too early.

Instead, a flag `OK` may be used as in the abstract specification:

```
class Meet {

  int K;
  boolean OK = false;

  public Meet() {
    K = 0;
  }

  public synchronized void Sync() {
    while (OK)
      try {wait();} catch (InterruptedException e) {}
    K++;
    if (K == N) {
      OK = true;
      notifyAll();
    }
    while (!OK)
      try {wait();} catch (InterruptedException e) {}
    K--;
    if (K == 0) {
      OK = false;
      notifyAll();
    }
  }

}
```

Here, a leave phase is initiated by the last thread to arrive and new threads are held back at the start of the method until all threads have left the barrier.

5.3 Reader/writer Synchronzation

When trying to solve the reader/writer problem one encounters the limitations of Java's single wait queue associated with each critical region. In particular, it is not possible to start readers and writers separately. Instead, the following *general scheme* can be used: Each wait condition is represented by a loop that waits as long as the condition is false. After each monitor operation (that could possibly enable one of the wait conditions), `notifyAll()` is performed to ensure that all waiting threads get a chance to evaluate their condition.

For the reader/writer problem, this general scheme results in:

```
class ReaderWriter {

  int pending = 0;                      // pending writers
  int reading = 0;                      // active readers
  boolean writing = false;              // writer active

  public synchronized void StartRead() {
    while (writing || pending > 0)
      try {wait();} catch (InterruptedException e) {}
    reading++;
  }

  public synchronized void EndRead() {
    reading--;
    if (reading == 0)
      notifyAll();
  }

  public synchronized void StartWrite() {
    pending++;
    while (reading > 0 || writing)
      try {wait();} catch (InterruptedException e) {}
    pending--;
    writing = true;
  }

  public synchronized void EndWrite() {
    writing = false;
    notifyAll();
  }

}
```

Since `StartRead` and `StartWrite` do not affect the wait conditions positively they need not perform `notifyAll`. Likewise, `reading` must be 0 before it is necessary to wake up anybody from `EndRead`.

Even though the monitor this way has been slightly optimized, one may still risk that a great number of readers are woken up after every writer just to realize that there are still pending writers and wait again. This may put a strain on the monitor.

If one wants a fair solution to the reader/writer problem, it becomes very difficult to use the above scheme. There are more advanced schemes using extra critical regions (see eg. [Lea99]) to implement separate wait queues. In such cases, however, it is recommended to use instead a general Java implementation of a more sofisticated synchronization mechanism like Hoare monitors as described in [hoa].

In [LB00] you can find a number of detailed examples covering many aspects of thread programming in Java.

6 Win32

Win32 is the common application program interface (API) to the Windows operating system family (Windows 95/98/2000/ME/NT/CE).

Win32 offers among many other things, *threads*. As above, threads are light-weight processes with a common address space. See Part I.

For synchronization of threads there are a number of primitive synchronization objects:

- `Mutex` is a mechanism for mutual exclusion.
- `Semaphore` is a classical general semaphore.
- `Event` is a simple flag that may be awaited.

As a unique facility, in Win32 it is possible to wait atomically on all or on one of a set of these synchronization objects.

Critical Regions

A critical region is created by `CreateMutex(`*access*`,`*reserve*`,`*name*`)`, where *access* indicates access rights, *reserve* indicates whether the region should be reserved initially and *name* is a global name for the region. In our examples *access* and *name* ar e not use as indicated by `null`-values. The operation returns a reference (`HANDLE`) that is used for identification of the region.

A critical region with reference m is reserved by `WaitForSingleObject(`m`,`*timeout*`)`. I our examples we do not use timeout indicated by the value `INFINITE`. The region is released with `ReleaseMutex(`m`)`. The region may be reserved several times by the same thread.

`Mutex`-regions may be shared among threads in different programs. For establishing critical regions within a single program, Win32 offers a more efficient mechanism called `CriticalSection` with the operations `InitializeCriticalSection(`*cs*`)`, `EnterCriticalSection(`*cs*`)`, and `LeaveCriticalSection(`*cs*`)`, where c is (the address of) a variable of the type `CRITICAL_SECTION`.

Semaphores

For conditional synchronization, Win32 offers Dijkstra's classical, general counting semaphores. A semaphore is created with `CreateSemaphore(`*access*`,`s_0`,`s_{max}`,`*name*`)` where *access* indicate access rights, s_0 is the initial value of the semaphore, s_{max} is a possible upper limit (with unknown semantics!) and *name* is a global name for the semaphore. The operations returns a reference (`HANDLE`) to identify the semaphore.

A standard *wait* operation (P-operation) on a semaphore is made by Win32's general wait operation: `WaitForSingleObject(`s`,`*timeout*`)` where s is the semaphore reference. As for `Mutex` wait, *timeout* can be set to `INFINITE`.

For signalling (V operation) on a semaphore one uses `ReleaseSemaphore(`s`, `n`, `l`)` that increments the semaphore s with n ($n > 0$). l is a result parameter in which one can get the old value of the semaphore (not useful for very much).

No liveness properties are specified for semaphores and thus only weak fairness should be assumed.

Events

Events are primitive flags on which threads may wait. An event flag can be created with *auto-reset* or *manual reset*. Events are created with `CreateEvent(`*access*`,`*manual*`,`*init*`,`*name*`)` where *access* are the access rights, *manual* is a boolean indicating whether the event should be of the manual reset type, *init* is the initial value of the flag and *name* is the global name of the event. The flag is set and reset by calling `SetEvent(`e`)` and `ResetEvent(`e`)` respectively. A call of `WaitForSingleEvent(`e`,`*timeout*`)` will await that the event flag e becomes set. For an auto-reset event, the flag is reset by the wait, for a manual reset event, the flag remains set.

Thus, events are just simple shared boolean variables and are as notoriously diffucult to use as such. However, the atomic multi-wait of Win32 opens for controlled use of events as guards at the entry to critical regions as described below.

Monitors

A monitor with non-waiting operations can be implemented as a class in which all operations reserve and release a `Mutex` (or `CriticalSection`) associated with each object of the class.

As said above, waiting operations can be implemented by a combination of critical regions and events. The idea is that each entry condition B in the abstract specification of the monitor operations is represented by a event flag E_B which is set exactly when B holds. Now, by waiting atomically on **both** the critical region m and the flag E_B it is ensured that the operation is started atomically with B holding. Of course, this requires all monitor operations to maintain the correspondence between the Bs and their associated event flags.

A fairly good description of the synchronization primitives in Win32 can be found in [Ric95].

6.1 Generel semaphore

As said, Win32 offers directly a classical semaphore. The below semaphore implementation is thus only to illustrate the use of `Mutex` and `Event` for construction of a monitor.

```
class Sem {

  int S;
  HANDLE mutex, nonzero;

public:

  Sem() {
    S = 0;
```

```
    mutex =   CreateMutex(null, false, null);
    nonzero = CreateEvent(null, true, false, null);  // Not set initially
  }

  void P() {
    WaitForTwo(mutex,nonzero);
    S-;
    if (S == 0) ResetEvent(nonzero);
    ReleaseMutex(mutex);
  }

  void V() {
    WaitForSingleObject(mutex,INFINITE);
    S++;
    SetEvent(nonzero);
    ReleaseMutex(mutex);
  }
}
```

Here `WaitForTwo` is an auxiliary operation that makes a atomic wait on two synchronization objects. It can be implemented using the multiple wait of Win32:

```
  void WaitForTwo(Handle h1, h2) {
    HANDLE handles[2];
    handles[0] = h1;
    handles[1] = h2;
    WaitForMultipleObjects(2,&handles, true, INFINITE);
  }
```

6.2 N-synchronization

The meeting problem can be solved by counting the number of processes arrived and then wait on a `flush`-event until all have arrived. To avoid that new processes interfere, the monitor blocks for new processes using an `ok` guard during the sychronization.

```
class Meet {

  int K;
  HANDLE mutex, ok, flush;

public:

  Meet() {
    K = 0;
    mutex =  CreateMutex(null, false, null);
    ok =     CreateEvent(null, true, true, null);   // Initially set
    flush =  CreateEvent(null, true, false, null);  // Initially not set
  }

  void Sync() {
    WaitForTwo(mutex,ok);
    K++;
```

```
    if (K < N) {
      ReleaseMutex(mutex);
      WaitForTwo(mutex,flush);
    }
    else {
      ResetEvent(ok);
      SetEvent(flush);
    }
    K-;
    if (K == 0) {
      ResetEvent(flush);
      SetEvent(free);
    }
    ReleaseMutex(mutex);
  }
}
```

As in the semaphore solution, the auxiliary operation `WaitForTwo` is used.

7 .NET and C$^{\#}$

For an introduction to Microsoft's .NET platform, see the first note in this series [Løv04] or the ECMA/ISO standards [Ecm02a, Ecm02b]. Like we did in the first note, the synchronization mechanisms will be described as they appear in C$^{\#}$, relative to Java.

As for the notion of threads, also the synchronization mechanisms of C$^{\#}$, owes a lot to Java. In addition, also the traditional mechanisms from the Win32 API have been included in the framework.

7.1 Monitors

The basic means of synchronization is that of *monitors*. As in Java, a monitor is a critical region that may be associated with any kind of object. The critical sections belonging to the region must be ecplicitly indicated, either by a synctactic `lock` construct, or through the use of entry and exit primitives. A region has a single anonymous condition queue with signal-and-continue semantics. Below a comparison between the monitor constructs in Java and C$^{\#}$ is presented.:

Facility	**Java**	**C$^{\#}$**
Critical section	`synchronized` P`(...)`	
	`synchronized (`o`) {...}`	`lock (`o`) {...}`
		`Monitor.Enter(`o`)`
		`Monitor.Exit(`o`)`
		`Monitor.TryEnter(`o`,`t`)`
Condition queue	o`.wait()`	`Monitor.Wait(`o`)`
	o`.notify()`	`Monitor.Pulse(`o`)`
	o`.notifyAll()`	`Monitor.PulseAll(`o`)`

where o is the monitor object and t is a timeout parameter. C$^{\#}$ has a few deviations from Java. First, there is no syntatic sugar for turning a whole method into a critical section. Thus, the explicit `lock(o)` construct must be used instead. Secondly, the region may be used in an unstructured way though the use of the primitives `Monitor.Enter(o)` and `Monitor.Exit(o)`. This gives more freedom for making the critical sections as small as possible, but requires a high discipline from the programmer. The `Monitor.TryEnter(o,t)` primitive allows for timeout on entrance to the critical region. This could be used for detecting regions that are never released, indicating a programming error.

The explicit use of the monitor object makes the code a bit more verbose than Java. For instance, a simple semaphore would look like:

```
class Sem {

  int S = 0;

  public void P() {
    lock(this) {
      while (S == 0) Monitor.Wait(this);
      S--;
    }
  }

  public void V() {
    lock(this) {
      S++;
      Monitor.Pulse(this);
    }
  }
}
```

Since the synchronization mechanism in C# is so similar to that of Java, any of the examples from the Java section will carry over. Also, the programming techniques presented in [Lea99] can be applied to C#.

7.2 Other synchronization primitives

Microsoft's implemenation of the .NET platform provides classes that give access to the underlying Win32 primitives that may be used across processes in the underlying operating system (cf. section 6). Thus, these primitives are not part of the ECMA/ISO standard. The primitives include mutex'es and events whereas the classical semaphore has been dismissed. The possibility of awaiting one or all of set of synchronization objects has been retained.

Furthermore, a new *reader/writer lock* has been introduced. It works as a traditional reader/writer lock with fairness for both readers and writers. The mechanism also provides a means of upgrading a reader lock to a writer lock, but unfortunately, this is not done atomically.

The `Interlocking` class, included in the ECMA/ISO standard, provides a set of routines to perform simple atomic actions, like incrementing or decrementing integers. These may be directly supported by indivisible machine instructions reducing the overhead associated with e.g. gathering of statistics.

8 Ada95

Ada95 is a revision of the Ada langauge that was originally developed around 1980 for use in US defense systems.

Ada has an integrated notion of processes called *tasks*. In the original language the basic form of communication was *rendezvous* between two tasks. This had the consequence that data shared among tasks had to be implemented by server tasks that other tasks had to communicate with in order to read or write the common data. Recognizing that shared data is a more efficient way of tranferring common data when processes have a common physical address space, a monitor-like construct was wished for at the revision. On the other side, the high-level synchronization obtained by **when** clauses in the **select** construct was much appreciated.

The result was a monitor-like construct in the form of *protected objects* that are records (structures) that can be accessed only through a number of operations. The operations are executed with mutual exclusion. The operations are divided into *entries* and *procedures/functions*. Each entry has an associated *waiting queue* and a boolean *guard*. An entry is *open* when the associated guard is true. A call of an entry will first await the the critical region is free and then reserve it. Then the guard of the entry is checked and if not true, the calling taks is put on the waiting queue of the entry (releasing the region).

After execution of a monitor operation (entry or procedure), all entry guards are reevaluated and if there are tasks waiting on open entries, one of them will take over the critical region. Only when there are no tasks waiting on open entries, new tasks will be allowed to reserve the region.

Basically a tasks can only wait for some condition at the start of an entry. However, to allow a task to register itself before waiting, it is possible for a taks to `requeue` itself on a new entry. Hereby the tasks will try to execute the new entry using its original parameters.

The most comprehensive description and discussion of Ada's concurrency mechanisms is found in [BW95]. A more general introduction to programming using Ada95 is given in [BA98]

8.1 General Semaphore

A general semaphore is implemented directly corresponding to the abstract specification:

```
protected Sem is
  entry P;
  procedure V;
private
  S: integer := 0;
end Sem;

protected body Sem is

  entry P when S > 0 is
  begin
    S := S - 1;
  end;
```

```
    procedure V is
    begin
      S := S + 1;
    end;

  end Sem;
```

The FIFO-semantics of waiting queues ensures that also the semaphore becomes FIFO.

8.2 N-synchronization

To solve the N-synchronization problem, one may use the possibility of querying the number of tasks waiting at a queue. The number of waiting tasks at entry E is denoted by E`'count`

```
  protected Meet is
    entry Sync;
  private
    flush : boolean := false;
  end Meet;

  protected body Meet is

    entry Sync when Sync'count = N or flush is
    begin
      flush := Sync'count > 0;
    end;

  end Meet;
```

`Meet` works as follows: Each call of `Sync` awaits that the critical region of the object is free. Then the entry guard is checked and if it is true, the calling tasks will be put on the waiting queue of `Sync`. If the entry is still not open, the region is released. The Nth call of `Sync` is thus to enter the queue because `Sync'count`$= N - 1$. After having entered the queue, however, the entry becomes open (`Sync'count` is now N) and the *first* on the waiting queue is thus to be woken up and take over the critical region. This tasks now sets `flush` true (if $N > 1$), whereby the next on the queue is taking over the critical region etc. This will continue as long as there are tasks waiting on `Sync`'s waiting queue and during this activity the region is occupied such that new tasks cannot get in from outside and interfere.

8.3 Reader/Writer

To implement the reader/writer problem with writer priority it is useful that one can see directly how many taksks are waiting at a given entry.

```
  protected ReadWrite is
    entry StartRead;
    entry StartWrite;
    procedure EndRead;
    procedure EndWrite;
```

```
private
  readers: integer := 0;
  writing: boolean := false;
end ReadWrite;

protected body ReadWrite is

  entry StartRead when not writing and StartWrite'count = 0 is
  begin
    readers := readers + 1;
  end;

  procedure EndRead is
  begin
    readers := readers -1;
  end;

  entry StartWrite when readers = 0 and not writing is
  begin
    writing := true;
  end;

  procedure EndWrite is
  begin
    writing := false;
  end;

end ReadWrite;
```

Again this solution is close to the abstract specification due to the automatic test of the entry conditions.

References

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

[BA98] M. Ben-Ari. *Ada for Software Engineers.* Wiley, 1998.

[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`.

[hoa] Hoare monitors in Java.
`http://www.imm.dtu.dk/courses/02220/CP/hoaremon.html`.

[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.

[Lea99] Dough Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2nd edition, 1999.

[Løv04] Hans Henrik Løvengreen. Processes and threads. Course notes, IMM, DTU, 2004. Version 1.4.

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

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