Some Idioms for Synchronisation Mechanisms

For the last 40 years, a long labour and experience has been gathered around concurrent, parallel, and distributed programming from the programming language point of view. Particularly, the approaches by Dijkstra, Hoare, and Brinch-Hansen, developed during the late 1960's and 1970's, provided with the very basic concepts, properties, and characteristics about how to model simultaneous processes and their interaction can be expressed in programming terms. These works represent the basic precedent of what we know nowadays as Parallel Programming.

Many further work and experience has been gathered until today, like the formalisation of the concepts, and their representation in different programming languages for concurrent, parallel and distributed programming. Whenever a program is developed for concurrent, parallel, or distributed execution, most authors refer to these seminal works in order to express basic communication components, making use of the synchronisation mechanisms originally proposed by Dijkstra, Hoare, and Brinch-Hansen [And91, Bac93, Lyn96, Har98, And00].

From the point of view of Parallel Software Design based on Software Patterns, two important concepts for parallel programming have been respectively developed: coordination and communication. To these important concepts within parallel programming, the present paper incorporates a third one: synchronisation. This concept has been implicitly treated when previously dealing with coordination and communication. It seems now the time to express it explicitly: coordination is strongly based on communication, but at the same time, communication is strongly based on synchronisation. And such a synchronisation can be expressed in programming terms as the mechanisms for communication and synchronisation proposed by Dijkstra, Hoare, and Brinch-Hansen. Nevertheless, only counting with these mechanisms is not sufficient for creating a complete parallel program. They neither describe a complete coordination system, nor represent complete communication sub-systems. In order to be effectively applied, the synchronisation mechanisms have to be organised and included within communication structures, which at the same time, have to be composed and included into a larger, whole coordination structure.

Thus, the objective of the present is to provide descriptions of the well-known, synchronisation mechanisms proposed by Dijkstra, Hoare, and Brinch-Hansen, in the form of idioms, expressed in terms of a parallel programming language. Therefore, here semaphores [Dij68, Har98], critical regions [Bri72, Hoa72], monitors [Hoa74, Bri75, Har98], message passing primitives [Hoa78, Har98], and remote procedure calls [Bri78, Har98] are presented using a pattern description as idioms for developing synchronisation mechanisms. Each one of them is introduced along with some programming examples expressed in a particular parallel programming language, which enable synchronisation and communication between parallel processing components. Such a description of synchronisation mechanisms as idioms aims to aiding parallel software designers and engineers with a description of some common programming structures within a particular programming language, used for synchronising the communication activities, as well as providing guidelines in their use and selection during the final design and initial implementation stages of a Parallel Software System. Therefore, this paper presents Some Idioms for Synchronisation Mechanisms, which describe the common programming structures used within a communication component, and whose development as implementation structures constitutes the main objective of the Detailed Design step within the Pattern-based Parallel Software Design Method. The Idioms presented here account for the common synchronisation mechanisms for concurrent, parallel, and distributed programming: the Semaphore idiom, the Critical Region idiom, the Monitor idiom, the Message Passing idiom, and the Remote Procedure Call idiom. In the following sections, all these idioms are presented more likely describing the use of the synchronisation mechanism regarding a particular parallel programming language rather than defining or characterising such a synchronisation mechanism.

The Semaphore Idiom

A semaphore is a synchronisation mechanism that allows two or more concurrent, parallel, or distributed software components, executing on a shared memory parallel platform, to block (or wait) for an event to occur. It is intended to solve the mutual exclusion problem, in which the software components should not be allowed to manipulate a shared variable at the same time [Dij68, And91, Bac93, KSS96, Har98, And00].

Example

The C programming language has been very commonly extended in order to cover aspects of concurrent, parallel, and distributed programming. As such, semaphores have been implemented in the C programming language using an extended library by POSIX, and commonly used for implementing concurrent programs, and particularly, operating systems.

For this example, let us consider the pipe component based on the Shared Variable Pipe pattern. Originally, in this section, this example is solved using Java-like monitors. In the present example, the objective is to use semaphores in C as the synchronisation mechanism involved in the Shared Variable Pipe pattern. So, in order to use semaphores in C, some details about POSIX semaphores should be considered, all defined in the file `<semaphore.h>` [KSS96, And00]:

- `sem_t* sem_open(cont char *name)`. Returns a pointer to a semaphore.
- `int sem_close(sem_t *semaphore)`. Destroys the pointer to a semaphore.
- `int sem_init(sem_t* semaphore, int pshared, unsigned int count)`. Sets an integer initial `count` value to the semaphore. If `pshared` is not zero, the semaphore may be used by more than one thread.
- `int sem_wait(sem_t* semaphore)`. Decrements the semaphore. If it is zero, blocks until other thread increments it.
- `int sem_post(sem_t* semaphore)`. Increments the semaphore. If the semaphore is incremented from zero and there are threads blocked, one is awakened.

Context

The context for the Semaphore idiom is in general the same than the context for the Critical Region idiom and the Monitor idiom: a concurrent, parallel, or distributed program is developed, in which two or more software components execute simultaneously on a shared memory parallel platform, and thus, communicating by shared variables. Within each software component, there is at least one critical section, this is, a sequence of instructions that access the shared variable. At least, one software component writes to the shared variable, and conceptually, more than one memory location is written.

Problem

In order to keep the integrity of data, it is required to give to a set of software components a synchronous and exclusive access to shared variables for an arbitrary number of read and write operations.

Forces

In order to apply the semaphore as an idiom, the following set of forces should be taken into consideration [Dij68]:

- The software components execute concurrently or simultaneously, at different relative speeds, and non-deterministically. Their synchronisation should be as independent as possible of any interaction pattern or action of any other software component.
- Operations of inspection and assignment for synchronisation purposes are defined as atomic or indivisible.
- Each software component should be able to enter its critical section and modify the shared variable if and only if this access is confirmed to be safe and secure. Any other software component should synchronise regarding this situation.
- The integrity of the values within the shared variable should be kept during all the communication.

Solution

Use semaphores for synchronising the access to the critical section associated with a shared variable, a process, or a resource. A semaphore is a type of variable or abstract data type, normally represented by a non-negative integer and a queue, with the following atomic operations [Dij68]:

- `signal(semaphore)`: If the value of the semaphore is greater than zero, then decrement it and allow the software component to continue, else suspend the software component process, noting that it is blocked on this semaphore.
- `wait(semaphore)`: If there are no software component processes waiting on the semaphore then increment it, else free one process, which continues at the instruction just after its `wait()` instruction.

Structure

Figure 1 sketches the concept of a semaphore, considering as an abstract data type with a value, a queue pointer, and an interface composed of two operations: `signal()` and `wait()`.

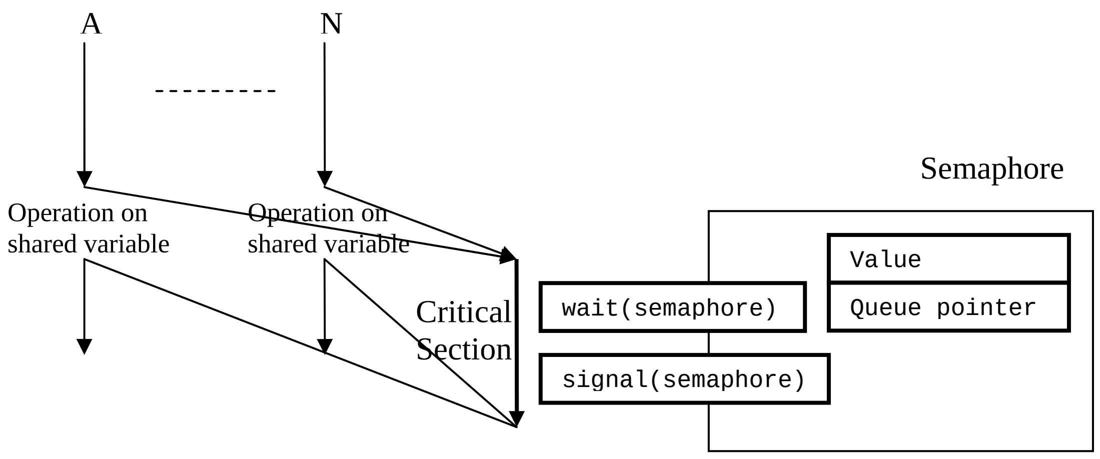


Figure 1.

If semaphores are made available in a programming language, their typical usage is as Figure 2 shows.

```
semaphore lock = 1;
int main(){
    ...
    wait(lock);
    // shared variable access
    signal(lock);
    ...
 }
```

Figure 2.

Dynamics

Semaphores are common synchronisation mechanisms which can be used in a number of different ways. In this section, let us consider that semaphores are used for mutual exclusion and synchronisation of cooperating software components.

- Case 1. Mutual exclusion. Figure 3 shows a possible UML Sequence Diagram depicting three concurrent or parallel software components, namely **A**, **B**, and **C**, which are expected to share a data structure. This shared data structure (not shown in the diagram) is protected by a semaphore called **sem,** which is initialised with a value of 1.

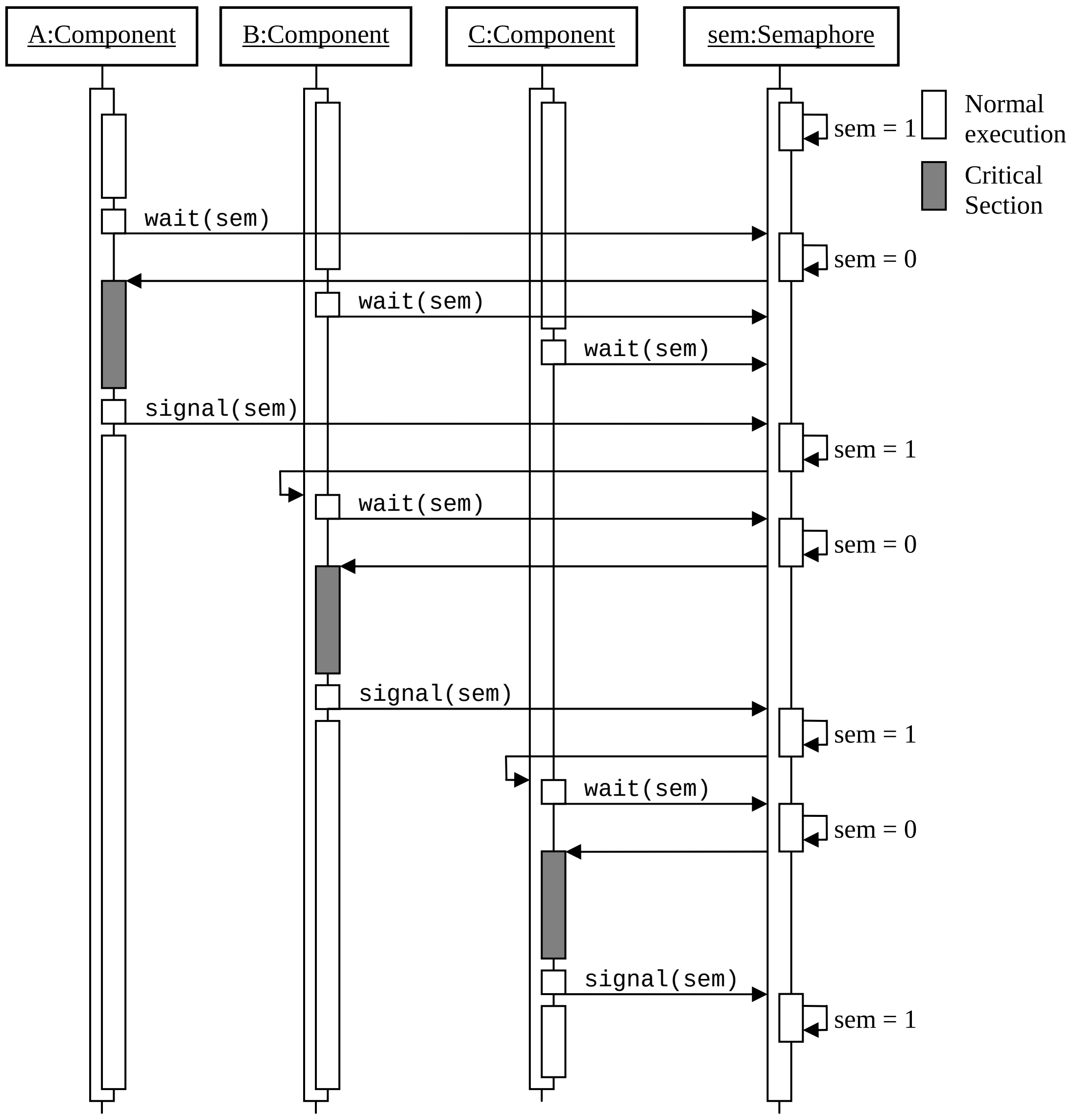


Figure 3.

The software component **A** first executes `wait(sem)` and enters its critical section, which accesses the shared data structure. While **A** stays in its critical section, **B**, and later, **C** attempt to enter their respective critical sections for the same shared data structure, executing `wait(sem)`. Notice that the three software components can proceed concurrently, but within their critical section, only one software component accesses the shared data structure at once. The semaphore **sem** goes through the states shown in Figure 3, while these changes occur. Notice as well that a scheduling policy has been considered on the semaphore, which is a first-in-first-out policy: the first software component waiting on the queue is the one freed on `signal(sem)`. This is a decision of the implementation. Another possibility is to free all waiting software components, and make them execute `wait(sem)` again, so one accesses the critical section while the rest goes waiting again. Yet, there are other alternatives, which obviously depend on the way the semaphore is implemented.

- Case 2. Synchronisation of cooperating software components. Figures 4 and 5 show respectively a UML Sequence Diagram with two concurrent or parallel software components, namely **A** and **B**, which synchronise their activities through a semaphore **sem**. When **A** reaches a certain point in its execution, it cannot continue until **B** has performed a certain task. This is achieved by using the semaphore **sem**, which has been initialised to zero, in which **A** performs `wait(sem)` at the synchronisation point, and in which **B** should perform `signal(sem)`. Figure 4 shows how **A** performs `wait()` before **B** performs `signal()`, and Figure 5 shows how **A** performs `signal()` before **B** performs `wait()`.

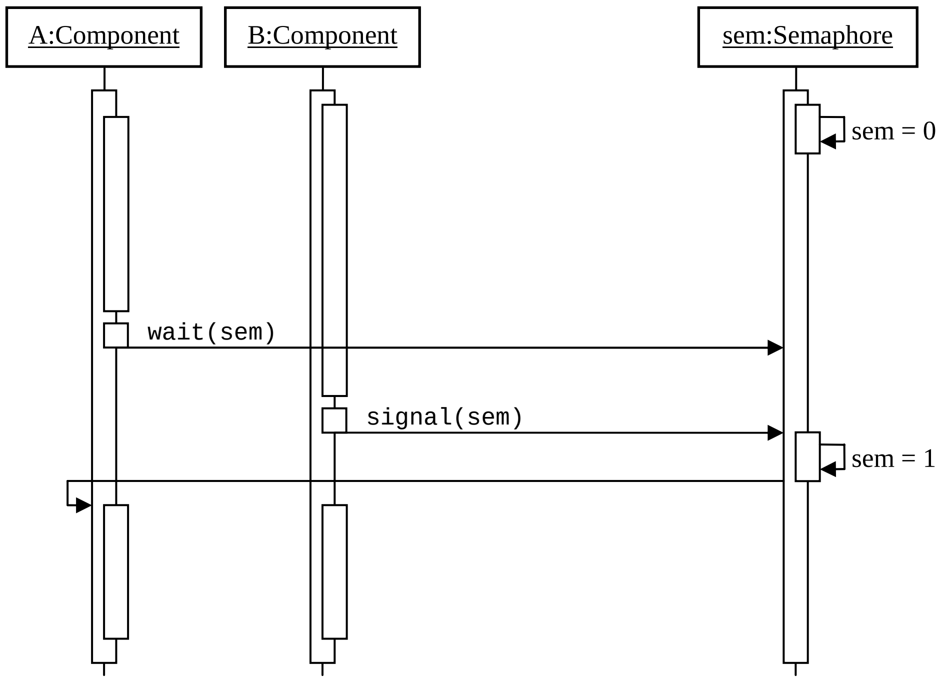


Figure 4.

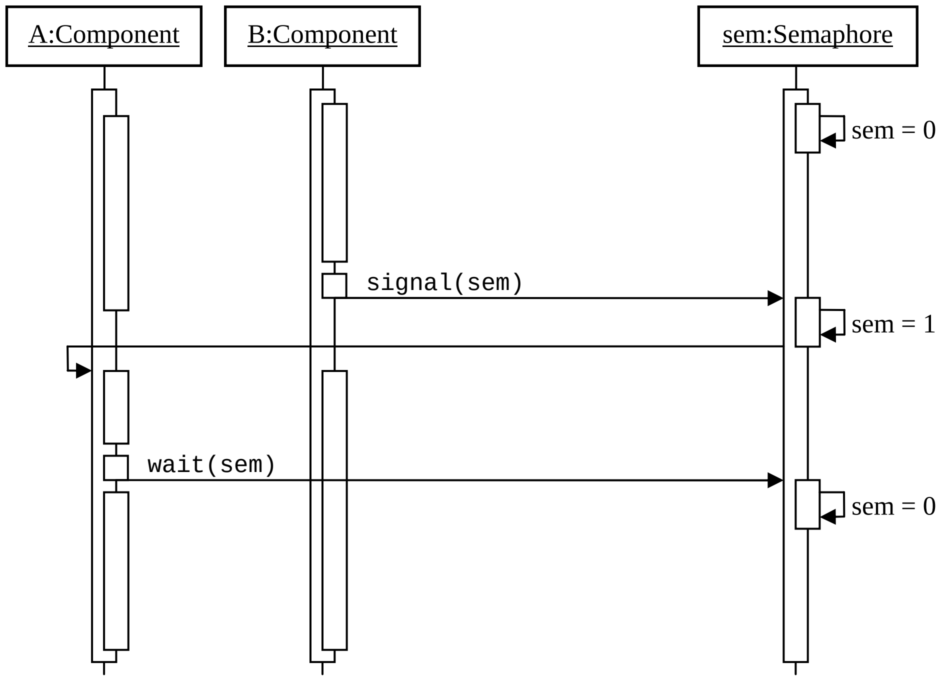


Figure 5.

Example Resolved

An implementation that makes use of the POSIX semaphore in C as the synchronisation mechanism for the pipe component based on the Shared Variable Pipe pattern is proposed here, as Figure 6 shows.

```
include <semaphore.h>

sem_t empty; // semaphore for empty buffer
sem_t full;  // sempahore for full buffer

struct{
   double data[BSIZE]; // buffer
   int n;              // buffer size
} buf[2];


int main(){
   pthread_t t_reader;
   extern void sender(), receiver();
   int err;

   sem_init(&empty,0,2);
   sem_init(&full,0,0);

   err = pthread_create(&t_sender, NULL, (void*(*)(void*))
      sender, NULL);
   if(err) exit(1);

   receiver();
   return 0;
}

void sender(){
   int i = 0;
   size_t = n;

   do{
      sem_wait(&empty);
      n = read(0,buf[i].data,BSIZE);
      buf[i].n = n;
      sem_post(&full);
      i = (i + 1)%2;
   } while (n > 0);
}

void receiver(){
   int i = 0;
   size_t = n;

   do{
      sem_wait(&full);
      n = buf[i].n;
      if (n > 0) {
         write(1,buf[i].data,nbytes);
         sem_post(&empty);
         i = (i + 1)%2;
      }
   } while (n > 0);
}
```

Figure 6.

Known Uses

Since its introduction by E.W. Dijkstra in 1968 [Dij68], the semaphore has been widely used in many applications by many authors as a synchronisation solution. Some of the (perhaps) mostly well known uses are:

- The THE operating system is a multi-process concurrent program, which makes use of semaphores as communication and synchronisation mechanisms between processes [Dij68a, Bac93].
- The producer-consumer bounded buffer problem is widely used as the basic use example of the semaphores by many authors [Dij68, And91, Bac93, KSS96, Har98, And00].
- The readers and writers problem is another classical example of the use of semaphores as synchronisation mechanisms among concurrent processes [Dij68, And91, Bac93, KSS96, Har98, And00].

Consequences

Benefits

- Semaphores are a more general and flexible synchronisation mechanism than a one-to-one (named-process to named-process) scheme, allowing concurrent or parallel software components to execute synchronously and as independently as possible.
- The semaphore operations `wait()` and `signal()` (inspection and assignment) are defined as atomic or indivisible, always aiming for synchronisation purposes.
- By applying a simple protocol over a semaphore, it is assured that each software component is able to enter its critical section and modify the shared variable safely and securely. Other software components actually synchronise regarding this situation: `wait()` can be used by many processes to wait for one or more signalling processes; `signal()` can be used by many processes to signal one waiting process.
- Given the synchronisation provided by the semaphore, the integrity of the values within the shared variable is normally kept during all the communication.

Liabilities

- The use of semaphores is carried out only by convention, and its use is generally not enforced within any programming language. This means that it is very easy to make mistakes when programming many semaphores. Commonly, it is difficult to keep in mind which semaphore has been associated with which shared variable, process, or resource. Also, it is fairly easy to use `wait()` and accidentally access the unprotected shared variable, or to use `signal()` and leave a shared variable locked indefinitely.
- The operations over a semaphore do not allow a 'test for busy' without a commitment to blocking. As an alternative, it might be preferable to wait on the semaphore.
- Semaphores must be individually used. It is simply not possible to specify a set of semaphores as an argument list to a single `wait()` operation. If this could be possible, alternative ordering of actions could be programmed according with the current state of arrival of signals. Such a facility would be difficult to implement, since it would introduce overhead.
- The time for which a software component remains blocked on a semaphore is not limited, based on the definition used here. A software component may block indefinitely until released by a signal.
- Using semaphores, there is simply no means by which one software component may control another without the cooperation of the controlled software component.
- If semaphores are the only synchronisation mechanism available, and it is necessary to pass information between software components, they must share (part of) their address space in order to directly access shared writeable data. A similar buffered scheme such as producer-consumer is required. The semaphore value could be used to convey minimal information, but it is normally not available to be processed.

Related Patterns

As a software pattern, the Semaphore idiom can be considered related with the components of all those concurrent, parallel and distributed software systems in which they are used as synchronisation mechanism [POSA1, Lea97, POSA2, MSM04, POSA4].

As part of the Pattern-based Parallel Software Design Method, the Semaphore idiom can be extensively used to implement synchronisation mechanisms for the Shared Variable Pipe pattern, the Message Passing Pipe pattern, the Multiple Local Call pattern, the Shared Variable Channel pattern, the Message Passing Channel pattern, and the Local Rendezvous pattern. The objective is to use it to synchronise the activity within a communication sub-structure.

Finally, the Semaphore idiom represents a way to describe the use of the semaphore as a synchronisation mechanism for concurrent and parallel applications, using shared variables. Nevertheless, its use can be replaced by other more sophisticated approaches presented as other idioms here: the Critical Region idiom and the Monitor idiom.

The Critical Region Idiom

A critical region is a synchronisation mechanism that allows two or more concurrent, parallel, or distributed software components executing on a shared memory parallel platform, to access code regions guaranteeing the mutual exclusion among them. Shared variables are grouped into such named regions, and tagged as being private resources. Software components are not allowed to enter a critical region when another software component is active in any associated critical region. Conditional synchronisation is performed by guards. When a software component attempts to enter a critical region, it evaluates the guard (under mutual exclusion). If the guard evaluates false, the software component is suspended or delayed. No access order can be assumed [Bri72, Hoa72, And91, Bac93].

Example

OpenMP is an Application Program Interface (API) specified as library extensions for C, Fortran, and C++, used to direct multithreaded, shared memory parallelism. It is a portable, scalable model for developing parallel software systems on a wide range of parallel programming platforms [OpenMP, HX998, And00, MSM04, CJV+07].

For the example here, a synchronisation mechanism based on the Critical Region idiom is to be developed for the example channel component of the Shared Variable Channel pattern. The structure of the solution is presented making use of semaphores in a Java-like code. Nevertheless, such a solution is re-taken here making use of the critical regions in the C programming language, which is extended using OpenMP. Critical regions are defined with the `critical` directive as part of the library `<omp.h>` (an equivalent form of critical region in OpenMP is defined for Fortran, although it is not described here) [OpenMP, HX998, And00, MSM04]. In C, this directive is used to define a critical region as shown in Figure 7.

```
#pragma omp critical [name]
{shared variable access block}
```

Figure 7.

The `critical` directive generates a section of code for mutual exclusion. This means that only one thread executes the structured block at a time within the critical region. Other threads have to wait their turn at the beginning of the directive. In the syntax shown in Figure 7, the identifier `name` is used as a support for disjoint different critical regions.

Context

The context for the Critical Region idiom is in general the same than the context for the Semaphore idiom and the Monitor idiom: a concurrent, parallel, or distributed program is developed, in which two or more software components execute simultaneously on a shared memory parallel platform, and thus, communicating by shared variables. Within each software component, there is at least one critical section, this is, a sequence of instructions that access the shared variable. At least, one software component writes to the shared variable, and conceptually, more than one memory location is written.

Problem

In order to keep the integrity of data, it is required to give to a set of software components a synchronous and exclusive access to shared variables for an arbitrary number of read and write operations.

Forces

In order to apply the critical region as an idiom, the following forces should be taken into consideration [Bri72, Hoa72]:

- There is a set of concurrent or parallel software components, executing non-deterministically and at different relative speeds. All software components should synchronise as independently as possible of any other software component.
- Synchronisation is performed by atomic or indivisible operations of inspection and assignment.
- Each software component should have the possibility to enter its critical section and modify the shared variable if and only if this access is confirmed to be safe and secure. Any other software component should synchronise regarding this situation.
- The integrity of the values within the shared variable should be kept during all the communication.
- The correct use of operations of the synchronisation mechanism should be enforced and ensured.

Solution

Use critical regions for synchronising the access to the critical section associated with a shared variable, a process, or a resource. A critical region is a subroutine programming construct based on semaphores, specifying *(a)* the data shared by processes of the program, *(b)* which semaphore is associated with which shared data, and *(c)* where in the program the shared data is accessed. Thus, critical regions are syntactically specified by [Bri72, Hoa72]:

- `shared`, as an attribute of any data type.
- `region`, declared as:

  `region` *shared_data* `{` *`structured_block`* `}`

In compile time, it is possible to create a semaphore for each shared data declaration, inserting a `wait()` operation at the start of the critical section and a `signal()` operation at the end.

During the evolution of concurrent programming languages, additionally to critical regions, a variation, the conditional critical regions, emerged as a proposal of another synchronisation mechanism, with an associated `await(condition)` primitive. In the proposed form, this is difficult to implement, since condition could be any conditional expression, for example, `await(c>0)`. It is difficult to establish whether the many conditions involving programming language variables awaited by processes have become true.

Structure

Figure 8 shows a sketch of the concept of a critical region, considering it as a structured construct to ensure mutual exclusion of a critical section.

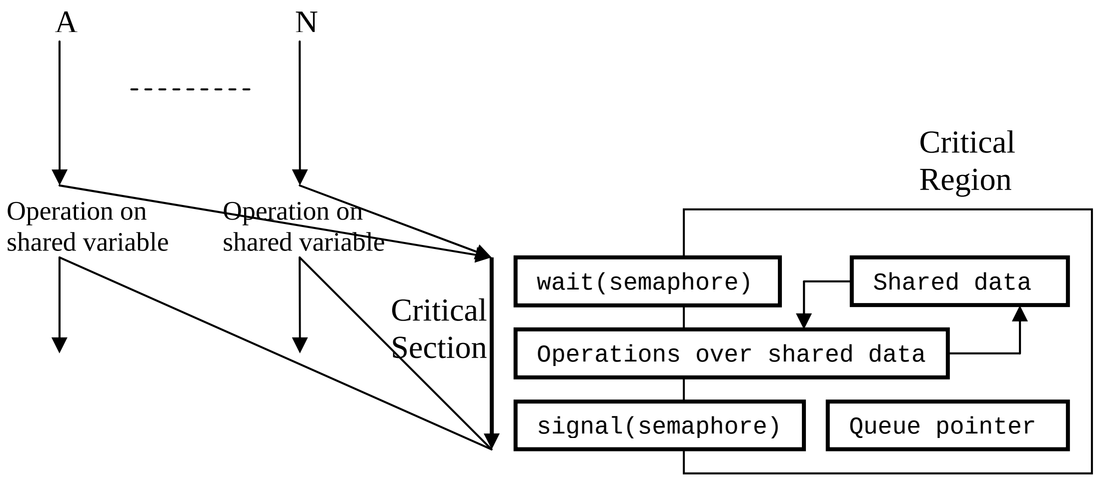


Figure 8.

If critical regions are available for a programming language, their usage is very similar to the code shown in Figure 9.

```
resource r(variable_declarations){
   // operations_on_shared_variables
};
...
int main(){
    ...
    region r when B;
    ...
 }
```

Figure 9.

Every shared variable must belong to a resource, declared in Figure 9 with the identifier `r`. The resource is composed by one or more declarations. The variables in a resource may be only accessed within `region` statements that explicitly call the resource. In this statement, `r` is the resource name, `B` is a Boolean guard, which implies that when invoked, the execution of `region` is delayed until `B` is true; and then the operations on the shared variables declared within the resource are executed. The execution of `region` statements that name the same resource is mutually exclusive. In particular, `B` is guaranteed to be true when the execution of the operations begins.

Dynamics

Like semaphores, critical regions are used in several ways as common synchronisation mechanisms. Let us consider that critical regions are used particularly for mutual exclusion. Figure 10 shows a UML Sequence Diagram of a possible execution of three concurrent or parallel software components, **A**, **B**, and **C**, which share a data structure (not shown in the diagram), which is accessed only through a critical region **r**.

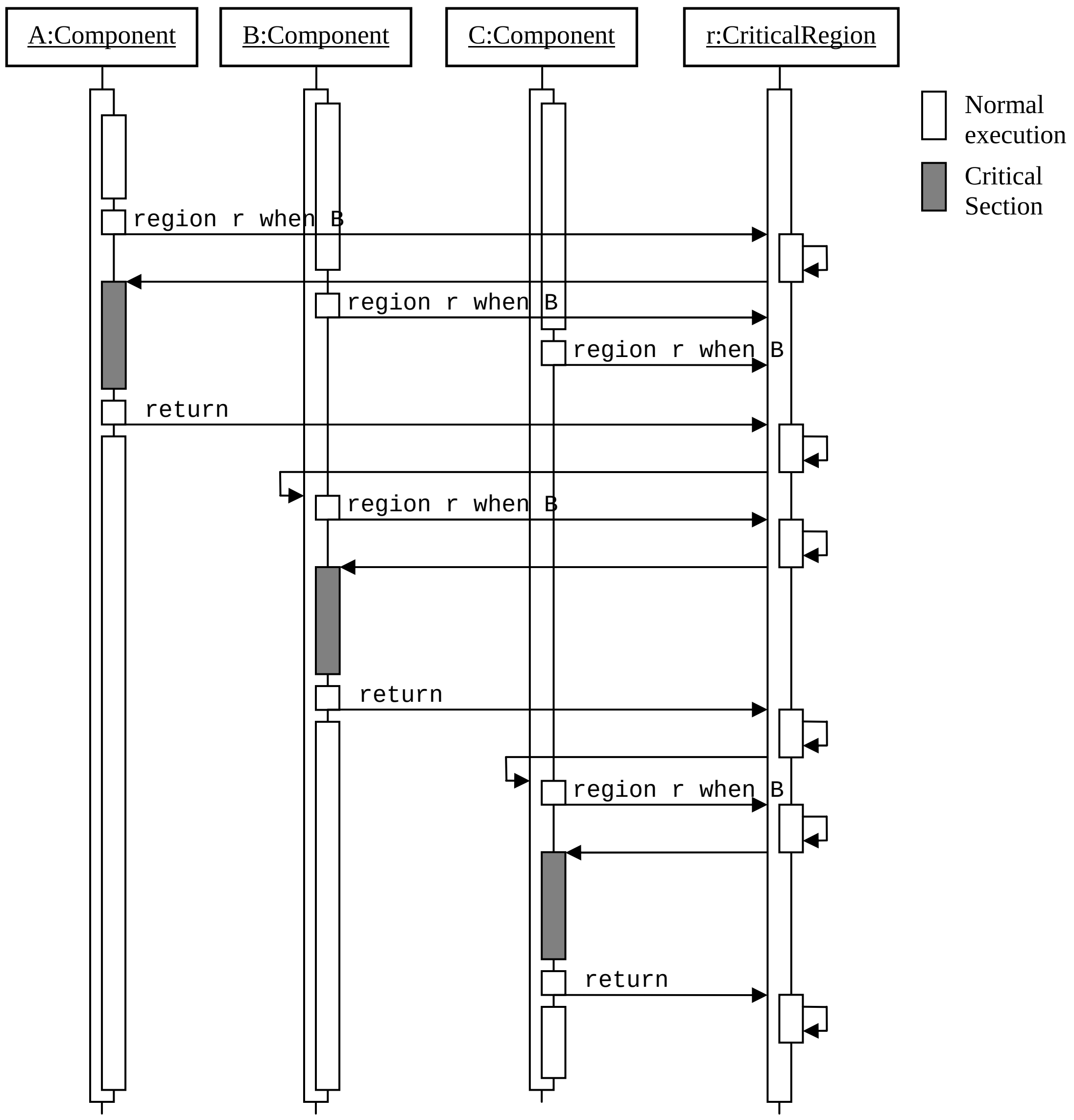


Figure 10

.

The synchronisation among software components starts when software component **A** executes `region r when B`. It is assumed that condition `B` is true, so **A** accesses the critical region **r**, in which the shared data structure is accessed, and locking it by making condition `B` to take the value of false. While **A** stays in the common critical region, **B** and **C** attempt to enter respectively executing `region r when B`. As condition B has been set to false by **A**, neither **B** nor **C** are able to continue, so they have to wait until **A** exits the critical region. Only then, **B** is able to enter the critical region. And only after **B** leaves the critical region, **C** is able to enter. Notice that even though the three software components can proceed concurrently, only one software component accesses the critical region, and thus, only this component is able to access the shared data structure at once.

Example Resolved

An implementation is presented using the OpenMP directive `critical` in C as the synchronisation mechanism for the channel component based on the Shared Variable Channel pattern is proposed here, as Figure 11 shows.

```
include <omp.h>

struct{
   double data[BSIZE]; // buffer
   int n;              // buffer size
   int putIn;          // index
   int takeOut;        // index
   int count;          // number of items
} buf[4];

void sendToNext(double data){
   buf.data[putIn] = data; // write to shared variable
   buf.putIn = (buf.putIn + 1) % buf.n;
   buf.count++;
}

double receiveFromPrevious(){
   double data;
   data = buf.data[takeOut]; // read from shared variable
   buf.takeOut = (buf.takeOut + 1) % buf.n;
   buf.count--;
   return data;
}

int main(){

   extern void sender(), receiver();

   # pragma omp parallel
   {
      sender();
      receiver();
   }
   return 0;
}

void sender(){
   double data;
   ...
   # pragma omp critical
   {
      sendToNext(data);
   }
   ...
}

void receiver(){
   double data;
   ...
   # pragma omp critical
   {
      data=receiveFromPrevious();
   }
   ...
}
```

Figure 11.

Known Uses

Since their introduction in 1972 by P. Brinch-Hansen [Bri72] and C.A.R. Hoare [Hoa72], the critical region has been proposed as a synchronisation solution. Nevertheless, its use was not so extensive in a number of programming languages, due to it soon became substituted by the concept of Monitor. However, several authors still consider the use of critical regions in some programming languages [And91, Bac93]. Some of the (perhaps) mostly well known uses are:

- Critical regions are used for scheduling and resource allocation in a complete programming example for operating systems [And91].
- Readers and writers is an example proposed originally by Dijkstra, which has been widely solved using several synchronisation mechanisms, in this case, critical regions [Bac93]
- Critical regions are used in a parallel implementation in OpenMP and C, aiming to provide a solution to the Jacobi iteration [And00].

Consequences

Benefits

- As with semaphores, the concurrent or parallel software components are allowed to execute non-deterministically and at different relative speeds. However, they are allowed as well to synchronise as independently as possible of any other software component.
- Synchronisation is an atomic operation within the critical region.
- Every software component has the opportunity to access the shared variables by entering the critical region. Any other software component has to synchronise regarding this situation.
- The integrity of the values within the shared variable is preserved during all the communication.
- The use of operations of the synchronisation mechanism are enforced and ensured by making use of structured programming concepts.

Liabilities

- The critical region construct, by itself, has no way to enforcing modularity, and a program may be structured so critical regions generate potential delays.

- In practice, as with semaphores, elaborate conventions and working practices would be used in addition to the language constructs.

Related Patterns

The Critical Region idiom, as a software pattern, is related with the components of all those concurrent, parallel and distributed software systems in which they are used as synchronisation mechanism [POSA1, Lea97, POSA2, MSM04, POSA4].

On the other hand, the Critical Region idiom can be used to implement synchronisation mechanisms for the Shared Variable Pipe pattern, the Message Passing Pipe pattern, the Multiple Local Call pattern, the Shared Variable Channel pattern, the Message Passing Channel pattern, and the Local Rendezvous pattern. Its objective here is to use it as a synchronisation mechanism within a communication sub-structure.

Finally, the Critical Region idiom represents another way to synchronise the activity of concurrent and parallel software components, which communicate using shared variables. Nevertheless, as a synchronisation mechanism, it can be replaced by other approaches, such as the Semaphore idiom and the Monitor idiom.

The Monitor Idiom

A monitor is a synchronisation mechanism based on the concept of object, which encapsulates shared variables. Inside the monitor, shared variables are tagged as being private resources, and thus, the only way to manipulate them is to call on methods of the interface that operate over the shared variables. This is the only way of allowing the exchange of data among two or more concurrent, parallel, or distributed software components executing on a shared memory parallel platform. Mutual exclusion among them is implicit and guaranteed by the compiler, allowing only one software component at a time to be active inside the monitor, this is, executing one of the methods. No execution order can be assumed [Hoa74, Bri75, And91, Bac93, Har98, HX98, And00].

Example

The Java programming language is capable of creating and executing threads on the same processor or on different processors. In order to allow communications among threads, Java specifies the `synchronized` modifier. Hence, to implement a monitor as an object in Java, the `synchronized` modifier is used for all methods of the class, in which only one thread should be executing at a time. These methods are normally declared as `public`, and modify shared variables declared as `private` inside the monitor. Nevertheless, the methods could also be declared as `private`, if the public access to the monitor consists of calls to several of these synchronized methods. [Har98, And00, MSM04].

For the present example, it is proposed the development of a synchronisation mechanism based on the Monitor idiom for the example of a synchronisation mechanisms component for the sender side of the Message Passing Pipe pattern. In this section, the structure of the sender is presented making use of a monitor in a Java-like pseudo-code, which allows synchronising the access to an output data stream, which is connected with a socket. In the general case, the Monitor idiom in the Java programming language has a form similar to the one shown in Figure 12.

```
class Monitor {
    ...
    private type shared_variables;
    ...
    public syncronized type method() {
       // operations on private shared_variables;
       ...
    }
    ...
}
```

Figure 12.

In Java, each object has an associated lock. Hence, a thread that invokes a method with the `synchronized` modifier in an object must first obtain the lock of the object before executing the code of the method, and thus, executing it in mutual exclusion with the invocations from other threads. Only one thread executes a `synchronized` method at a time within the object. Other threads block if the lock is currently held by other thread.

Context

The context for the Monitor idiom is, in general, very similar to the context for the Semaphore idiom and the Critical Region idiom: a concurrent, parallel, or distributed program is developed, in which two or more software components execute simultaneously on a shared memory parallel platform, and thus, communicating by shared variables. Each software component accesses at least one critical section, this is, a sequence of instructions that access the shared variable. At least, one software component writes to the shared variable.

Problem

In order to keep the integrity of data, it is required to give to a set of software components a synchronous and exclusive access to shared variables for an arbitrary number of read and write operations.

Forces

In order to apply the Monitor idiom, the following forces are taken into consideration [Hoa74, Bri75]:

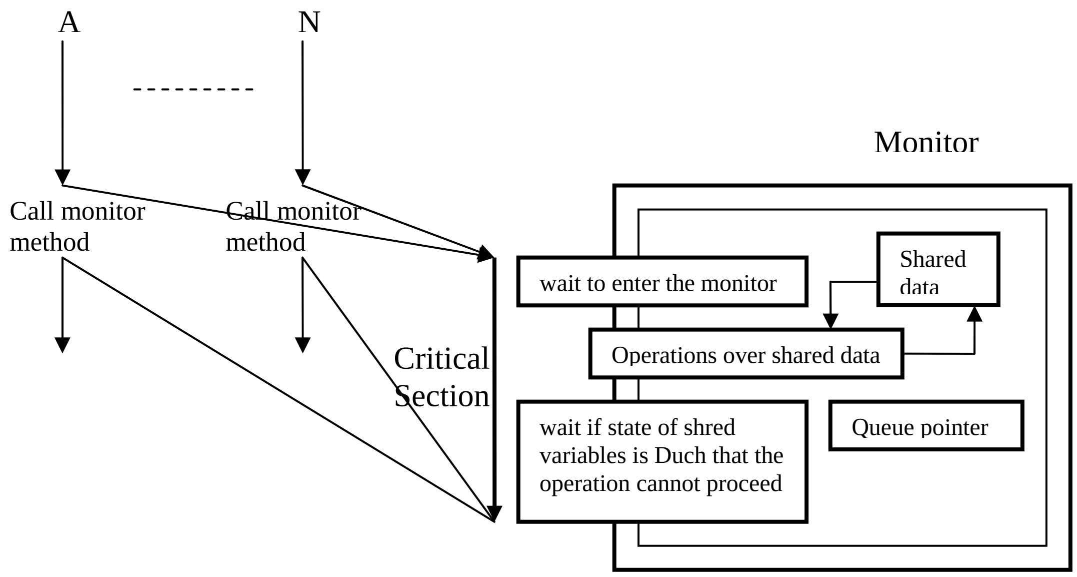
- A set of concurrent or parallel software components non-deterministically execute at different relative speeds. All of them should synchronously act as independently as possible of the others.
- Synchronisation is carried out by operations of inspection and assignment, which have to be atomic or indivisible.
- Each software component should have the possibility to execute the code associated with a critical section, accessing the shared variables if and only if such an access is safe and secure. Any other software component should block, waiting for this software component to finish its access.
- The values of the shared variables should keep their integrity during all the communication.
- The correct use of operations over shared variables should be enforced and ensured.

Solution

Use monitors for synchronising the access to the critical section associated with a shared variable, a process, or a resource. A monitor has the structure of an abstract data object, in which the encapsulated data is shared, and each operation is executed under mutual exclusion. Only one process is active in the monitor at any time [Hoa74, Bri75].

Structure

A sketch is shown Figure 13, presenting the concept of monitor as an object that, due to its encapsulation characteristic and its locking mechanism, ensures mutual exclusion over a critical section.


A
N

Monitor
Call monitor method
Call monitor method
Critical Section
wait to enter the monitor
Shared data
Operations over shared data
wait if state of shred variables is Duch that the operation cannot proceed
Queue pointer

Figure 13.

If a programming language has defined monitors as synchronisation mechanisms between concurrent processes, their usage is normally very similar to the code shown in Figure 14 [Hoa74].

```
monitor monitor_name{
   ...
   // declarations of shared variables and local data
   private type shared_variables;
   private type local_data;
   ...
   // declaractions of methods
   public synchronized type method(type formal_parameters){
      ...
      // operations_on_shared_variables
      ...
   }
};
...
int main(){
   ...
   monitor m;
   ...
   m.method(actual_parameters);
}
```

Figure 14.

Every shared variable is encapsulated within a monitor. The monitor, as an abstract data type (a class) is composed of one or more declarations of `private` variables and `public` methods. The variables may be only accessed by the `synchronized` methods that explicitly are called to access the shared variables. In the invocation statement, `m.method()` within the main function executes the defined operations over the shared variables with actual parameters. The execution inside the monitor is mutually exclusive among software components that access it.

Dynamics

Just as it is the case with semaphores and critical regions, monitors are used in several ways as common synchronisation mechanisms. Here, monitors are used particularly for mutual exclusion. Figure 15 shows a UML Sequence Diagram of the possible execution of three concurrent or parallel software components, **A**, **B**, and **C**, which share a data structure (not shown in the diagram), which is encapsulated within a monitor, and thus, can only be accessed through invocations to the monitor's methods.

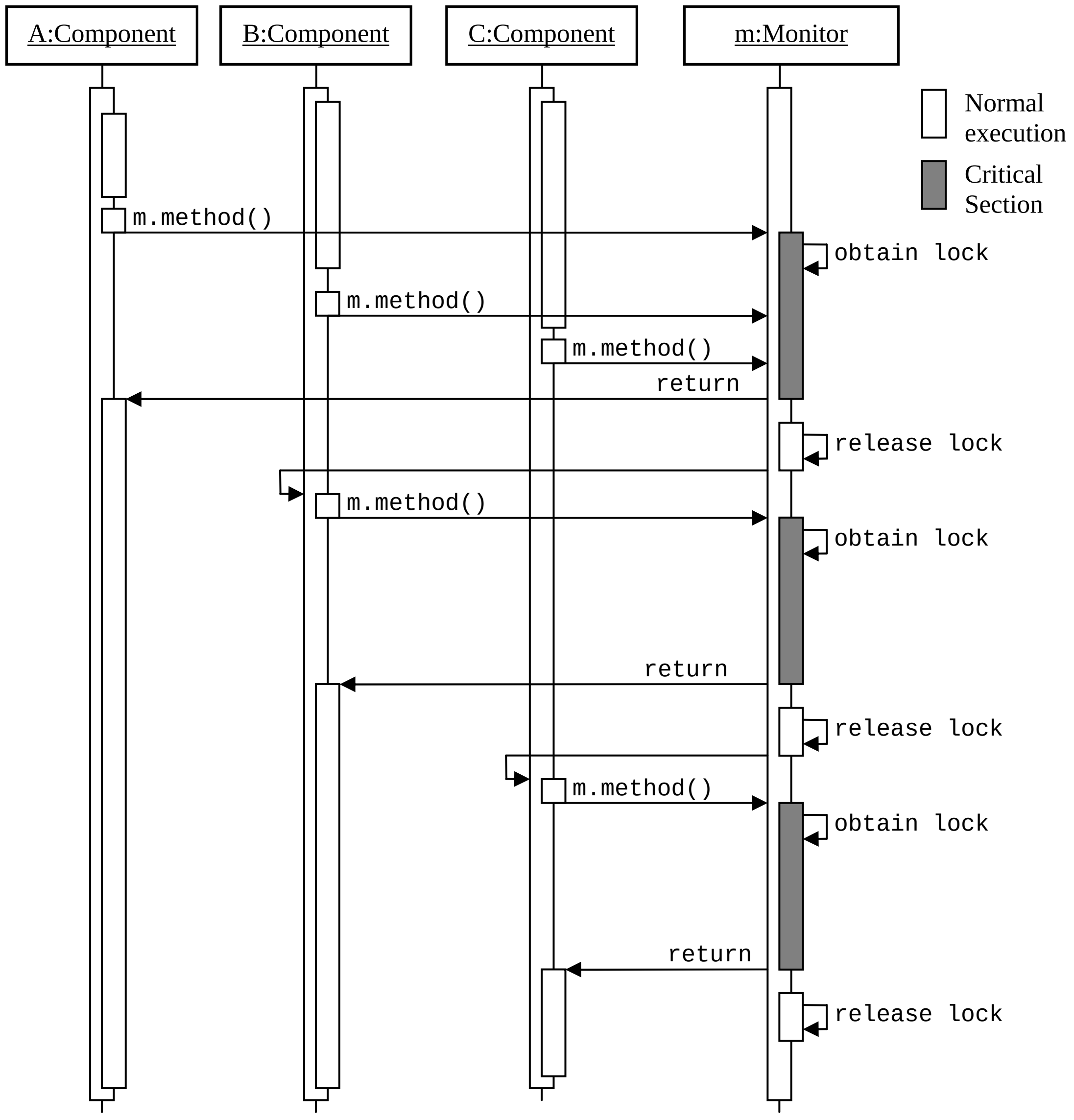


Figure 15.

The mutual exclusion among software components starts when **A** invokes `m.method()`. It is assumed that the monitor is free at such a moment, so **A** obtains its lock and performs `method()`, which allows the accesses to the shared variables. As long as **A** remains inside the monitor, **B** and **C** may attempt to respectively invoke the same call `m.method()`. However, as **A** owns the monitor's lock, neither **B** nor **C** are able to succeed, so they have to wait until **A** leaves the monitor. Only when this has happened, **B** is able to enter the monitor. And only after **B** leaves it, then **C** is able to enter. Notice that even though the three software components proceed concurrently, only one software component accesses the monitor, and thus, only this component is able to access the shared variables at once.

Example Resolved

An implementation for the synchronisation mechanism for the sender side of the Message Passing Pipe pattern is proposed here is presented using the `synchronized` modifier in Java, as shown in Figure 16.

```
class Monitor {
    ...
    private int numMessages = 0;
    private Vector messages = new Vector();
    ...
    public syncronized void send(double data) {
         if (data == null) throw NullPointerException();
         numMessages++;
         try { outObj.writeObject(data);}
         match (IOException e)
              {throw new MessagePassingException();}
         if (numMessages <= 1) notify();
    }
    ...

}
...
class SendingFilter implements Runnable{
   Monitor monitor; // referente to the sending monitor
   double data; // data to be sent
   ...
   public void run(){
      ...
      // Operations on local data
      ...
      monitor.send(data);
      ...
   }
}
```

Figure 16.

Known Uses

Monitors are proposed by C.A.R. Hoare in 1974 [Hoa74] and P. Brinch-Hansen in 1975 [Bri75], as yet another synchronisation solution to the mutual exclusion problem. It soon became popular among the programming languages which included the use of concurrent and parallel processes in single-processor and shared memory parallel platforms. Up to now, several authors still consider monitors as a basic construction for inter-process communication in some programming languages [And91, Bac93, Har98, HX98, And00]. Some of the most widely known uses of monitors are:

- Monitors are used for communication and synchronisation of process activities and resource use within concurrent operating systems such as Solo [Bri77] and others [Bac93].
- Monitors are used as synchronisation mechanisms in the implementation of scheduling the access to a moving head disk, used to store data files [And00].
- Monitors are the basic synchronisation mechanisms for a real-time scheduler, inspired in a small process control system for an ammonia nitrate plant, implemented by P. Brinch-Hansen and P. Kraft in 1967 [Bri77].

Consequences

Benefits

- The complete set of concurrent or parallel software components are allowed to execute non-deterministically and at different relative speeds, each one acting as independently as possible of the others.
- Synchronisation is carried out by atomic or indivisible operations over the monitor.
- Each software component is able to execute the critical section within the monitor, accessing the shared variables in a safely and securely manner. Any other software component attempting to enter the monitor blocks, waiting for the servicing software component to finish its access.
- The shared variables keep their integrity during all the communication exchange.
- The use of monitors enforces and ensures the correct use of operations over shared variables.

Liabilities

- Mutual exclusion using monitors requires to be implemented at the compiler level. Commonly, the compiler associates a semaphore with each monitor. However, this implementation introduces potential delays when there is a commitment of the semaphore to a `wait()` operation, if necessary, on calling a monitor procedure.
- Mutual exclusion some times is not sufficient for programming concurrent systems. Conditional synchronisation is also needed (a resource may be busy when it is required to acquire it; the buffer may be full to put something into it,

and so on). Therefore, most monitor-based systems provide a new type of variable called condition variable. These condition variables should be incorporated as part of the programming, and they are needed by the application and the monitor implementation, managing them as synchronisation queues.

- A calling software component must not be allowed to block while holding a monitor lock. If a process must wait for condition synchronisation, the implementation must release the monitor for use by other software components and queue the software component on the condition variable.
- It is essential that before a software component leaves the monitor, its data is in a consistent state. It might be desirable to enforce that a software component can only read (and not write) the monitor data before leaving.
- The implementation of monitors based on semaphores has a potential problem with the `signal()` operation. Suppose that a signalling software component is active inside the monitor, and another software component is freed from a condition queue, and thus, potentially active inside the monitor. By definition, only one software component can be active inside a monitor at any time. A solution is to enforce that a `signal()` is immediately followed by exit the monitor, that is, the signalling process is forced to leave the monitor. If this method is not used, one of the software components may be delayed temporarily and resume execution in the monitor later.
- Monitors, as programming language synchronisation mechanisms, must be implemented with great care. Furthermore, programming the monitor methods or procedures must be carried out always aware of the constraints imposed by the mechanism itself.

Related Patterns

As part of the set of software patterns available, the Monitor idiom is related with the components of all those concurrent, parallel and distributed software systems in which monitors are used as synchronisation mechanism [POSA1, Lea97, POSA2, MSM04, POSA4].

Moreover, the Monitor idiom objective is to synchronise the software components within a communication sub-structure. As such, monitors can be used to implement synchronisation mechanisms for the Shared Variable Pipe pattern, the Message Passing

Pipe pattern, the Multiple Local Call pattern, the Shared Variable Channel pattern, the Message Passing Channel pattern, and the Local Rendezvous pattern.

The Monitor idiom simply represents a way to synchronise using shared variables of concurrent and parallel software components. It can be used as a synchronisation mechanism, just like the Semaphore idiom and the Critical Region idiom.

The Message Passing Idiom

Message passing is an inter-process communication and synchronisation mechanism between two or more parallel or distributed software components, executing simultaneously, non-deterministically, and at different relative speeds, on different address spaces of different computers of a distributed memory parallel platform. Message passing allows the synchronisation and data transfer of a message, mainly using two communication primitives: send and receive. These are the only way of allowing the exchange of data between the parallel or distributed software components. No assumptions can be made about when messages are sent or received [Hoa78, And91, Bac93, GBD+94, Bri95, Har98, HX98, And00].

Example

Parallel Virtual Machine (PVM) is a message passing library extension for C, Fortran, and C++ to exploit distributed, heterogeneous computing resources, capable of creating and executing processes on different computers of a distributed memory platform. In order to allow data exchange among distributed processes, PVM specifies several routines for sending and receiving data between processes [GBD+94, And00]:

- Sending a message comprises three steps: *(a)* a send buffer must be initialised by the routines `pvm_initsend()` or `pvm_mkbuf()`; *(b)* the message is ‘packed’ into the buffer, using some of the `pvm_pk*()` routines; and *(c)* the message is actually sent to another process by the `pvm_send()` or `pvm_mcast()` routines.
- Receiving a message requires two steps: *(a)* messages are received by a blocking or non-blocking routine, such as `pvm_recv()`, `pvm_nrec()`, or `pvm_precv()`, which place the received message into a receive buffer; and *(b)* the message is ‘unpacked’ using any of the `pvm_upk*()` routines.

For the present example, it is proposed the development of the synchronisation mechanism based on the Message Passing idiom for the example of a channel component for the Message Passing Channel pattern. The structure of the sender and the receiver are presented making use of a Java-like pseudo-code, in which message passing is allowed by synchronising the access to a socket. Here, a PVM version in C is used to express just a one way communication of the channel, by showing only a sending and a receiving in C.

In the general case, the Message Passing idiom in the C programming language, making use of PVM primitives for communication and synchronisation, has a form as the one shown in Figure 17 (an equivalent form for message passing in PVM is defined for Fortran, although it is not described here).

```
#include <pvm3.h>
...

int main(int argc, char**argv){
   int mytid, tids[n], me, i, N, rc, parent;
   ...
   me = pvm_joingroup(“name”);
   parent = pvm_parent();
   if (me == 0) {
      pvm_spawn(“process_name”,(char**)0, 0, “”,n-1,tids);
      ...
      pvm_initsend(PvmDataRaw);
      pvm_pkint(&N,1,1);
      pvm_mcast(tids,n-1,5);
   }
   else {
      pvm_recv(parent,5);
      pvm_upkint(&N,1,1);
   }
   pvm_barrier(“name”,n); // optional barrier synchronisation
   ...
   // operations on the data
   ...
   pvm_lvgroup(“name”);
   pvm_exit();
   return 0;
}
```

Figure 17.

Context

A parallel or distributed application is developed, in which two or more software components execute simultaneously on a distributed memory platform. These software components require exchanging data and synchronise in order to cooperate. Each software component is able to recognise and directly access its local address space, and to recognise remote address spaces of the other software components, which can be accessed only through I/O.

Problem

In order to allow data exchange between two or more parallel software components, executing on different computers of a distributed memory parallel platform, it is required to provide access between their address spaces, for an arbitrary number of read and write operations.

Forces

In order to apply the Message Passing idiom, the following forces are taken into consideration [Hoa78]:

- A set of parallel or distributed software components execute simultaneously, non-deterministically, and at different relative speeds, on different address spaces of a distributed memory parallel platform. All software components act synchronously and independently of the rest.
- Synchronisation is carried out by blocking or non-blocking, buffered or non-buffered operations which must be atomic or indivisible.
- Each software component is able to freely read and write its own address space, but should have the possibility of reading from or write to the remote address space of other software components, using I/O facilities. During these read and write operations, no other software component should be allowed to interfere.
- Data is transferred as messages. The values introduced into every message should keep their integrity during all the communication.
- The correct use of remote read and write operations should be enforced and ensured.

Solution

Use message passing for synchronising the access (read from or write to) the remote address space of software components, executing simultaneously on different computers of a distributed memory parallel platform. Message passing is a communication and synchronisation mechanism, mainly based on two communication primitives to support both synchronisation and data transfer: send and to receive. Data is transferred as a message. As it can be considered as a synchronous remote assignment operation, message passing is able to be used on a shared memory platform as well [Hoa78]. Message passing can be carried out with variations regarding synchronisation and buffering [Bac93, Har98, And00]:

- Message passing can be blocking (synchronous) or non-blocking (asynchronous). This refers that one software component, called sender or receiver, blocks or not during a communication, waiting or not for its counterpart.
- Message passing can be buffered or non-buffered, referring to the capacity of the sender or the receiver to provide a temporal storage for the message being transferred.

Structure

A sketch of the message passing concept is shown Figure 18, which allows at least two software components to transfer a message from a sender to a receiver, in the form of a remote assignment.

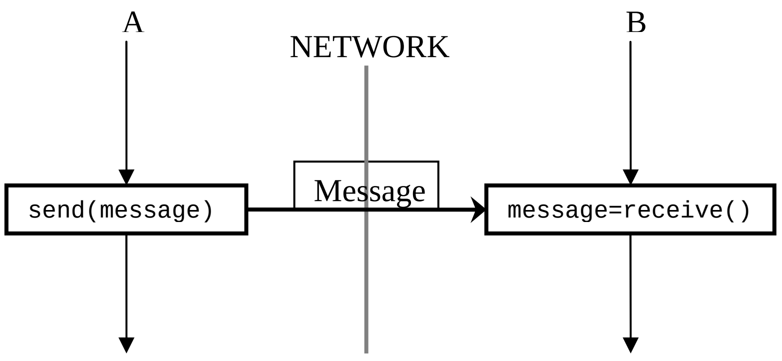


Figure 18.

If a programming language has defined message passing as the communication and synchronisation mechanism between parallel or distributed processes, their usage normally is similar to the interface Java-like pseudo-code shown in Figure 19 [Hoa78].

```
class MessagePassing{
   ...
   // declarations of local data
   private final Object message;
   ...
   // declaractions of procedures
   public synchronized void send(Object message);
   public synchronized Object receive();
};

class Sender implements Runnable{
   private Object data;  // data to be sent
   private MessagePassing mp; // reference to message passing
   ...
   public void run(){
      ...
      mp.send(data);
      ...
};

class Receiver implements Runnable{
   private Object data;  // data to be received
   private MessagePassing mp; // referente to message passing
   ...
   public void run(){
      ...
      data = mp.receive();
      ...
};

int main(){
   ...
   MessagePassing mp;
   Sender s;
   Receiver r;
   ...
   return 0;
}
```

Figure 19.

Every data is encapsulated within a software component. The message passing component, as an abstract data type (a class) is composed of declarations of `send()` and `receive()`methods. The data is read from and written to other software component's address space by these methods. In the main function, it is considered that software components can be mapped on different computers, and the message passing component is actually a distributed object between these two computers. The execution of the `send()` and `receive()`methods inside the message passing component is mutually exclusive among the software components that access it.

Dynamics

Message passing is commonly used as a simple communication and synchronisation mechanism, depending on a combination of its features, synchronous or asynchronous, and buffered or non-buffered, in the sender and/or receiver. Here, only some combinations are shown as possible behaviours of message passing.

Figure 20 shows a UML Sequence Diagram of the possible execution of two parallel or distributed software components, **A** and **B**, which communicate using a message passing component which encapsulates the send and receive primitives. Thus, a message can only be sent or received through explicit invocations to send or receive. Notice that for this diagram, the sender and the receiver are considered synchronous and non-buffered (both block whenever they find the send and receive primitives).

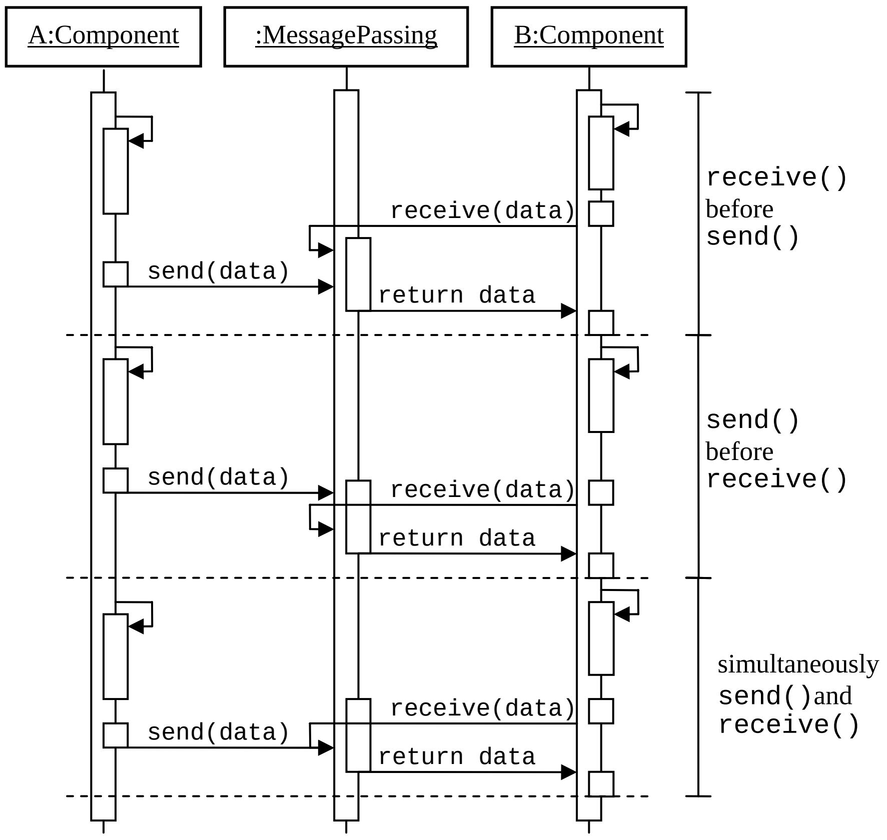


Figure 20.

Figure 21 shows another possible execution of two parallel or distributed software components, **A** and **B**, which communicate using a message passing, this time considering an asynchronous and non-buffered sender (the sender does not wait for the receiver) and a synchronous and buffered receiver communication (the receiver blocks to wait a message contained into a buffer).

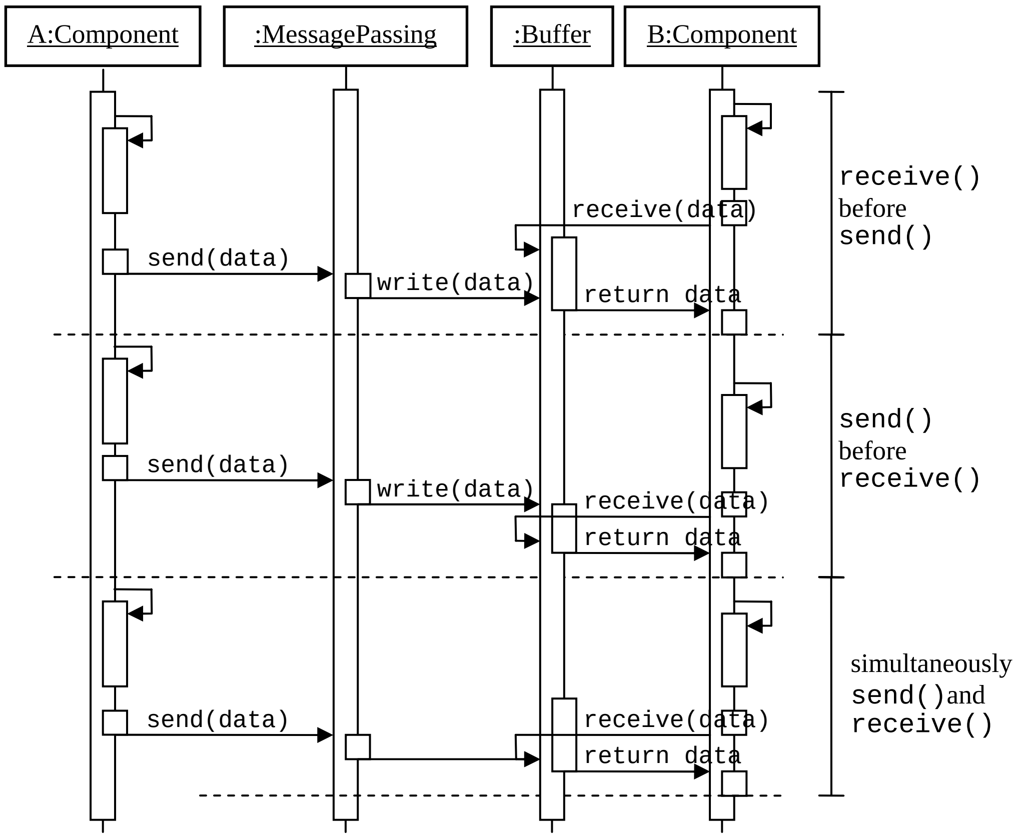


Figure 21.

Example Resolved

An implementation of the synchronisation mechanism based on the Message Passing idiom for a channel component for the Message Passing Channel pattern is proposed here, expressed in C using PVM routines, as shown in Figure 22.

```
#include <pvm3.h>
...
void sendToNext(int receiver, double data){
   ...
   pvm_initsend(PvmDataRaw);
   pvm_pkdouble(&N,1,1);
   pvm_send(receiver,1,1);
   return;
}

double receiveFromPrevious(sender){
   double data;
   ...
   pvm_recv(sender,1);
   data = pvm_upkdouble(&N,1,1);
   return data;
}

double channel(int sender, int receiver, double temperature){
   ...
   sendToNext(receiver, temperature);
   temperature = receiveFromPrevious(sender);
   ...
   return temperature;
}

int main(int argc, char**argv){
   int mytid, tids[n], me, i, N, rc;
   double temp[n]; // array of temperature data
   ...
   me = pvm_joingroup(“name”);
   parent = pvm_parent();
   if (me == 0) {
      pvm_spawn(“process_name”,(char**)0, 0, “”,n-1,tids);
      ...
   }
   for (i = 1; i < n; i++) {
      temp[i] = channel(tids[i],tids[i+1], temp[i]);
   ...
   // operations over local data
   }
   pvm_barrier(“name”,n); // optional barrier synchronisation

   pvm_lvgroup(“name”);
   pvm_exit();
   return 0;
}
```

Figure 22.

Known Uses

Message passing primitives are proposed as basic communication operations by C.A.R. Hoare [Hoa78], as a communication and synchronisation solution to the data exchange between processes executing in parallel. They represent the basic communication means within the programming languages used for distributed memory parallel platforms. Up to date, several authors make use of message passing as the basic inter-process communication construction in many programming languages [CM88, MCS90, And91, Bac93, Bri95, Har98, HX98, And00]. Some of the most widely known uses of message passing are:

- Message passing is used in an example for specifying and coding a telephone network in Occam [MCS90]. Occam is a parallel programming language [PM87, MCS90, Bac93, Bri95, HX98, And00] based on the Communicating Sequential Processes (CSP) specification by C.A.R. Hoare [Hoa78, Hoa85]. It has been used extensively as an example language for programming many parallel applications by several authors [Gree91, NHST94].
- The message passing organisation is used in an example of a remote file reader as the base for communication and synchronisation mechanisms using sockets in the Java programming language [And00].
- Message passing is used in several examples of the Message Passing Interface (MPI) standard, used for inter-process communication and synchronisation on a distributed memory platform [Bac93, HX98, And00, MSM04].

Consequences

Benefits

- A set of parallel or distributed software components execute simultaneously, non-deterministically, and at different relative speeds, on different address spaces of a distributed memory parallel platform. All software components act synchronously and independently of the rest.
- Synchronisation is carried out by blocking or non-blocking, buffered or non-buffered operations which must be atomic or indivisible.
- Each software component is able to freely read and write its own address space, but should have the possibility of reading from or write to the remote address space of other software components, using I/O facilities. During these read and write operations, no other software component should be allowed to interfere.
- Data is transferred as messages. The values introduced into every message should keep their integrity during all the communication.

Liabilities

- The correct use of remote read and write operations is not completely enforced and ensured.
- Depending on the synchronisation and buffering features, applications based on message passing may fall into a deadlock or a livelock.

Related Patterns

As a software patterns, the Message Passing idiom is related with the communication and synchronisation components of all parallel and distributed software systems executing on a distributed memory platform, in which message passing is used as synchronisation mechanism [POSA1, Lea97, POSA2, MSM04, POSA4].

The Message Passing idiom objective is to communicate and synchronise between software components. In such a way, message passing can be used to implement communication interfaces and synchronisation mechanisms for the Message Passing Pipe pattern and the Message Passing Channel pattern.

The Message Passing idiom simply represents a one-way communication to synchronise the actions between parallel and distributed software components. When it is paired, using a synchronous, two-way communication protocol between these software components, message passing can be used as a communication unit for implementing the Remote Procedure Call idiom.

The Remote Procedure Call Idiom

Remote Procedure Call is an inter-process, synchronous, bi-directional distributed communication and synchronisation mechanism between two parallel or distributed software components. These components execute simultaneously, non-deterministically, and at different relative speeds, on different address spaces of different computers of a distributed memory parallel platform. A remote Procedure call is carried out by a synchronous invocation, call, or request by a software component (acting as a client) of executing a function or procedure that belong to another software component (acting as a server) normally executing on another computer. This is considered as the only way of communication between the parallel or distributed software components. No assumptions can be made about when calls are issued [Bri78, And91, Bac93, Har98, HX98, And00].

Example

The Java programming language can be used to create and execute objects at different address spaces, executing on different processors of a distributed memory system. In order to allow communications between two distributed, remote objects, Java makes use of the Remote Method Invocation (RMI) as a remote procedure call communication and synchronisation mechanism between two remote objects. It is normally supported by two packages: `java.rmi` and `java.rmi.server` [Har98, And00, Smi00].

For the present example, it is proposed the development of the remote procedure call component, based on the Remote Procedure Call idiom between the root layer and the Multithread Server components for the Multiple Remote Call pattern. In this section, the structure of the component is presented as a Java-like pseudo-code, which allows synchronising the action of root and Multithread Server. In the general case, the Remote Procedure Call idiom in the Java programming language has a form similar to the one shown in Figure 23.

```
import java.rmi.*;
import java.rmi.server.*;

public interface RemoteServer extends Remote{
   public int read() throws RemoteException;
   public void write(int data) throws RemoteException;
}

class Client {
   int data;
   ...
   try{
      // Set the Standard RMI security manager (optional)
      System.setSecurityManager(new RMISecurityManager());
      // Get remote server object
      String name = “rmi://my_host:9999/server”;
      RemoteDataServer rds = (RemoteServer) Namig.lookup(name);
      ...
      // Write data to the server
      rds.write(data);
      ...
      // Read data from the Server
      data = rds.read();
      ...
   }
   catch(Exception e){}
}

class RemoteDataServer extends UnicastRemoteObject
                   implements RemoteServer {
   protected int data;

   public int read() throws RemoteException{
      return data;
   }
   public void write(int d) throws RemoteException{
      data = d;
   }

   public static void main(String[] args){
      try{
         // Create a data Server object
         RemoteDataServer rds = new RemoteDataServer();
         // Register name and start serving
         String name = “rmi://my_host:9999/server”;
         Naming.bind(name,rds);
      }
      catch(Exception e){}
   }
}
```

Figure 23.

In Java, using remote method invocation requires three elements [Har98, And00, Smi00]:

- A Java interface that extends Remote (defined in `java.rmi`), declaring headers for the remote methods, which throw remote exceptions.
- A Java class server that extends `UnicastRemoteObject`, implementing the methods in the interface, including protected data and definition of methods, as well as creating an instance of the server and registering its name with the *registry service*.
- One or more Java classes client that call the remote methods of the server. It has (optionally, depending on the compiler version) to set the standard RMI security manager, and then call the `Naming.lookup()` to get a server object from the *registry service*.

A registry service is a program that maintains a list of registered server names on a host. Normally, it is started in the server machine by executing `rmiregistry port`, where `port` is a valid port number.

Context

A parallel or distributed application is developed, in which two or more software components execute simultaneously on a distributed memory platform. Particularly, two software components require to communicate, synchronise and exchanging data in order to cooperate. Each software component is able to recognise the procedures or functions in the remote address space of the other software component, which is accessed only through I/O operations.

Problem

In order to allow communications between two parallel software components, executing on different computers of a distributed memory parallel platform, it is required to provide a synchronous access of calls between their address spaces, for an arbitrary number of call and reply operations.

Forces

Applying the Remote Procedure Call idiom has to take into consideration the following forces [Bri78]:

- Several parallel or distributed software components are created and execute simultaneously, non-deterministically, and at different relative speeds, on different address spaces of the computers of a distributed memory parallel platform. All software components act independently from the rest.

- Synchronisation is carried out by blocking call operations, which must be atomic or indivisible.
- Each software component is able to freely work on its own address space, but should have the possibility of accessing the procedures of a remote address space of other software components, using I/O facilities. During this access, no other software component should be allowed to interfere.
- Data is transferred as arguments of the function calls. The values introduced into every call and the results from performing the procedure should keep their integrity during all the communication.

Solution

Use remote procedure calls for synchronising and accessing the procedures or functions contained in the remote address space of software components which execute simultaneously on different computers of a distributed memory parallel platform. A remote procedure call is a communication and synchronisation mechanism, mainly connecting two software components, generically known as ‘client’ and ‘server’. The client calls or invokes the procedures of the server, which executes on a remote computer, as a request for service. The server processes the call, and returns a reply, which is sent to the client. The call is synchronous: the client blocks until it receives the reply from the server. Data is transferred as part of the call, in the form of arguments. The remote procedure call is considered as a bi-directional, synchronous *rendezvous* between client and server. Even though it has been originally defined for distributed memory systems, a remote procedure call message can be used as well on shared memory systems [Bri78].

Structure

Figure 24 shows a sketch of the concept of remote procedure call, which allows that two software components to have access to the procedures or functions contained in their respective address spaces, which execute on different computers of a distributed memory system. The invocation from the client to the server, in the form of a remote call, transfers data as arguments of the call.

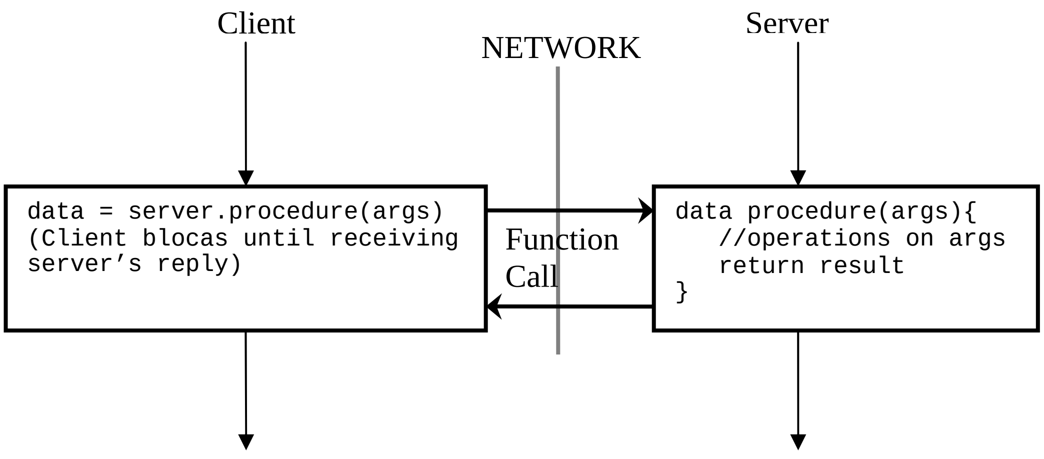


Figure 24.

If a programming language has defined remote procedure calls as a communication and synchronisation mechanism between parallel or distributed processes, their usage normally is similar to the interface Java-like pseudo-code shown in Figure 25 [Bri78].

```
class Server{
   // declaration of local variables
   type local_data;
   ...
   // declaration of exported operations
   type procedure(type formal_params){
      // operations on arguments
      return result;
   }
   ...
   // other local procedures and processes
   ...
   // inicial statements
   ...
}

class Client {
   // declaration of local procedures and variables
   ...
   type local_data;
   type actual_parameters;
   ...
   // Conection to server
   Server s;
   ...
   // function call
   data = call s.procedure(actual_parameters);
   ...
}
```

Figure 25.

Every procedure and data is encapsulated within a software component. The remote procedure call component is composed of declarations to allow the interaction between client and server. These declarations should finally contain message passing channels shared by client and server. So, the remote procedure call can be presented as implementing the interface shown in Figure 26.

```
interface RemoteProcedureCall {
    public abstract Object makeRequestWaitReply(Object m);
    public abstract Object getRequest();
    public abstract void makeReply();
}
```

Figure 26.

Client and server share a remote procedure call component for addressing. Each time a client wants to call the server, it calls the remote procedure call component, whose procedure `makeRequestWaitReply()` is used by the client to interact with the server, blocking until receiving a reply. On the other side, the server synchronises and communicates with the remote procedure call component using the procedure `getRequest()`. Once communication is established, the server processes the call and the data which comes with it, finally producing a result, which is sent back to the remote procedure call component using its `makeReply()` procedure. Once the remote procedure call receives the reply, it makes it available to the client, which unblocks and continues with its processing.

Dynamics

Remote procedure call constructs are used in several distributed systems as common synchronisation mechanisms. They are used particularly for communicating remote software components that normally act as client and server. Figure 27 shows a UML Sequence Diagram of the possible typical execution between of two parallel or distributed software components, Client and Server, which communicate using synchronous function calls.

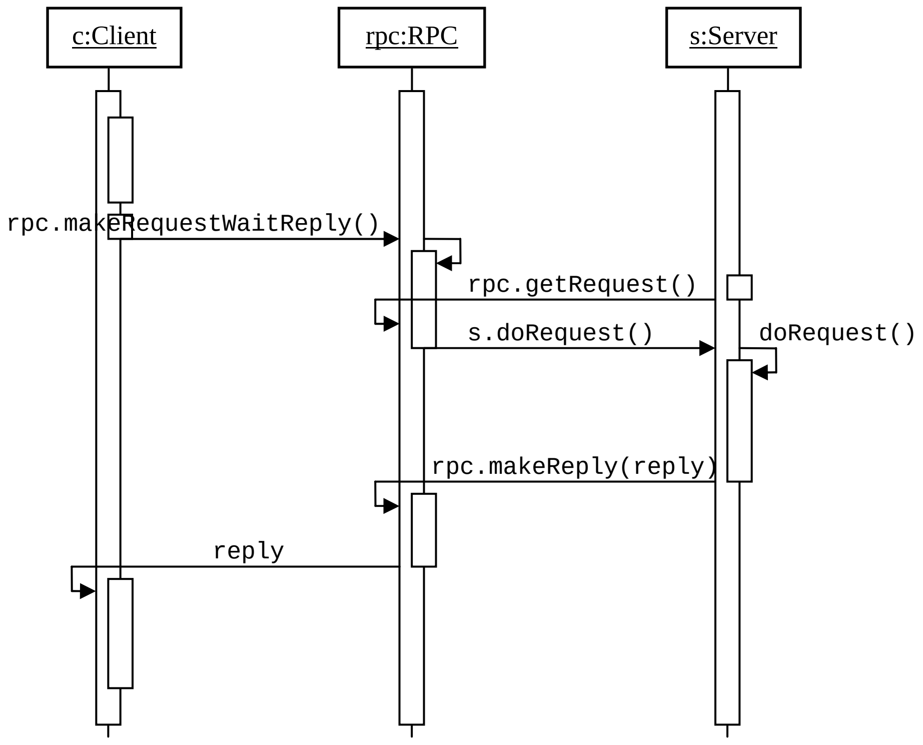


Figure 27.

The communication between software components starts when client invokes `rpc.makeRequestWaitReply()`. It is assumed that the remote procedure call component is free at such a moment, so it receives the call along with its arguments. The client keeps waiting until later, when the remote procedure call component issues it a reply. On the other side, the server invokes `rpc.getRequest()`, in order to retrieve any requests issued to the remote procedure call component. This triggers the execution of a procedure within the server, here `doRequest()`, which serves the call issued by the client, operating on the actual parameters of the call. Once the execution of this procedure finishes, the server invokes `rpc.makeReply()`, which encapsulates the reply, and allows to send it to the remote procedure call component. Once the remote procedure call has the reply, it makes it available to the client, which unblocks and continues. Notice that the remote procedure call acts as a synchronisation mechanism between client and server.

Example Resolved

An implementation for the synchronisation mechanism between the root layer and the Multithread Server components for the Multiple Remote Call pattern is presented using the `rmi` related classes in Java, as shown in Figure 28.

```
import java.rmi.*;
import java.rmi.server.*;

public interface RemoteProcedureCallInterface extends Remote{
   public abstract Object makeRequestWaitReply(Object m)
          throws RemoteException;
   public abstract Object getRequest()
          throws RemoteException;
   public abstract void makeReply(Object m)
          throws RemoteException;
}

class RootLayer extends Layer{
   private Object data;
   private Object result;
   try{
      // Set the Standard RMI security manager
      System.setSecurityManager(new RMISecurityManager());
      // Get remote server object
      String name = “rmi://my_host:9999/server”;
      RemoteProcedureCall rpc = (RemoteProcedureCall)
                                 Namig.lookup(name);
      // Generate the request to the Multithread Server
      result = rpc. makeRequestWaitReply(data);
   }
   catch(Exception e){}
}

class RemoteProcedureCall extends UnicastRemoteObject
                   implements RemoteProcedureCallInterface {
   protected Object data;
   protected Object reply;
   private MultithreadedServer ms;

   public Object makeRequestWaitReply(Object m)
                 throws RemoteException{
      // keep data for the call
      data = m;
   }

   public Object getRequest()
          throws RemoteException{
      // call remote method
      reply = ms.doRequest(m);
      return reply;
   }

   public void makeReply(Object m)
          throws RemoteException{
      //keep result for the reply
      reply = m;
   }
}
```

Figure 28.

Known Uses

Remote procedure calls are proposed as basic communication constructs by P. Brinch-Hansen [Bri78], as a communication and synchronisation solution to the remote interaction between processes executing on a distributed system. Along with message passing primitives, they represent a basic communication means within the programming languages used for distributed memory parallel platforms.

The use of client-server systems is pervasive along all network applications as the basic inter-process communication construction in many programming languages [And91, Bac93, Har98, HX98, And00]. Some of the most widely known uses of remote procedure calls are:

- Remote procedure calls are used as a communication and synchronisation mechanism in most Unix and Unix-like operating systems. Remote procedure call functions and procedures are part of this operating system's `rpc.h` library [And91, Bac93, HX98, And00].
- Remote procedure calls are used as a basic inter-process communication mechanism in the Ada programming Language [BD93, BW97, And00].
- Remote procedure calls are the base for all browsing activities of any Web browser.

Consequences

Benefits

- Several parallel or distributed software components can be created on different address spaces of the computers of a distributed memory parallel platform, and they are able to execute simultaneously, non-deterministically, and at different relative speeds. All of them are able to execute independently from the rest, although they synchronise in order to communicate.
- Synchronisation is achieved by blocking the client until it receives a reply from the server. For implementing remote procedure calls, blocking is more manageable than non-blocking. In general, remote procedure call implementations map very well onto a blocking communication paradigm.
- Each software component works its own address space, and issue calls for accessing the procedures of a remote address space of other software components, using network facilities. No other software component interferes during communication.
- Data is passed as arguments of the function or procedure calls. Arguments and results keep their integrity during all the communication.

Liabilities

- An implementation issue regarding remote procedure calls is the number of calls that can be in progress at any time from different threads within a particular software component. It is important that a number of software components on a computer of the distributed system should be able to initiate remote procedure calls and, in particular, that several threads of the same software component should be able to initiate remote procedure calls to the same destination. Consider for example, a server A, employing several threads to serve remote procedure call requests from different clients. Server A may itself need to invoke the service of another server, namely B. So, it must be possible for a thread on server A to initiate a remote procedure calls to server B and, while in progress, another thread on server A should be able to initiate other remote procedure calls to server B.
- It is commonly argued that a simple and efficient remote procedure call can be used as a base for all distributed communication requirements. This contrast with the approach of having different alternatives to select from. Alternatives of variations could be *(a)* a simple send for event notification with no requirement of reply; *(b)* an asynchronous version of remote procedure calls which requests the server to perform the operation, and keep the result so the client picks it up later; *(c)* a stream protocol for different sources and destinations, such as terminals, I/O, and so on.
- Some systems have real-time requirements when transferring large amounts of data, for example, multimedia or real-time systems. It is unlikely that remote procedure calls are sufficient for these purposes.
- Many times, the overhead of marshalling (packing and unpacking data) is not needed for certain types of data, and thus, tend to delay the operation of the construct.

Related Patterns

The Remote Procedure Call idiom, as a software pattern, is related with all those communication and synchronisation components of all parallel and distributed software systems executing on a distributed memory platform, in which it is used as synchronisation mechanism [POSA1, Lea97, POSA2, MSM04, POSA4].

Here, the Remote Procedure Call idiom objective is to communicate and synchronise between remote software components. In such a way, it can be used to implement communication interfaces and synchronisation mechanisms for the Multiple Remote Call pattern and the Remote Rendezvous pattern.

The Remote Procedure Call idiom simply represents a two-way communication construct used to synchronise the actions between parallel and distributed software components. It can be considered as implemented by two one-way message passing communications, based on the Message Passing idiom, between two software components.

Summary

The goal of the present paper is to introduce to parallel software designers and engineers to Some Idioms for Synchronisation Mechanisms, providing an overview of the common synchronisation codes used in communication components of parallel programs. Their selection constitutes the third main step towards the Detailed Design and Implementation of a coordination of a parallel application within the Pattern-based Parallel Software Design Method.

Some Idioms for Synchronisation Mechanisms have the objective to include the classic synchronisation mechanisms in order to implement communication and synchronisation components. They describe available and coded synchronisation mechanisms which allow the communication as described by a particular Design Pattern for Communication Components. However, as an initial attempt to the creation of a Pattern System for parallel programming, the idioms here are not as complete or detailed as to be considered that they cover every synchronisation issue within a parallel program. In that sense, these Idioms must be related with other current pattern developments for concurrent, parallel and distributed systems, by several authors.

The Idioms for Synchronisation Mechanisms are presented here along with the guidelines on their classification and selection, in order to help the software designer with deciding which synchronisation code potentially solve a given communication problem. The idioms described here are: the Semaphore idiom, the Critical Region idiom, the Monitor idiom, the Message Passing idiom, and the Remote Procedure Call idiom.