9. Concurrent programming

In this chapter we take look at the programming technique called Concurrent programming. We shall begin with a quote:

> *Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems. Implementation of parallelism is a topic of computer systems (hardware and software) that is essentially independent of concurrent programming. Concurrent programming is important, because it provides an abstract setting in which to study parallelism without getting bogged down in the implementation details'.*
> *(Ben-Ari, 1982)*

There are two points especially worth noting in this quote. First, we talk about **potential parallelism**. We must be prepared for **synchronisation and communication problems**, as described at the end of the sentence. But, depending on the system and the scheduling method, such things may not occur. If we have a static scheduling or there are no shared resources we may not need to be aware of the concurrent programming techniques. But, as shown earlier, such systems are often not very productive.

Second, note the **abstract setting** for studying parallelism, **without getting bogged down in the implementation details**. We try to raise the abstraction level and find some general principles which can be applied to virtually any environment. In this chapter our target is not to get bogged down!

9.1 Processes

At the beginning of the course we defined tasks and jobs. Every now and then the word **process** has also been mentioned during the course. Here we connect these three together.

One process encapsulates thread of execution (thread of control). A process in modern operating systems can embody many threads of controls. Process is dynamic, program is static. Concurrency implicates potential parallelism: the concept of a process can be found in

- operating systems
  - Windows ☺
- programming languages
  - modula-2,
  - Ada,
  - Java

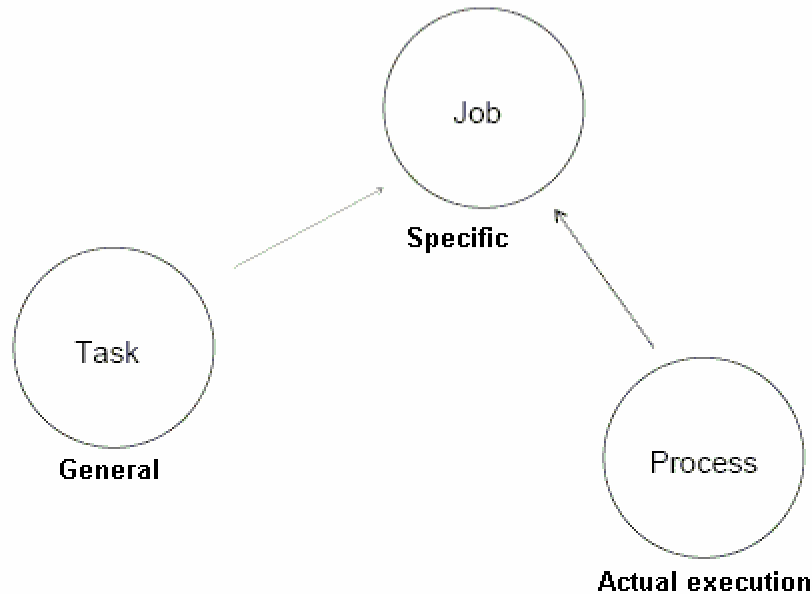

Figure 9.1 Task – Job - Process

Figure 9.1 illustrates the relation of task and process to a job: When we have a piece of code (=task) and the processor resources to execute it (=process), then we have a job which is doing something useful. There may various instances of a single task executed by various instances of system processes. So when our focus turns to the question “how to synchronise processes” we are actually looking at the question “how to synchronise the instances of our tasks handed to the system for execution”.

Consider the following situation: We have two processes, P1 and P2, which both use variable, say, v. They want to execute the following pieces of code:

P1	**v := 1;**
P2	**v := 0; v = v + 1;**

If these two pieces of code are independent, the results for both is the same, i.e. for both give v =1. However, if programs are executed concurrently, the results of P1 and P2 may be different. As follows:

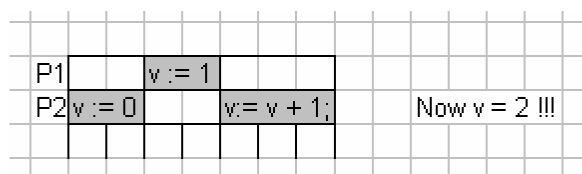

Figure 9.2 Concurrent execution

This kind of ”interleaving” (we’ve been talking about all the time) makes it difficult to deal with global properties from the local analysis. Obviously there should exist some method for exclusive access of the variable `v`. The big picture is quite similar to the situation where we had the resource R: One task (P2) should have exclusive access to the resource and other ones (P1) should be blocked until they can safely use the resource. A possible but a bit heavy method would be to create a fake (virtual?) resource R with the appropriate semaphore/mutex object, and use it to gain exclusive access to variable `v`.

(As figure 9.2 suggests, we’ve silently assumed that access to the memory is atomic: A task switch cannot occur while an assignment is going on.)

When getting concurrent on the program code level, there are two basic schemes often used for communication between the processes:

- Messages
- Shared variables

We shall study both of these in this chapter.

9.3 Concurrency in Programming Languages

Some programming languages contain features for supporting concurrency. Examples from these will be seen soon, but first we’ll take look at the pros and cons of such features.

On the **pro** side we may find the following facts:

- More readable program.
- More portable program.
- No strong requirements of operating system in embedded systems.
- Compiler can (in principle…) analyse faulty interactions between processes.

On the **con** side we have:

- Different languages have different models of concurrency (may be difficult to combinate programs).
- It may be difficult to implement the model used in a specific language to a specific operating system.
- There are standards of operating systems that makes program portable (=support for porting exist on the OS side, too).

The concept of process is implemented in a different way in different programming languages and operating systems. We’ll take a quick look at these features, since they have a huge effect on the interprocess communication.

9.3.1 Properties of a Process

The “concurrency potential” of a process may depend on several of its properties. What is the organisation of the processes in the system and how processes may be started and terminated. The following lists provide a crude “roadmap” for navigating in the process type jungle.

Structure and level of a process may be defined as follows:

- Static: the number of processes is given at the compilation time
- Dynamic: the number of processes can be changed at runtime
- Nested: processes can be defined at arbitrary level in program code, e.g. a process can be defined in a process
- Flat: processes can be defined only at the highest level of program code

Termination of a process may vary:

- The program code of the process ends (`END.`)
- Suicide: process executes a ”self-termination” statement (`abort()`)
- Abortion: by explicit action of an another process (`kill -9`)
- An exceptional event (exception) (`throw Xxxx…`)
- Never: process runs an infinite loop (`while (true) {…`)
- When there is no more need for the process (?)

The parent/child relation of a process may contain the following properties:

- guardian/dependant
- active/reactive/passive
- protected
- server/client
- process/thread

The lifespan of a single process may be presented with a **state diagram** of a process (Figure 9.3).

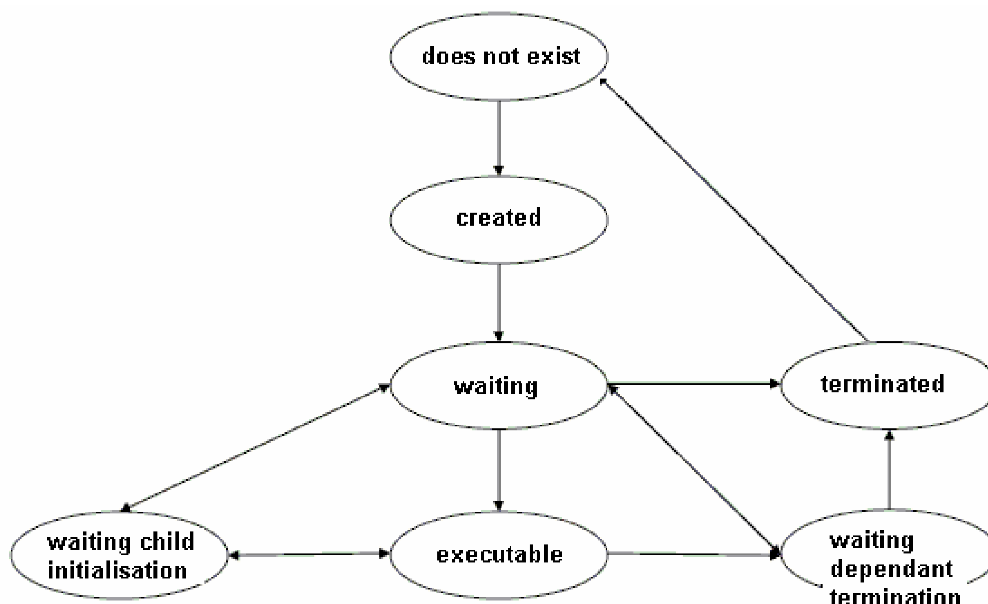

Figure 9.3 Process state diagram

9.3.2 How to Get Concurrent?

When there are various processes co-existing in the same system, their relation may be arranged as **co-routines** or on "f**ork and join**" basis. We'll take first a look at the latter one. Both scenarios also have their own methods to guarantee proper termination of the co-existence.

When a new process is created via a fork operation (`pid = fork();`), both the original and the new routine are run simultaneously. In UNIX the whole image of the caller is copied, only PID and PPID are different. The pid value returned by the fork is usually used for determining "who is who": For the child process the value is zero, for the parent the pid of the child is returned. As follows:

```
#include <unistd.h>
int main(void)
{
     pid_t pid = 0;
     int status = 0;
     pid = fork();
     If (pid == 0) {
          /* I am the child */
          doTheChildStuff();
     } else {
          /* I am the parent */
          doTheParentStuff();
          status = join();
     }
     return 0;
}
```

Exercise 9.1: Why it is useful for the parent to receive the child PID?

The parent, in general, cannot be sure about when the child is going to terminate, so the `join()` call is built to handle both conditions shown on figure 8.4.

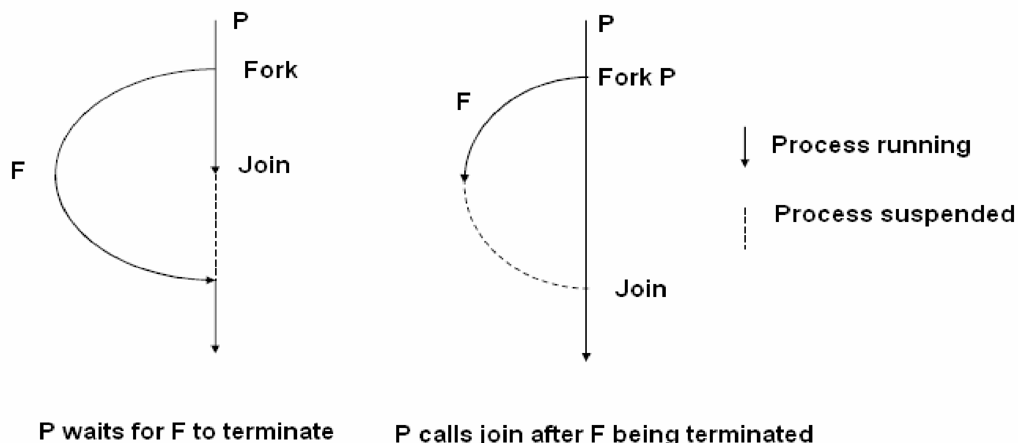

Figure 8.4 Fork and Join

Cobegin and **coend** is mentioned to be a simple and structured way to describe simultaneous running of program statements. (Nobody mentioned anything about how simple it is to implement…) The programming language may contain instructions similar to these:

```
cobegin
     S1;
     S2;
     S3;
     S4;
coend
```

This causes the four program statements S1-S4 being executed simultaneously. Figure 8.5 illustrates this.

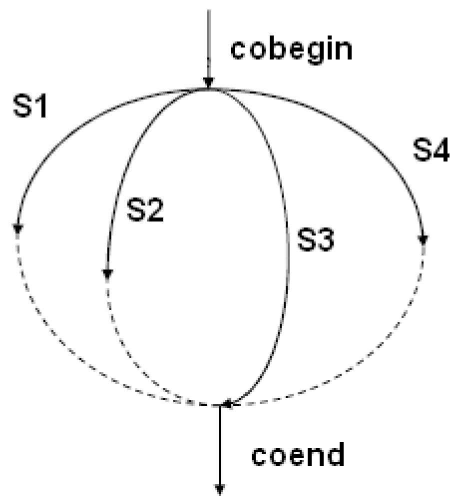

Figure 8.5 cobegin-coend

It is also possible to have **co-routines**. Essentially this is a set of subroutines which allow the control of execution being transformed explicitly between them.

8.4. Interprocess Communication

To get the processes run as concurrent items is good. Even better is to allow them communicate with each other. Here we describe with more detail some concepts we've already encountered so far.

Shared variables are an effective way to communicate between processes. As seen earlier, this may lead to trouble if two processes are allowed to update the same variable. Why does this happen? The operation `v := v + 1` can be implemented as three

separate machine language instructions: load the value `v` to register, increment the register, store the register value to memory. Like this (iX86):

```
MOV AX, [0013]
INC AX
MOV [0013], AX
```

(Note: “can be”, does not have to be. See PIC: `incf 0x13,F`)

To provide correct results, the update operation must be atomic. A common tool for this is to mark the “critical” code to be, surprise, a **critical section**. Access to critical sections is given exclusively to one process, and no other process may enter the section until it is no longer in use. Critical sections perform synchronisation via **mutual exclusion**.

A very simple (not to say crude) way to implement mutual exclusion is called **busy waiting**: Define a shared variable, `flag`, which indicates if the critical section is “occupied” or not. Until the P2 process sets the flag, the process P1 stays waiting for the resource to become available.

```
Process P1
     …
     While flag = down do
        Null
     End
     …
End P1
```

```
Process P2
     …
     flag = up
     …
End P2
```

(Here `Null` stands for wasting some time.) OK, we’ve reached an eternal loop: To provide protection for shared variable access we’ve defined another shared variable which should also be protected… :-) Seriously, the method is quite ineffective, the `Null` loop uses processor resources but is doing nothing useful.

The next code sample displays a generalisation of this scheme. If we have two (or more) such processes accessing the same critical section, then what is an appropriate method to implement the protocol in order to guarantee the mutual exclusion?

```
Process P
     do
          entry protocol
          critical section
          exit protocol
     loop
End P1
```

We present a solution called **Dekker’s algorithm**. (Unfortunately the author was not able to find detailed information about this particular Dekker. Descriptions of the algorithm can be easily found with your favourite search engine.)

Solution 1: Try the following:

```
Process P1                            Process P2
   loop                                 loop
     flag1 = up                         flag2 = up
     while flag2 = up do                while flag1 = up do
          Null                            Null
     end                                end
     // critical section                // critical section
     flag1 = down                       flag2 = down
     // non-critical section            // non-critical section
end P1                                end P2;
```

There are two flags used for the entry protocol. The idea is to use one of them to mark that "I'm going to go there" and the other one to check if the other one has already "gone there". This should prevent both of the processes executing the critical section at the same time. However, there's another problem here. Assume that the execution order of the instruction is as follows:

```
Process P1                            Process P2
   loop
                                           loop
     flag1 = up
                                           flag2 = up
     while flag2 = up do
          null
     end


                                           while flag1 = up do
                                             null
                                           end


     etc…
```

(Remember, we may have a (RT)OS which gives CPU time as fixed-length ticks. If P1 and P2 are running on the same priority level, anything can happen...)

What we have now: `flag1` is up, `flag2` is up, and nobody will be able to set them down. In other words, both processes stop in the null loop and neither will come out of it. This is a condition called **livelock**. It is different from the deadlock since both processes are running, but the overall result is the same: Nothing gets done. And there is no way to escape the condition.

Solution 2: Try testing before setting:

```
Process P1
   loop
     while flag2 = up do
          null
     end
     flag1 = up
     // critical section
     flag1 = down
     // non-critical section
end P1
```

```
Process P2
  loop
  while flag1 = up do
    null
  end
  flag2 = up
  // critical section
  flag2 = down
  // non-critical section
end P2;
```

This prevents the livelock, but it may cause both the processes to enter to the critical section, so this one may be even worse. (**Exercise 9.2** Why?)

Basically, the problem is that a process cannot set its own flag and test the other flag in one, atomic action. If there are two things to read/write, then the process to be executed may be switched between the two operations.

Solution 3: Use the turn variable:

```
Process P1
   loop
     while turn = 2 do
          null
     end
     // critical section
     turn = 2
     // non-critical section
end P1
```

```
Process P2
  loop
  while turn = 1 do
    null
  end
  // critical section
  turn = 1
  // non-critical section
end P2;
```

Now the variable turn must have value 1 or 2. If it is 1, then P1 avoids an eternal null loop and P2 cannot enter the critical section. And the turn variable cannot get value 2 before P1 has leaved the critical section. The same argument can be presented for P2, so the system appears to be safe. (The difference to the first, flag up/down –version is the fact that both processes now have their "own" value to read/write.)

But this, either, is a perfect solution: If, for some reason, P1 crashes before it reaches the critical section, then P2 is locked. Also if this arrangement is used the two processes need to operate at the same frequency. It is not possible to have P1 run three times while P2 is run only once. So more advanced solutions are still needed…

Solution 4: Gary L. Peterson published the following solution in 1981:

```
Process P1
   loop
     flag1 = up
     turn = 2
     while flag2=up and turn=2 do
       null
     end
     // critical section
     flag1 = down
     // non-critical section
   End
end P1
```

```
Process P2
  loop
    flag2 = up
    turn = 1
    while flag1=up and turn=1 do
      null
    end
    // critical section
    flag2 = down
    // non-critical section
    end
 end P2;
```

This should work. Now we can guarantee that both processes will complete the critical section in finite time. Also the process which starts the pre-protocol will, sooner or later, enter the critical section, regardless of the behaviour of the other process. Also the requirement for equal frequencies no longer applies. So having both the “personal flags” and the “common turn” appears to solve the problem.

The most amazing thing here is the year when the solution was published! 1981? I'd buy 1891, but only 25 years ago…! Seriously, this is relatively new research area: The original **critical section problem** was introduced by Dijkstra in the 1960's, and the Dekker algorithm was introduced in 1986.

Exercise 8.3 What if P1 crashes while in the critical section…?

We may conclude this mutual exclusion discussion with the following remarks: Implementation of synchronisation with only shared variables is difficult, since

- Busy-waiting technique is (relatively) complicated to use and (especially) test, since
- Testing can miss certain combinations of events (“interleavings”) which may cause problems
- Busy-wait is also inefficient
- One task not following the rules of entry/exit protocol may cause the whole system to crash.

So instead of running in a dummy loop we should halt the process until it may continue. **Suspend** and **resume** operations are used for this purpose. As follows:

```
Process P1
     // ...
     If flag = down do
       Suspend
     end
     Flag = down
     // critical section
End P1
```

```
Process P2
     // critical section
     flag = up
     resume P1
     // ...
End P1
```

So instead of circling around a `null` loop the process P1 suspends itself. P2, on the other hand, knows when P1 may continue and can resume it. No assumptions are made about the frequency the processes are running. The `suspend` and `resume` operations must be atomic.

Note that this example is actually a half of the whole implementation. Similar procedure should exist the other way around, too. The whole discussion presented above naturally applies to this version, the only change is that instead of wasting CPU time the `suspend` operation is used.

Exercise 9.4 What if P1 is not suspended when P2 resumes it?

Exercise 9.5 What assumption is made about the relationship between P1 and P2?

9.5 Semaphore in Synchronisation

Next we take a more detailed look at the semaphore/mutex system discussed earlier. The concept was introduced by Dijkstra in 1968. Using semaphores allows the synchronization protocol being simplified. Also the dummy `null` loops may be avoided.

Basically, a semaphore S is an (unsigned) integer value with two atomic operations, `wait()` and `signal()`. These work as follows:

- `wait(S)`: If S has a nonzero value, it is decremented by one, otherwise the caller is suspended until the value is greater than zero
- `signal(S)`: Increments the value by one.

Due to the atomicity no other process may interfere with these operations. (How this is achieved will be discussed a bit later.) Mutual exclusion is now simple to implement:

```
Process P1
   loop
     wait(S)
     // critical section
     signal(S)
     // Blaablaa…
   End
end P1
```

```
Process P2
  loop
     wait(S)
     // critical section
     signal(S)
     // Blaa2blaa2…
    end
 end P2;
```

At the beginning of the execution of the system, the value of `S` = 1. After the first `wait()` call `S` = 0, and possible other `wait()`call will cause the caller to be suspended until somebody calls `signal(S)`.

So one piece of each of the mentioned instructions will do the job. Considerable savings in code space compared to the earlier solutions. Of course, having two system calls may cause an increase in the execution time.

The question about how to make the operations atomic still reminds. Higher-level explanation to this is that the semaphores are implemented within the operating system. And since the (RT)OS also handles the task switches, it has the necessary control over the semaphore operations, as well as the suspend and resume mentioned earlier. The suspend operation may be implemented in such a way that the suspended process is waiting for an event. An example of such operation may be the increment of a semaphore, i.e. the `signal()` call.

In multiprocessor system the implementation will be more complicated. And even on a single-processor environment the external interrupts may need to be disabled. But since the semaphore operations are very simple (=they don't take much time) this is usually not a problem.

A lower-level explanation may be provided with the following example: Intel IA-32 instruction set has a special instruction CMPXCHG for low-level multiprocessor semaphore implementation. It is atomic operation (atomic instruction) and if prefixed with LOCK instruction, the hardware bus is reserved totally for the operation of the instruction.

CMPXCHG - Compare and Exchange

```
        Usage:  CMPXCHG dest,src  (486+)
        Modifies flags: AF CF OF PF SF ZF

        Compares the accumulator (8-32 bits) with dest.  If equal
        The dest is loaded with src, otherwise the accumulator is
        Loaded with dest.

                                Clocks                  Size
        Operands         808x  286   386   486          Bytes

        reg,reg            -     -     -     6            2
        mem,reg            -     -     -     7            2

        - add 3 clocks if the mem,reg comparison fails
```

LOCK - Lock Bus

```
        Usage:  LOCK
                  LOCK: (386+ prefix)
        Modifies flags: None

        This instruction is a prefix that causes the CPU assert
        bus lock signal during the execution of the next
```

```
instruction.  Used to avoid two processors from updating the
same data location.  The 286 always asserts lock during an
XCHG with memory operands.  This should only be used to lock
the bus prior to XCHG, MOV, IN and OUT instructions.

                      Clocks                  Size
Operands          808x  286   386   486          Bytes

none                2     0     0     1            1
```

These two may be combined as follows:

```
MOV EAX,#1
LOCK CMPXCHG SEMAPHORE,#0
JZ sem_closed
; semaphore open and reserved (closed)
```

To be precise, the **atomicity** of an action has been defined as follows (Lomte, 1977, Randel et al. 1978):

> *An action is atomic if the processes performing it are not aware of the existence of any other active process, and no other active process is aware of the activity of the process during the time the processes are performing the action.*

This definition continues by mentioning that no communication or state change may be observed during an atomic action. Atomic actions hence may be said to move the system from one consistent state to another consistent state.

Semaphores are effective but quite a low-level solution to the synchronisation problems. (Our target was not to get bogged down to the details, and we're examining machine language instructions…!) Incorrect use of semaphores may lead to deadlock or livelock condition. Wrongly programmed semaphore can cause the system to crash every now and then, making the error extremely difficult to find.

Consider the following:

```
Process P1                         Process P2
     wait(S1)                           wait(S2)
     wait(S2)                           wait(S1)
     // ...                             // ...
     Signal(S2)                         Signal(S1)
     Signal(S1)                         signal(S2)
end P1                              end P2;
```

Both processes pass the first `wait()` but then they will be suspended in the second `wait()`. And nobody's there to wake them up!

This is perhaps a good place to make a brief note about somebody who might be there to wake ‘em all up: The **watchdog** or the **watchdog timer** is a timing device, which resets the system if it is not refreshed within a predefined time interval. More complicated watchdog systems may also save debug information and/or put the system to a safe state (“limp home mode”). A description of the LonWorks technology Neuron C watchdog timer can be found on page 7 - 7 (the 215th page from the start) of the document http://www.echelon.com/support/documentation/manuals/devtools/078-0002-02G.pdf .

9.6 Communicating with Messages

So far we’ve discussed mostly about problems and solutions concerning synchronization and communication via shared variables. The other communication method mentioned earlier, messages, is described in this subsection.

Message exchange may be used for both synchronization and communication. A message needs to be send before it can be received. There are several ways which these two can be related to each other:

- **Asynchronic**: the sender proceeds immediately
- **Syncronic**: The sender proceeds after the message has been received. This is also called **rendezvous**.
- **Remote invocation**: The sender proceeds when the receiver has sent a response

(On some systems these may have names such as **unacknowledged** / **acknowledged** services.)

It is possible to build a synchronic / remote invocation service using asynchronic sevice:

```
Process P1
     send_async(msg_a)
     wait(ack_a)

     //...

     Send_sync(msg_b)
     Wait(rep_b)
     // use the response
End P2
```

```
Process P2
     wait(msg_a)
     send_async(ack_a)

     //...

     wait(msg_b)
     // process msg_b
     Send(rep_b)
End P2;
```

(This structure also illustrates well the name “remote invocation”: Essentially a remote procedure call is being made.)

Above we made no assumptions about how the message knows where to go. When sending something we need to know where to send it. There are two basic solutions to this addressing problem:

- A particular medium (channel, mailbox, pipe, whatever…) is used.

- The receiver and perhaps also the sender are identified in the message

On the second case one may talk about **symmetric** and **asymmetric sending** depending on whether both addresses or only the target address are included in the message.