Appendix C

On processes and threads: synchronization and communication in parallel programs

C.1 Sequential and parallel programs: main differences

In all our programs, there are some execution phases in which our processes are taking some time in an I/O operation or in a synchronization operation waiting for the system or another process to finish a task. During these phases, our programs are not progressing on their work, it is rather the operating system or some other process running on the computer the ones which are taking all the valuable CPU time. This is the reason why programs that abuse too much in using file read/write operations are so slow.

Processes that need to wait for an operating system service do usually leave the CPU free for other processes. This waiting is referred to as a process to be in the state of *blocked*. In the case of a sequential problem such as the EDA program, the whole algorithm is being executed as a single task, and the fact that our process is waiting one of these *blocking* operations makes the algorithm to be completely stuck during these periods of time. During the time spent waiting in the blocked state, another process takes ownership of the CPU and the execution of our algorithms does not progress at all.

Parallel programs provide a means for another process involved in the same calculation to make use of the CPU and continue with another part of the common job while our process is blocked (i.e. due to be waiting for an I/O or a synchronization operation). This happens both in single-processor and multiprocessor machines.

It is also important to note that when we write a sequential program (i.e. a non-parallel algorithm), our program will not be able under any circumstance of making use of more than a single CPU, even if we have more than one available. Only compilers that will convert automatically a sequential source program into a parallel one could help, but as explained before in the introduction these can reach these objective only in special cases. Therefore, if we want to use more than a CPU at the same time, we need to apply some kind of parallelization technique manually (writing specific instruction in the source code) when writing our program.

Parallel programs are based on dividing the job in smaller pieces so that more than

a single process work together[1] and collaborate between them to obtain the same result hopefully in less time. Doing it so, during the time that the I/O device takes to perform its task, the parallel program is able to make progress with another part of the job concurrently. Moreover, the fact of having a parallel program also allows us to make use of up to all the available CPUs at the same time.

Unfortunately, dividing the job in different processes that could be executed independently requires to rewrite some parts of the sequential program, or even to design a parallel version of it from scratch. This is not always an easy task, as several aspects need to be taken into account: the selection of the inter-process communication mechanism that will be used (which also consumes some extra execution time), the selection of the inter-process synchronization mechanism, and the identification of the parts of the sequential program that are candidate to be parallelized. The latter is important, as in most sequential programs there are always some parts that require to be done sequentially and cannot be executed in parallel. A simple example of the later is the moment when all the results of each of the working processes are to be gathered and a final result given. In many cases the preparation of the final result has to be done by a single process, and the rest of them cannot do anything during this period apart from waiting for more working instructions.

C.2 Processes and threads

In traditional operating systems, the concept of *process* is described as the instance of an executing program. In these, a process can only contain a running program where there is a single execution flow, that is, the program will be executing a single instruction at any time. This means in other words that in this traditional systems a process is represented as a single execution unit and that it only can be executed in a single CPU at any particular time.

In addition, the memory space assigned to a process in these traditional operating systems is defined to be *private*: no other process is allowed to access another process' memory space. In these systems no shared memory is available between processes, and therefore the only inter-process communication mechanism available for programmers is the use of message passing primitives –another possible mechanism is also the use of disk files, but this is very inefficient in terms of time performance. Only the operating system could make use of shared variables for communication between its internal parts.

In the recent years, some operating systems such as UNIX have included particular system calls[2] in order to allocate and to free shared memory from the user's address space in order to make possible for processes to share memory between them and to use shared variables (e.g. a shared buffer) as a inter-process communication mechanism. However, this mechanism is nowadays misused in UNIX in favor of more efficient and user-friendly mechanisms.

A next step forward in the operating systems history is the arrival of *multithreading*

[1] Term *process* is not the most appropriated in some cases, and *execution unit* should be used instead. However, the term *process* will be used in this thesis due to the fact that it is the one of choice in many parallelism books.

[2] *System calls* are standard programmable functions that any operating system provides as an interface for programmers to use system's resources and services. In the Windows family the term *Application Program Interface* or simply *API* is used instead. Any program running in any operating system can only access hardware by using these system calls, and all of them form the formal interface between the operating system and the executing programs. There are many types of system calls. Modern operating systems do also provide specific ones for inter-process synchronization.

operating systems. These systems included a new concept known as *thread*[3] that allowed a program to have more than an internal function running at the same time within the same memory space of a single process. As a result, a process could have more than a execution flow, and therefore more than a single line of the same program could be executing in parallel in two different CPUs simultaneously. Multithreading operating systems provide dedicated system calls for creating and managing execution threads as well as for creating and managing processes. In this systems, the memory address space of every process is still private for the rest of the processes, and as execution threads are attached to a unique process, they can use its whole memory space. In a multithread operating system two threads attached to a same process can use global shared variables within the process's memory space for communication. However, two threads attached to different process could not share any memory space and message passing is the only communication mechanism for them[4].

Nowadays, all commercial and general purpose operating systems are multithreading ones. When we write a sequential (i.e. non-parallel) program, the compiler will create an executable file that will execute in a traditional way, with a single thread or execution unit per process. However, we can use the specialized system calls that our operating systems provide, at a lower level than the programming language, and compile a program that will create more than a thread per process. Unfortunately, there is no a standard for system call interfaces applied to thread management that can be used in all the operating systems. Even within the UNIX family there are many different and incompatible interfaces available. In the recent years the POSIX standard's *pthread* interface library appears to be the most broadly used, and even an implementation of this library for the windows operating system family can be found on the Internet.

Whichever the selected parallel programming approach, either by making use of threads or processes, in both cases the communication and synchronization elements available are the same. The only restriction is that in the case of the processes there cannot be any communication mechanism based on shared memory, due to the privacy of the memory address spaces of them[5]. However, some books try to avoid distinction between processes and threads when speaking about parallelism, and the term *task* is used. In our case, we will use the term *process* to refer to both processes and threads, as this is done commonly in the literature.

C.3 Communication and synchronization between processes or threads

In any operating system, processes compete for accessing shared resources or they co-operate within a single application to communicate information to each other. Both situations are managed by the operating system by using synchronization mechanisms that allow exclusive

[3]Threads are also known as *light weight processes* or *subprocesses*. These are in fact execution flows or execution units within the same memory address space of a process. Operating systems that allow more than a thread to be running at the same time within a process (i.e. more than an instruction within the same process is executing at the same time) are called *multithreading operating systems*. Threads share all the global variables and functions within their process, and therefore they can use shared memory and variables for communication. However, synchronization mechanisms are required to avoid race conditions.

[4]Hard disk files could also be used for inter-process communication or inter-thread communication, but this mechanism is very inefficient. In parallel systems this communication solution is avoided, specially for the case of threads.

[5]Unless you use specific mechanisms that are only available in some operating systems, such as in UNIX.

access to shared resources and communication elements in a coordinated way.

In multithreading operating systems, the address space within any single process is shared between its threads, and therefore communication is performed by means of data structures within this shared memory space. This mechanism is quite efficient in terms of execution time, but on the other hand explicit communication mechanisms are required to ensure exclusive access to these shared structures. If shared buffers or queues are defined, and if exclusive access to them is provided, elaborated communication schemes such as the producer-consumer can be used. The client-server scheme can be regarded as a particular case of the producer-consumer where clients *produce* requests and servers *consume* them. When tasks (either processes or threads) communicate using a client-server approach, they do not share the address space. In this case, message passing primitives that provide implicit synchronization are required, and as a result they simplify the programming of the communication between processes.

In general terms, in parallel programming the term *communication* refers to the information exchange between processes. The communication between processes requires some kind of *synchronization*, either implicit or explicit, in the form of a norm or a protocol, that makes possible the information exchange between the communicating processes.

This section focuses on the different communication and synchronization mechanisms available for processes at a user level, as well as in the problems that programmers have to face when programming parallel applications.

C.3.1 The model of race conditions and critical sections

Let us consider two processes P_i and P_j that form part of a parallel program where these produce results for other processes not shown in this example. P_i and P_j store their results in a shared buffer buf. In order to find out which is the next free position in buf to store a result, there is also a shared pointer *inp*. If a process wants to store a result in buf, it has to call the following function:

```
void StoreResult(char item) {
    int p;
    ...
    p = inp;
    buf[p]= item;
    inp++;
    ...
}
```

This function seems to be correct, and no apparent problems can be noticed at a first glance. However, let us imagine the case in which the parallel execution progresses as follows:

```
          Pi                          Pj
    (1) p = inp;
                              (2) p = inp;
                              (3) buf[p]= item;
    (4) buf[p]= item;
    (5) inp++;
                              (6) inp++;
```

As we can see, this particular execution order will lead to storing the second value *item* at the same position of the buffer as the first one. P_j will overwrite its value over P_i's one. This problem is known as the *race condition*. Any program accessing shared resources can have this problem, however it may not be present in all the runs, as it depends on the order in which different processes execute their instructions. Any part of a program that can create race conditions is called a *critical section*.

Critical sections are present in general terms whenever a resource (i.e. a buffer or, simply, a variable) is shared among many concurrent processes. Programmers can use the operating system primitives to avoid race conditions. For this, it is important firstly that critical sections are identified in our programs. The programmer could afterwards use the appropriated system calls for communication or synchronization that will provide exclusive access to the critical section. Doing it so, the programmer can be sure that at most one single process can be executing a critical section at any time. In addition, the programmer can also use one of the many specialized libraries available for parallel programming: these libraries are installed just above the operating system, they hide the system calls interface, and usually they provide a simpler and more powerful way of avoiding race conditions.

C.3.2 Exclusive access to critical sections

Let assume that we have n processes, $\{P_1, P_2, ..., P_n\}$ where for each process P_i, $i = 1, \ldots, n$ there is a program section (critical section, CS) which accesses or manipulates shared information. In order to provide exclusive access to the CS, we have to ensure that when a process P_i executes the CS no other process can enter into it until P_i exits from it.

The way of solving this problem is to use a protocol to access critical sections, and for that purpose we will introduce two generic primitives, `enter_CS()` and `exit_CS()`, as shown next:

```
...
enter_CS()  /* if nobody in CS, continue, otherwise wait */

[the Critical Section ]

exit_CS()   /* now another process can enter the CS */
...
```

When the CS is free and some processes are waiting to enter it, the protocol has to specify which of them enters, forcing the rest to keep waiting. Usually, a First Come First Serve (FCFS) ordering is desirable, but the selection of the next process could also be based on the priorities of processes or in another aspect.

Critical sections are also very common in concurrent applications. Just as an example, we will use the producer-consumer example (Figure C.1) to illustrate the critical section problem and its possible solutions. This example was originally thought only for 1 producer-process and 1 consumer-process. A process is producing elements which are stored in a shared buffer calling a function `store_element()`. The full buffer condition is controlled using a shared variable `counter`. Analogously, a consumer process obtains elements from this buffer by using a function `obtain_element()` that also requires a critical section, therefore decreasing the counter in an unit. Apart from the access to the buffer, the producer-consumer example has another critical section on the access to the counter variable, which is also a shared

```
int counter= 0;

producer() {
    int element;

    while (true) {
        produce_element(&element);
        while (counter == N) NOP;
        store_element(element);
        counter=counter+1;
    }
}

consumer() {
    int element;

    while (true) {
        while (counter == 0) NOP;
        obtain_element(&element);
        counter=counter-1;
        consume_element(element);
    }
}
```

Figure C.1: Producer-consumer example with race conditions.

resource. The code in Figure C.1 has race conditions in the access to the buffer when it is either full or empty.

The solution to the problem of the exclusive access to critical sections is fundamental for the field of inter-process communication, as any other concurrent or parallel problem can be expressed by its means.

C.3.3 Conditions for critical sections

There are many possible implementations to the primitives `enter_CS()` and `exit_CS()`, but whichever implementation we choose it has to satisfy the following conditions as stated in [Dijkstra, 1965]:

Mutual exclusion. There cannot be more than a process executing the CS simultaneously.

No deadlock. No process blocked outside the CS can avoid any other to enter the CS.

Bounded waiting. A process cannot be waiting indefinitely for entering the CS.

Hardware independence. No supposition can be done about the number of processors or the relative processing speed of the processes.

Any protocol that implements the two primitives `enter_CS()` and `exit_CS()` has to fulfill these four conditions if it is to be regarded as a valid communication mechanism. In

order to compare the performance of two different implementations of this protocol, we will always have to do an initial assumption: the primitives `enter_CS()` and `exit_CS()` need to be atomic and are executed before entering and after exiting the critical section respectively.

C.3.4 Communication primitives

Inter-process communication solutions are normally classified following the synchronization method that they use, explicit or implicit. Doing it so, we have two main solution groups:

1. Using shared variables (it requires explicit synchronization)

   Active waiting :

   - Software: lock variables, Dekker's and Petterson's algorithms, Lamport's algorithm...
   - Hardware: interrupt inhibition, specific machine language instructions...

   Blocked waiting :

   - Basic primitives: sleep and wake up, semaphores...
   - High-level constructions: critical regions, monitors...

2. Using message passing (implicit synchronization)

Shared variables can be often mapped in memory. When this is the case, these are an adequate communication mechanism for synchronizing for instance execution-threads within a process. Disk files could also be used for communicating independent processes, but this solution is much less efficient. The message passing mechanism (explained in Section C.3.5) makes also use of shared objects for communication purposes, and its primitives guarantee exclusive access to these. Generally speaking, two processes communicate using a message passing mechanism, which is regarded as a very efficient and high level mechanism.

A.- Active waiting

Lock variables

If there is no specific synchronization mechanism to access a common resource, a possibility is to access critical sections by means of an active waiting algorithm that uses shared variables. These variables will be used to control the access to critical sections. But it is important to check the validity of the implementation, as the intuitive use of *lock variables* does not ensure exclusive access to the critical section, as shown here:

```
int lock=0;

Enter_CS: while (lock) NOP;    /* active waiting */
          lock= 1;

Exit_CS: lock= 0;
```

It is easy to check that this implementation of the `enter_CS()` and `exit_CS()` protocol does not satisfy the mutual exclusion property. The fact of requiring two separated instructions for both read the value of `lock` and activate it generates a new critical section in these

two instructions: another race condition is created between the first and the second line of `enter_CS()`.

The only existing software solutions are complex and computationally very time consuming –the algorithms of Peterson and Lamport are possible solutions, and they can be found for instance in [Silberschatz et al., 2000].

Inhibition and activation of interrupts

Some operating systems also provide the inhibition and activation of interrupt levels as a mechanism to solve the critical section problem. This solution is only valid for single-processor computers as it is highly restrictive. This method is based on the idea of stopping interrupt signals to reach the processor, and therefore on inhibiting the system from preemption[6] the process that is executing. It is implemented as follows:

```
Enter_CS: s = inhibit()  /* Inhibits all the interrupt levels */

Exit_CS:  activate(s)    /* Activates interrupts */
```

This mechanism of manipulating the interrupt levels is very restrictive and could create many additional drawbacks. Firstly, it does not satisfy the hardware independence condition, as it is dependent on hardware availability for this type of mechanism. On the other hand, if we use such a mechanism for a critical section that requires a long execution time, absolutely all the interrupts will be inhibited for a long time, leading to additional problems depending on the type of interrupts that should have been activated on that time. The inhibition of interrupts does not distinguish between types of interrupts, and as a result even very critical interrupt levels that would not threaten the integrity of the critical section will be inhibited.

B.- Blocked waiting

Every active waiting mechanism has some important drawbacks due to the fact that the protocol keeps on waiting by consultation to the lock variable. The main problems that this type of mechanisms carry are:

- During the waiting time, the CPU is in use preventing other processes to have access to it.
- A bad scheduler can lead all the processes to block to each other[7]: let us consider as an example that an operating system using two priorities for its processes, high (H) and low (L) levels, and two processes, P_H of high priority and P_L with low priority. Then if P_L is in the critical section and P_H wants to enter it, the scheduler will always decide to choose P_H giving P_L no chance to keep with its execution and therefore keeping P_H in active waiting indefinitely. This situation constitutes what it is called a *deadlock*. Also, this particular example shows that the active waiting mechanisms behave differently depending on their implementation in the operating system.

[6] *Preemption* is an operating system concept related to process planning. It is the result of a process being stopped in its execution on the CPU and another being executed in its place. This happens for instance when a higher priority process than the one in the CPU is ready to continue its execution: as a result the higher priority process replaces the lower one in the CPU. If lower priority processes are executing a critical section and if interrupts and inhibited no other process will change its status and thrown it away from the CPU. The interested reader can find more information in [Silberschatz et al., 2000].

[7] A scheduler is the internal routine of the operating system that selects will pass next to occupy a freed CPU.

The alternative for the active waiting is to block a process when it wants to enter the critical sections that it is busy. This section reviews some of the basic techniques for blocked waiting.

Sleep* & *wake-up

This two primitives allow to block a process P_i when executing `sleep()`. Another process P_j will wake up the sleeping process by calling the primitive `wake-up(` P_i `)`.

```
Enter_CS (for a process Pi):   /* if CS busy */ sleep();

Exit_CS (for a process Pj):    wake-up(Pi)
```

These two primitives are not able to provide exclusive access to critical sections by themselves, but they are to be used by any mechanisms that implement primitives for the protocol to access critical sections.

As with the rest of mechanisms reviewed so far, the sleep and wake-up primitives do also have some problems. If the process P_i is not sleeping, the primitive `wake-up(Pi)` will not have any effect, and therefore its call will be ignored and forgotten. Unfortunately, this situation can create race conditions in schemes such as the producer-consumer as it is written in Figure C.2 applying sleep & wake-up primitives, where the condition of full (or empty) buffer and the action of sleeping do again constitute a new critical section. This problem can be seen in the following example: if the producer has detected the full buffer function and just before executing the `sleep()` primitive if it is preempted from the CPU, the consumer could consume an element and execute the `wake-up(producer)` primitive before the system returns the control to the producer, and therefore the posterior call will be ignored. When the producer reaches again the CPU, it will not check again whether the buffer is full or not, and as a result it will go to sleep with no possibility of the consumer to execute again the waking up primitive. This situation constitutes again a deadlock state.

Semaphores

A more general abstraction is the *semaphore* [Dijkstra, 1965], which is built over the sleep & wake-up primitives. Semaphores keep record of all the sleeping and waking up primitives, waking up a single blocked process later when it is required.

Every semaphore contains a queue of processes that are blocked waiting to a waking-up event to arrive, as well as a counter for waking up signals already received. These features allow us to use semaphores both for inter-process synchronization as well as for managing resources. Semaphores are used for these purposes inside the operating system and at user level. Once defined a semaphore `s`, two basic atomic operations are allowed:

```
wait(s)   --also: p(s), down(s), sem_wait(s) ...

signal(s) --also: v(s), up(s), sem_post(s) ...
```

When a process executes the primitive `wait(s)` over a semaphore s, if the counter associated to s is strictly bigger than zero the process will continue and the counter will be decreased in a unit; otherwise, the process will be blocked (i.e. sent to sleeping by executing `sleep()`). When a process executes the primitive `signal(s)` the counter is incremented by one unit; however, if there are blocked processes, one of them will also be awakened.

```
int counter= 0;

producer() {
    int element;

    while (1) {
        produce_element(&element);
        if (counter == N) sleep();
        store_element(element);
        counter=counter+1;
        if (counter == 1) wake-up(consumer);
    }
}

consumer() {
    int element;

    while (1) {
        if (counter == 0) sleep();
        obtain_element(&element);
        counter=counter-1;
        if (counter == N-1) wake-up(producer);
        consume_element(element);
    }
}
```

Figure C.2: Example of the producer-consumer example solved with sleep & wake-up primitives. The solution proposed here is only valid for one producer and one consumer.

Initially a value is assigned to the semaphore's counter by means of an initialization primitive: `ini_semaphore(s,value)` which assigns the `value` to the counter associated to `s` and also inititializes the associated queue as an empty one.

Semaphores are a very common synchronization mechanism, and they are available in most of the modern operating systems. These systems provide the *wait*, *signal* and *ini_semaphore* operations in the form of system call primitives that can directly be used in our programs. The next section provides a revision of the many uses of the semaphores in order to show a general idea of their usefulness.

Use of semaphores. Semaphores can be used for instance for long term mutual exclusion management, as they only affect the processes willing to access the critical section at a particular period of time, without having influence on the rest of processes running on the system. The critical section problem can be solved with semaphores by initializing the semaphore to one[8], and then used in the following way:

```
Enter_CS: wait(mutex); /* if (CS busy) the process is blocked*/
```

[8]These semaphores initialized to one that are used for addressing the mutual exclusion property are often called *mutex* or *binary semaphores*.

```
struct semaphore_t mutex, holes, items;

ini_semaphore(mutex, 1); ini_semaphore(holes, N);
ini_semaphore(items, 0);

producer() {
    int element;

    while (1) {
        produce_element(&element);
        wait(holes);
        wait(mutex);
        store_element(element);
        signal(mutex);
        signal(items);
    }
}

consumer() {
    int element;

    while (1) {
        wait(items);
        wait(mutex);
        obtain_element(&element);
        signal(mutex);
        signal(holes);
        consume_element(element);
    }
}
```

Figure C.3: Example of solving the producer-consumer problem using semaphores.

```
Exit_CS:  signal(mutex);
```

Semaphores are often used as a more general synchronization tool, and they can also be applied for assigning shared resources between variables. Given n units of a shared resource that any process can use, the semaphore R for the resource is initialized,

```
ini_semaphore(R, n);
```

and a process will use the resource following the next protocol:

```
wait(R);

    /* use the shared resource */

signal(R);
```

Please, note that $n = 1$ is the particular case of the mutual exclusion. The producer-consumer problem, in which a process produces items and store them in a shared buffer while the consumer process reads them from the buffer and consumes them, is taken as an example of the use of semaphores for mutual exclusion and resource management (Figure C.3). A semaphore is used for the mutual exclusion, the one called `mutex`, in order to solve the exclusive access on the operations for storing and obtaining elements on the shared *buffer* resource[9]. Two other semaphores are also defined, one for blocking the producer when the buffer is full (*holes*) and another for blocking the consumer when the buffer is empty (*items*). Note that the initialization of all the semaphores is done in the example of Figure C.3. It is also worth saying that this solution of the producer-consumer problem is also valid for the more general case of having n producers and m consumers, $n, m > 1$.

Semaphores can also be used for communication purposes. For instance, in a client-server scheme the client will use a semaphore to synchronize with the end of server work, while the server will use a semaphore as a event for waiting clients' requests.

Finally, there are other more complex and typical synchronization problems such as the problem of the readers-writers that can be solved using semaphores –more information about this problem can be found in [Stallings, 2000] and [Andrews, 1991].

Implementation of semaphores. A semaphore is a predefined structure that allows only one of the three basic operations `wait`, `signal`, and `ini_semaphore` shown before. Once again, the implementation of these primitives has to be atomic, although in the following example the specification of critical sections has been omitted. The semaphores as well as their basic operations can be implemented in C language as follows:

```
struct semaphore_t {
  int counter;
  struct queue q;
}

void ini_semaphore(struct semaphore_t *sem, int val) {
  sem->counter= val;
  ini_queue(sem.q);
}

void wait(struct semaphore_t *sem) {
  if (sem->counter == 0) {
    put_process_in_queue(sem->q, my_process);
    sleep(my_process);     /* sleep the process to make it wait */
  }
  else --sem->counter;
}

void signal(struct semaphore_t *sem) {
  if (empty_queue(sem->q)) ++sem->counter;
  else wake-up(first(sem->q)); /* wake up the first in the queue */
}
```

[9] A more efficient implementation would use another short-term mutual exclusion mechanism such as lock variables instead of mutex semaphores for this simple case.

Barriers

Barriers are synchronization elements used in parallel programming to synchronize a finite set of n processes between them. A barrier will stop the first processes arriving to the synchronization primitive until all the n processes have executed the call. The generic synchronization primitive is executed as follows:

```
struct barrier_t bar; int n; ... barrier (bar, n) ...
```

where `bar` is the name of the barrier that we are applying and `n` is the number of processes that want to synchronize with it. The necessary condition continue the execution of a process executing the primitive `barrier` is that n processes have called this function.

Barriers are available in many parallel programming libraries and standards, and due to its simplicity, this mechanism is used very usually for synchronization purposes. When an operating system or a selected parallel programming library does not provide barriers, these can be easily implemented using either semaphores or lock variables.

C.3.5 Message passing

In all the solutions described so far we have assumed the existence of shared elements between execution threads (shared memory or shared files) where the shared variables for communication would be mapped in memory. In case of using threads within a single process, the use of global variables in memory is direct and intuitive, but they require explicit synchronization mechanisms using one of the many mechanisms already described.

Another alternative for inter-process communication, which is also suitable when these processes are executing on different computers communicating through a network, is to use *message passing* primitives. These message passing primitives make use directly or indirectly of specific communication elements[10] that behave as First In First Out (FIFO) queues. Message passing primitives ensure exclusive access to the communication channel, and therefore their use does no require explicit synchronization to be programmed when reading or writing a message.

The two generic communication primitives available in any message passing system are defined as follows:

```
send    --also called: write_message,
receive --also called: read_message,
```

Message passing communication types

There are many communication types available when message passing:

- direct communication

  ```
  send(process_id, message)
  receive(process_id, message)
  ```

[10]These communication elements receive many names in the literature: communication links, channels, or communication ports. In this dissertation we call them communication channels or just *channels*.

In this communication type, processes that want to communicate to each other need to know their id numbers, although sometimes an extension of this is used so that a process can send messages to any receiver or to receive messages from any sender (*broadcasting*):

```
send(ALL, message)
receive(ANY, message)
```

- Communication using mailboxes

```
send (mailbox, message)
receive (mailbox, message)
```

A mailbox is a named FIFO-type communication channel. The processes need to know the mailbox's name if they want to communicate to each other, but they do not need to know the other's id number. In general terms, there will be many processes sending messages to the mailbox and also many other reading from it.

In this section we will consider a message to be a character string. This is typically the case also in networking protocols. The length of the message can be either fixed or variable depending on the message passing implementation.

Synchronization with message passing primitives

In message passing, it is important to understand how the implicit synchronization behaves when two processes are communicating. The primitive `receive` will block the process if the communication channel is empty of messages, and otherwise the first message will be returned and the process will continue its execution. The behavior of the `send` operation can be different from an implementation to another, and this is dependent on many characteristics of the communication channel: alternatives such as being buffered or not buffered, or to be synchronous or asynchronous, will determine the behavior of the two primitives at a great extent. Another of these main characteristics is the *capacity*. The capacity of the communication channel determines the way of synchronizing the processes when accessing the mailbox for writing. Depending on this characteristic the communication channels can have:

Unlimited capacity: this is the ideal communication channel. The sending process will never, under any condition, be blocked.

Limited capacity: the sender-process will be blocked only when the communication channel is full.

Null capacity: this possibility makes sense only in direct communication. As no message can be stored, the two processes have to synchronize to each other every time they want to communicate: the first of them that is ready for sending or receiving is blocked until the other is ready for the opposite operation. This mechanism is also known as *rendezvous*. This procedure is similar to the use of barriers by two processes, but in this case the communication of information is also included.

```
producer() {
    int element;
    char message[50];

    while (1) {
        produce_element(&element);
        compose_message (&message, element);
        send(MBX, message);
    }
}

consumer() {
    int element;
    char message[50];

    while (1) {
        receive(MBX, &message);
        decompose_message (message, &element);
        consume_element(element);
    }
}
```

Figure C.4: Example of the producer-consumer scheme using message passing.

Except from the case of null capacity, the message passing mechanism leads to an asynchronous model for processes to work. This means that the sender can usually continue its execution even if the receiver has not still received the message. As an example of using message passing primitives, Figure C.4 shows how to solve the producer-consumer problem using a mailbox called `MBX`.

C.3.6 Communication and synchronization paradigms

The exclusive access to critical sections is a fundamental problem in inter-process communication and synchronization. The previous sections have reviewed the basic mechanisms that can be used to solve this problem in our programs. The paradigms described on this section are typical communication and synchronization situations (the producer-consumer one is just an example of them) that can be solved by means of these basic mechanisms, once they usefulness on the critical section problem has been proved. These paradigms are different from the ones introduced in Section 5.2.1 in the sense that the ones introduced here are typical basic applications that are used for measuring the performance of the different communication and synchronization mechanisms for different types of problems. Some other more complex paradigms are also briefly described in this section.

The typical communication and synchronization paradigms are the following:

Readers & **writers** : The access of shared information in disk files, databases, and other types of shared resources is usually asymmetric, that is, most of the accesses are for reading and much less for writing. In these cases, the definition of critical sections for exclusive access is too restrictive and inefficient, as in most of the cases n reader

processes could be allowed to be accessing the critical section at the same time without restrictions. Only when the information needs to be updated is required that the critical section is free before the writer enters it. This problem can be solved in different ways using for instance semaphores. This solution is shown in [Stallings, 2000].

Assigning multiple resources : The basic problem of assigning n resources has been introduced in Section C.3.4, as semaphores can be applied in an easy and simple way. When the resources are of different types assignments can be done to different processes at the same time, the problem acquires a very complex nature, and usually falls on unbounded waiting and deadlock problems. There are many solutions for this problem to avoid the deadlock state that can be found in [Dijkstra, 1965].

Client-server : This paradigm is a particular case of the producer-consumer problem, and it is applied in many situations (i.e. general applications, inside the operating system, in networking...) to control the use of resources. A process plays the role of the server, and it centralizes the requests to access a resource (or many resources) requested by the client-processes. In the producer-consumer case, clients and server communicate with each other by a shared buffer (a communication channel or a mailbox if we use a message passing interface) to store clients' requests. The server will treat them sequentially, avoiding any conflict in accessing resources between clients. The client-server scheme is used extensively in concurrent applications, and it is well suited for implementing services between processes, specially for processes in different computers communicating through a network.