<u>Exercice 1</u>

In this exercice, we rely on Hoare monitors as presented in the lectures (this means priority to the signal receiver and implicit locking of the monitor).
We want to control the lifecycle of a software component (like a box). This component can be stopped and started. If the component is in the stopped state, there must not be any activity running inside the component. The monitor which implements such a lifecycle for a component is composed of the following procedures:

- *enter()* : executed by each activity which wants to enter the component (must block if the component is stopped, until the component is started).
- *exit()* : executed by each activity which wants to exit the component.
- *stop()* : executed by an activity which wants to stop the component. This procedure returns only when there isn't any activity running in the component (must block if activities are running inside).
- *start()* : executed by an activity which wants to start the component. The *start()* procedure is always called by the activity which previously stopped the component.

We assume here that only one activity can invoke the *stop()* procedure (i.e. *stop()* cannot be invoked while the component is stopped).
You must program this monitor. Clarity is an important evaluation criteria.
Indication: the problem is very similar to a reader/writer scheme, you can implement it with two conditions: *start* and *stop*.

<u>Exercice 2</u>

In this question, we rely on Hoare monitors as presented in the lectures (this means priority to the signal receiver and implicit locking of the monitor).

You must program a monitor which implements a producer-consumer scheme (as explained in the lectures). This monitor called *ProdCons* provides two procedures:

- *void produce(Item it);*
- *Item consume();*

The number of places in the buffer is a constant *N* and the two above procedures rely on the two following procedures to effectively manage the buffer (you don't have to implement the code which manages the buffer).

- void effective_production(Item it);
- Item effective_consumption();

<u>You have to implement *ProdCons* with ONE AND ONLY ONE condition variable.</u>

<u>Exercice 3</u>

In this question, we use Semaphores as presented in the lectures.
We consider two computation processes P1() and P2() that we launch in parallel (on a multi-core architecture). A third process Display(), which shows the results of the computations, should start only when P1() and P2() completed their computations.
Describe the code (using Semaphores) that you must add to P1(), P2() and Display() in order to implement this synchronization scheme.

Exercice 4

In this exercise, we rely on POSIX monitors that we studied in the OS lecture.

We want to implement a service for sending messages over the network. This service is used by concurrent processes. Each sent message is composed of a set of packets. A message can be sent with the function *sendMessage()*. This function records the packets in a *buffer* and sends (with *sendBuffer()*) the packets when the buffer is full.
The implementation code of this service is given below :

```
Packet buffer[N];
int np = 0;

void sendMessage (int nb, Packet p[]) {
        int i = 0;
        while (i<nb) {
                buffer[np++] = p[i++];
                if (np==N) {
                        sendBuffer(buffer);
                        np = 0;
                }
        }
}
```

Question 1
Why can't we let concurrent processes execute this code as is ?
Propose a modification which can allow concurrent execution.

Question 2
We now want each process which sends a message (with *sendMessage()*) to be blocked until the message (i.e. all of its packets) are effectively sent (with *sendBuffer()*). Propose an implementation of this synchronisation with monitors.

Exercice 5

In this exercise, we rely on Hoare monitors that we studied in the OS lecture. You don't have to manage the monitor's mutual exclusion (mutex) and we assume priority to the signal receiver.

We want to control the execution of two programs A and B.

Program A	Program B

resA = Acompute1();
I_need_B();
Acompute2(resB);
	resB = Bcompute1();
	I_need_A();
	Bcompute2(resA);

You have to implement a monitor which provides the 2 functions I_need_A() and I_need_B().

Acompute2() requires the result from Bcompute1().
Bcompute2() requires the result from Acompute1().

<u>Exercice 6</u>

In this exercise, we rely on Hoare monitors that we studied in the OS lecture.

We want to implement a resource allocation service which allows the allocation of resources from a pool of N instances. Each process allocates its resources in one call and frees them before invoking another allocation (no nested calls). This allocation service provides the two following functions:

```
void alloc (int nb) {
  // synchronisation code – to be written
  // effective allocation  – don't have to be written
}
```

```
void free (int nb) {
     // synchronisation code – to be written
    // effective free  – don't have to be written
}
```

You only need the following global variables in the monitor:

```
        Condition alloc;
        int nbfree = N;
```

- In a first step, we implement a very simple solution. An alloc() which cannot be satisfied (according to the nbfree variable) blocks the calling process (it waits), and a free() wakes up (signals) a unique blocked process (this process blocks again if it cannot be satisfied). Implement this solution (the synchronisation parts of the alloc and free functions).
- The previous solution is not satisfactory as it wakes up a unique process on a free(). Modify the previous solution to wake up all the blocked processes on a free(). You don't need any other variable. Use a cascade signal.