
MARCIN RADLAK

CONCURRENT AND REAL TIME SOFTWARE DESIGN

ASSIGNMENT 1

PROGRAMMING IN SR

*Coventry University
January 2006*

Content

1. Introduction.....3

2. PART I – Communication and synchronization mechanisms.....3

2.1. Semaphores.....3

2.2. Monitors.....6

2.3. Message Passing9

3. PART II – Concurrent processes9

3.1. Process data flow diagram (DFD)..... 10

3.2. Description of the chosen inter process communication 11

4. PART III – Program code in SR concurrent language..... 12

4.1. Program listing 12

4.2. Example output..... 14

5. Summary..... 16

1. Introduction

In a computer system often many processes are executed simultaneously. This happens because all the environment around us works this way. And a computer simulates the environment and is a part of it. All the processes that are a part of concurrent system do what they are designed to do. But it always happens, that they have to communicate between themselves to share data that are crucial to finish the work. This is what the concurrent programming was developed for: simulate and synchronize processes that runs at the same time. This type of programming is very much different from sequential programming and because of that is a source of new problems. One of them, most important, is to control them to work properly while sharing common data.

This is the aim of the project: show and learn difficulties in developing concurrent programs while simulating surrounding environment. I have started from describing 3 most important process synchronization methods:

- semaphores
- monitors
- message passing

After gaining the knowledge about them I have chosen the method to solve problem which I called Air Traffic Control. All the reasons of choosing monitors I have described in Part II of the project.

Part III is an implementation of an algorithm that solves described in Part II concurrent problem. It also presents some examples of program output.

2. PART I – Communication and synchronization mechanisms

2.1. Semaphores

Semaphores are kind of data structure of preemptive multitasking. Their main task is to reduce the time for task switching and further enhance the usage of CPU time. They regulates the execution of concurrent processes. Semaphore stops process (sends sleep) or resumes it (wakes up). Because of that, it can have two main states. When the process gets to the semaphore, it is put down, execution is stopped and process waits for up signal. It is common to use P() as a wait signal, and V() as go which are defined as follows:

<pre> P(S) { If (s ≥ 1) { s -= 1; } Else if (s == 0) { block process and add to queue } } </pre>	<pre> V(S) { If (queue == empty) { s += 1; } Else if (queue != empty) { resume first process in queue and reschedule queue } } </pre>
--	---

Here we can see one problem arising from a described situation. Process is being stopped, because the semaphore does not allow it to go further until the condition is fulfilled. Process has to be in loop and check every time if the semaphore changed its state to up. This solution, very unpractical, is called "busy waiting". The CPU have much more work to do because of the execution of the loop. Considering above situation, semaphores are implemented using system scheduler, to avoid busy waiting. It is done the way, that when the process needs to be stopped, it is added to the list of waiting processes. This list (of pipe type – first come first served) is operated by mentioned system scheduler, which resumes process after "go" signal appears. Now it is clearly shown how much of processor time we can save using described data structure.

Semaphore is an integer variable (S) that can be accessed (apart from initialization) by two mentioned before operations: P(S) – wait, and V(S) – signal. These two operations are called atomic. It means, that modification of semaphore variable is indivisible, so when one process modifies semaphore value, no other process can simultaneously modify the same semaphore value. In addition to what have just been told, in the case of P(S), the testing of the integer value of S ($S < 0$) and its possible modification $S -= 1$ must also be executed without any interruption.

After explanation how semaphores work, comes another question: what are they for? We can use them for solving two main synchronization problems. One is a "critical section" and the other is when we want to execute processes in fixed order. Critical section is a piece of code, in which shared by all the processes data are modified. It is important then that only one process can access and modify these data to avoid further conflicts.

Semaphores can be used to execute processes in an order we would like them to be executed. Process that is being executed, when gets to a semaphore which is put down, is being send sleep, until "go" signal is shown. Until that time, other process can be executed, and when is finished can change the state of semaphore to allow waiting processes enter blocked section.

To better explain what has just been told, following example is shown. Lets consider three outputs: A, B, C. The rules that apply to these outputs are:

A or B must be output before any C's can be output.

B's and C's must alternate in the output string, that is, after the first B is output, another B cannot be output until a C is output. Similarly once a C is output, another C cannot be output until a B is output.

The total number of B's and C's which have been output at any given point in the output string cannot exceed the number of A's which have been output up to that point.

```
# Example solutions
```

```
#
```

```
# AACB      -- invalid, violates a)
```

```
# ABACAC    -- invalid, violates b)
```

```
# AABCABC   -- invalid, violates c)
```

```
# AABCAAABC -- valid
```

```
# AAAABCBC  -- valid
```

```
# AB        -- valid
```

```
-----
```

```
resource ABC()
```

```
sem B := 0, C := 1 # binary semaphores: only take on values 0 and 1
```

```
sem sum := 0      # counting semaphore: can have any positive value
```

```
process Pa
```

```
do true -> nap(int(random(5000)))
```

```
  writes("A")
```

```
  V(sum)
```

```
od
```

```
end Pa
```

```
process Pb
```

```
do true -> nap(int(random(10000)))
```

```
  P(C); P(sum)
```

```
  writes("B")
```

```
  V(B)
```

```
od
```

```
end Pb
```

```
process Pc
```

```
do true -> nap(int(random(10000)))
```

```
  P(B); P(sum)
```

```
  writes("C")
```

```

    V(C)
  od
end Pc
end ABC
-----

```

Example output:

```
ABAACAAAABCBCACAABCAABCBCACBACABAACBAACABCAAAAABCBAACBAACBCA
```

When we analyze above example closer, it is seen that semaphore sum stops processes Pb and Pc from execution to fulfil statement C of a task. Then we have two semaphores C and B, that are used to fulfil statement b of a task. Once B is executed, C is stopped and on the other hand, when C is executed, B waits. Finally, we have initialized semaphore C with value equals to 1. It means that process Pc is stopped from the beginning and waits until B and A is printed. This fulfils statement A of a task.

Semaphores may be very confusing and are easily to be used mistakenly. This is because one semaphore can be used to mutual exclusion synchronization as well as for condition synchronization. They are simply to low level structures, and in more complex tasks they are difficult to implement.

2.2. Monitors

A monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package. Processes may call the procedure in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor. The following example shows the structure of a monitor:

```

monitor example
integer i;
condition c;

procedure producer(x);
.
end;

procedure consumer(x);
.

```

```
end;  
end monitor
```

Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant. Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls. Typically when a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within monitor. If so, the calling process will be suspended until the other process has left monitor. If no other process is using the monitor, the calling process may enter.

It is up to a compiler to implement the mutual exclusion on monitor entries, but a common way is to use a binary semaphore. Because the compiler, not the programmer is arranging mutual exclusion, it is much less likely that something will go wrong. In any event, the person writing monitors does not have to be aware of how the compiler arranges for mutual exclusion. It is sufficient to know that by turning all the critical regions into monitor procedures, no two processes will ever execute their critical region at the same time.

Although monitors provide an easy way to achieve mutual exclusion, as we have seen above, that is not enough. We also need a way for processes to block when they cannot proceed. In the producer consumer problem, it is easy enough to put all the tests for buffer-full and buffer empty in monitor procedures, but now comes a problem how should the producer block when it finds the buffer full?

The solution for this are condition variables, along with two operations on them: WAIT and SIGNAL. When a monitor procedure discovers that it cannot continue it does a WAIT on some condition variable (i.e. full). This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

This other process, for example, the consumer, can wake up its sleeping partner by doing a SIGNAL on the condition variable that its partner is waiting on. To avoid having two active processes in the monitor at the same time, we need a rule telling what happens after SIGNAL. This rule is that a process doing a SIGNAL must exit the monitor immediately. It means that SIGNAL statement may appear only as the final statement in a monitor procedure. So if a SIGNAL is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler is revived.

Condition variables are not counters. They do not accumulate signals for later use the way semaphores do. Thus if a condition variable is signalled with no one waiting on it, the signal is lost. The WAIT must come before SIGNAL. This rule makes the implementation much simpler. In practice it is not a problem because it is easy to keep track of the state of each process with variables, if need be. A process that might otherwise do a SIGNAL can see that this operation is not necessary by looking at the variables.

The following example shows a Producer – Consumer problem written in description code:

<pre> monitor ProducerConsumer condition full, empty; integer count; process enter; if (count = N) -> wait(full); //enter item count ++; if (count = 1) -> signal(empty); end enter; process remove; if (count == 0) -> wait(empty); //remove item count --; if (count == N - 1) -> signal (full); end remove end monitor </pre>	<pre> process producer do true -> produce_item; ProducerConsumer.enter; od end producer process consumer do true -> ProducerConsumer.remove; Consume item; od end consumer </pre>
--	--

Listing 2.1. An outline of Producer Consumer problem using monitors. The buffer has N slots.

The automatic mutual exclusion on the monitor procedures guarantees that if the producer inside a monitor procedure discovers that the buffer is full, it will be able to complete the WAIT operation without having to worry about the possibility that the scheduler may switch to the consumer just before the WAIT completes. The consumer will not even be let into the monitor at all until the wait is finished and the producer has been marked as no longer runnable.

By making the mutual exclusion of critical regions automatic, monitors make parallel programming much less error sensitive than with semaphores. Still, they too have some disadvantages, like i.e. to use monitors, the language that has them built in is needed. The other problem with monitors is that they were designed for solving the mutual exclusion problem on one or more CPUs that all have access to a common memory. In a distributed system consisting of multiple CPUs, each with its own private memory, connected by a local area network, these primitives become inapplicable. Something else is needed.

2.3. Message Passing

This method of interprocess communication uses two primitives: SEND and RECEIVE, which like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

send (destination, & message);

and

receive (source, &message);

The former call sends a message to a given destination and the latter one receives a message from a given source (or from ANY, if the receiver does not care). If no message is available, the receiver could block until one arrives. Alternatively, it could return immediately with an error code.

Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating process are on different machines connected by a network.

That is the message is the sender and receiver can agree that the sender and receiver will send back a special acknowledgement message. If the sender has not received their acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver can distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored.

Message systems also have to deal with the question of how processes are named, so that the process specified in a SEND or RECEIVE call is unambiguous. Authentication is also an issue in message systems: how can the client tell that he is communicating with the real server, and not with an imposter?

At the other end of the spectrum, there are also design issues that are important when the sender and receiver are on the same machine. One of these is performance. Copying message from one process to another is always slower than doing a semaphore operation or entering a monitor. Much work has gone into making message passing efficient.

3. PART II – Concurrent processes

In part II of my project I will show an example of concurrent environment. It shows an instance of Air Traffic Control System. The rules of the system is shown on the following diagram.

3.1. Process data flow diagram (DFD)

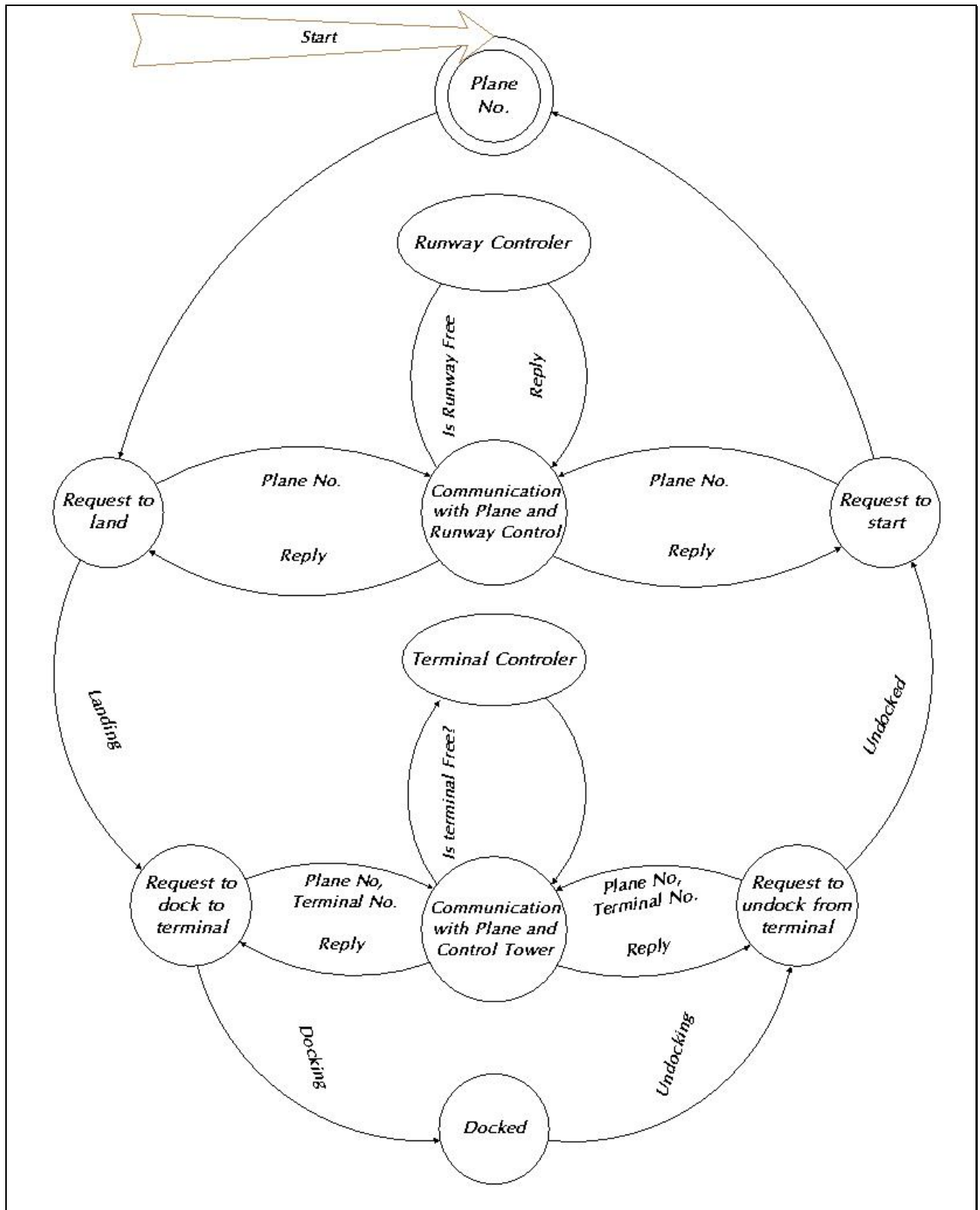


Fig. 2.1 Process Data Flow Diagram (DFD)

Next Fig. 2.2 shows a part of above system. This is because I have programmed only a piece of the whole Air Traffic Control.

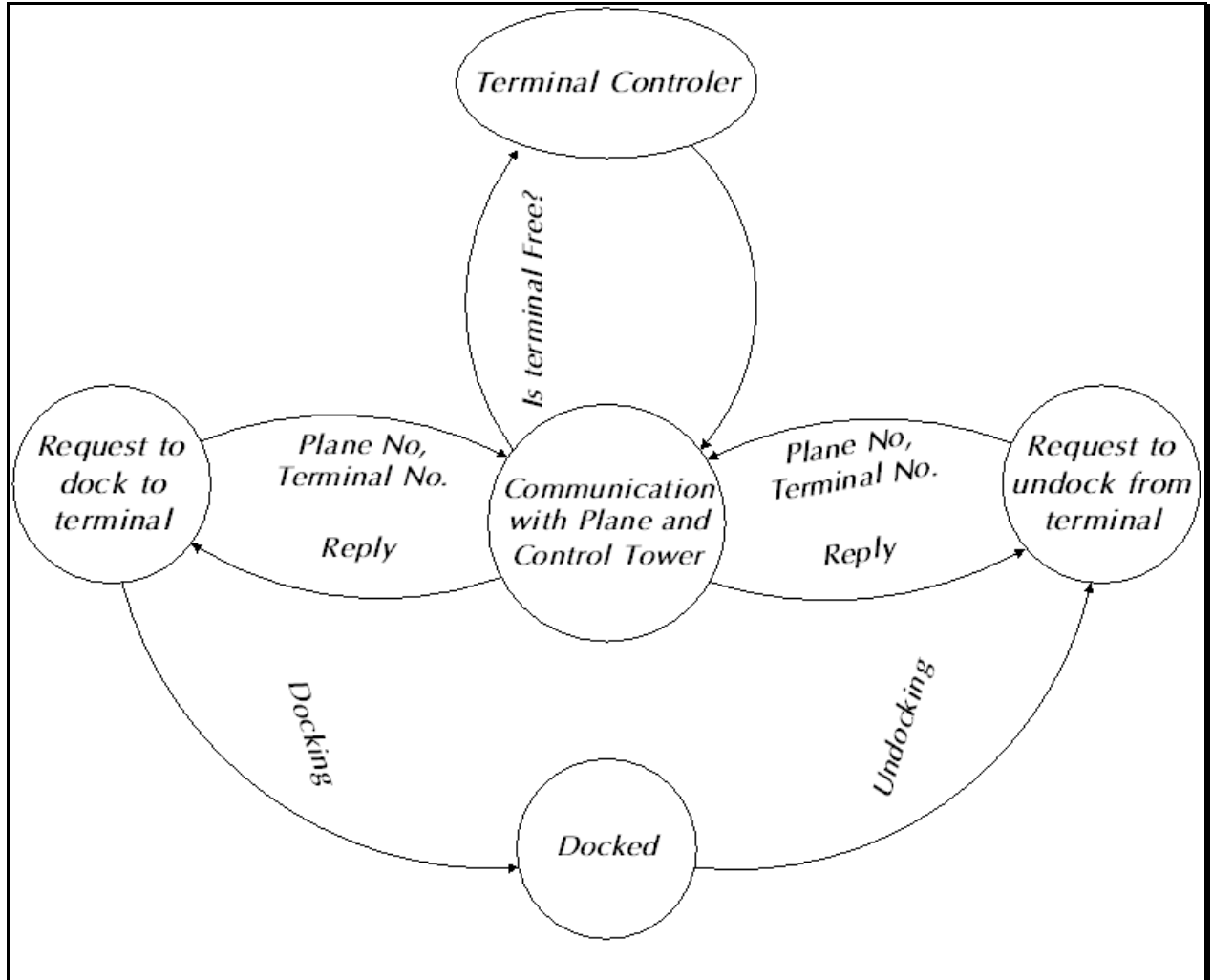


Fig 2.2 Part of overall system that is implemented In SR

3.2. Description of the chosen inter process communication

The algorithm which I have implemented shows simplified processes of normal Air Traffic Control System. It is based on allowing to use a runway by airplane and also directing it to the correct terminal after landing.

My project is based on processes that runs while the plane requests to dock to terminal. It was later also extended to include undocking aircrafts.

Problem in concurrency was that there are many planes, that lands and starts from the airport. Each of them has to exchange important data with Air Traffic Controller. There is only one plane at once allowed to communicate. After the communication is finished, the plane is directed to requested terminal. In case that the terminal is not available yet, the plane has to wait until other plane docked to it will depart. While plane is waiting for a terminal to be free, Air traffic Controller can communicate with other airplanes. It is also possible that other planes dock to their terminals,

and do not have to wait until the waiting plane is docked. Finally, when the terminal requested is left by other plane, waiting one is notified by Terminal Controller, that it can dock to it.

Everything described above is shown on DFD in fig 2.1 and 2.2

I think, this problem is very similar to sleeping barber problem. This is the reason why I have implemented my program I was based on the sleeping barber algorithm. It had to be modified to meet exactly my needs but still it is very similar.

As a synchronization method I have chosen monitors. The reason for this is because my knowledge about them, how do they work and how to use them was the smallest. On the other side they allow to implement very difficult concurrent problems in rather well organised and simple way. The result of my work is shown in Part III of this report where the program listing and example outputs are printed.

4. PART III - Program Code in SR concurrent language

Program was written and tested on Zappa computer under Unix environment. The editor program used was Pico.

4.1. Program listing

Program listing as well as the example output file have been sent via email from my zappa account (radlakm). It is also shown in two following listings

Listing 4.1 Main file controlmain3.sr

```

1  #Source code by Marcin Radlak, radlakm@coventry.ac.uk
2  resource controlmain()
3
4
5  var T, N: int          # T - no of terminals (max 100), N - no of planes
6  var napping : int
7
8
9  getarg(1,N)
10 getarg(2,T)
11
12 import MonControl
13
14 process controler
15 var ttt : int          # number of plane which controler communicates with
16 do true ->
17   ttt:=MonControl.NextPlane()
18   write("  communication with plane", ttt)
19   nap(100)             # time of communication, always fixed
20   write("  end communication with plane", ttt)
21   MonControl.EndCommunication()
22 od
23 end
24
25
26 process plane(i:= 1 to N)
27 do true ->
28   var tno : int
29   napping:=int(random()*1000)
30   nap(napping*10)      # wait for need to land
31   tno := int(random()*T)+1  # assign terminal number to a plane
32   write("Plane", i ,"requests to dock to terminal", tno)
33   MonControl.Request(i, tno, dock)
34   write("Plane", i, "docks to terminal", tno)
35   write("Plane", i, "docked to terminal", tno)

```

```

36     nap(napping*10)
37     write("Plane", i, "requests to undock from terminal", tno)
38     MonControl.Request(i, tno, undock)
39     write("Plane", i, "undocked from terminal", tno)
40   od
41   end
42
43
44 end controlmain

```

Listing 4.2 Monitor file control3.m

```

1  _monitor(MonControl)
2
3     type operation = enum(dock, undock)
4
5     op Request(p : int; t : int; oper : operation)
6     op NextPlane() returns i : int
7     op EndCommunication()
8
9     var T : int := 100    #max number of available terminals
10
11
12 _body(MonControl)
13     var Tfree[1:T]: bool := ([T] true) # idicates if terminal x free
14     _condvar1(Tfree_q, 1:T)           # signalled when terminal x f
15     var avail: bool := false          # indicates if controler avai
16     _condvar(avail_q)
17     var occup: bool := false          # indicates if plane communicating with controler
18     _condvar(occup_q)
19     var open: bool := false           # indicates if open way to terminal for plane
20     _condvar(open_q)
21     var left: bool := false           # indicates if plane left controler
22     _condvar(left_q)
23
24     var pid : int                     # plane number
25
26     _proc( Request(p, t, oper) )       # called by plane - request terminal
27     do not avail -> _wait(avail_q) od # wait until controler available
28     avail := false
29     occup := true                      # plane communicating with controler
30     pid := p
31     _signal(occup_q)                  # wake up controler
32     do not open -> _wait(open_q) od # wait until controler finishe job
33     open := false
34     left := true
35     _signal(left_q)                   # signal that plane finished communicating with
                                         controler
36     if oper = dock ->                  # if docking
37     if Tfree[t] = false ->             # check if terminal free
38     write("---- Terminal", t, "not available. Plane", p, "have wait")
39     fi                                  # if termianl not available, plane waits until
                                         free. Other planes can now communicating with
                                         controler
40     do not Tfree[t] -> _wait(Tfree_q[t]) od
41     Tfree[t] := false
42     [] else ->                          # else if undocking
43     Tfree[t] := true
44     _signal(Tfree_q[t])                # notify that terminal t free
45     fi
46     _proc_end
47
48     _proc( NextPlane() returns i )     # called by controler
49     avail := true
50     _signal(avail_q)                   # notify that controler available
51     do not occup -> _wait(occup_q) od # wait for plane
52     occup := false
53     i := pid                            # get plane number
54     _proc_end
55
56     _proc( EndCommunication() )        # called by controler

```

```
57         open :=true
58         _signal(open_q)
59         do not left -> _wait(left_q) od # wait until plane stops communicating with
                                         controller and take its way to terminal
60         left := false
61     _proc_end
62
63 _monitor_end
```

All the program is described with comments so no further explanation is necessary

4.2. Example output

To run a program, source files have to be compiled: control3.m and controlmain3.sr. Control3.m is a file containing monitors definitions and controlmain3.sr is a user process which makes use of these monitors. After that, executable file takes two arguments: first is number of terminals, the second, number of planes. After execution of the program, it is clearly seen that concurrent processes are served in order, that was supposed to be. The following listing shows example output:

```
Script started on Wed Jan 04 20:13:04 2006
zappa% control.out 10 10
Plane 2 requests to dock to terminal 3
  communication with plane 2
  end communication with plane 2
Plane 2 docks to terminal 3
Plane 2 docked to terminal 3
Plane 7 requests to dock to terminal 3
  communication with plane 7
  end communication with plane 7
---- Terminal 3 not available. Plane 7 have to wait
Plane 8 requests to dock to terminal 2
  communication with plane 8
  end communication with plane 8
Plane 8 docks to terminal 2
Plane 8 docked to terminal 2
Plane 6 requests to dock to terminal 6
  communication with plane 6
  end communication with plane 6
Plane 6 docks to terminal 6
Plane 6 docked to terminal 6
Plane 3 requests to dock to terminal 8
  communication with plane 3
  end communication with plane 3
Plane 3 docks to terminal 8
Plane 3 docked to terminal 8
Plane 4 requests to dock to terminal 10
  communication with plane 4
  end communication with plane 4
Plane 4 docks to terminal 10
Plane 4 docked to terminal 10
Plane 5 requests to dock to terminal 6
  communication with plane 5
Plane 1 requests to dock to terminal 5
  end communication with plane 5
---- Terminal 6 not available. Plane 5 have to wait
  communication with plane 1
  end communication with plane 1
Plane 1 docks to terminal 5
Plane 1 docked to terminal 5
Plane 9 requests to dock to terminal 3
  communication with plane 9
  end communication with plane 9
---- Terminal 3 not available. Plane 9 have to wait
Plane 10 requests to dock to terminal 6
  communication with plane 10
  end communication with plane 10
```

```
---- Terminal 6 not available. Plane 10 have to wait
Plane 2 requests to undock from terminal 3
  communication with plane 2
  end communication with plane 2
Plane 2 undocked from terminal 3
Plane 7 docks to terminal 3
Plane 7 docked to terminal 3
Plane 8 requests to undock from terminal 2
  communication with plane 8
  end communication with plane 8
Plane 8 undocked from terminal 2
Plane 6 requests to undock from terminal 6
  communication with plane 6
  end communication with plane 6
Plane 6 undocked from terminal 6
Plane 5 docks to terminal 6
Plane 5 docked to terminal 6
Plane 7 requests to undock from terminal 3
Plane 2 requests to dock to terminal 1
  communication with plane 7
Plane 3 requests to undock from terminal 8
  end communication with plane 7
Plane 7 undocked from terminal 3
Plane 9 docks to terminal 3
Plane 9 docked to terminal 3
  communication with plane 2
  end communication with plane 2
Plane 2 docks to terminal 1
Plane 2 docked to terminal 1
  communication with plane 3
  end communication with plane 3
Plane 3 undocked from terminal 8
Plane 4 requests to undock from terminal 10
  communication with plane 4
  end communication with plane 4
Plane 4 undocked from terminal 10
Plane 1 requests to undock from terminal 5
  communication with plane 1
  end communication with plane 1
Plane 1 undocked from terminal 5
Plane 4 requests to dock to terminal 1
  communication with plane 4
  end communication with plane 4
--- Terminal 1 not available. Plane 4 have to wait
lane 8 requests to dock to terminal 6
  communication with plane 8
  end communication with plane 8
--- Terminal 6 not available. Plane 8 have to wait
lane 5 requests to undock from terminal 6
lane 6 requests to dock to terminal 10
  communication with plane 5
  end communication with plane 5
Plane 5 undocked from terminal 6
Plane 10 docks to terminal 6
Plane 10 docked to terminal 6
  communication with plane 6
Plane 9 requests to undock from terminal 3
Plane 7 requests to dock to terminal 3
  end communication with plane 6
Plane 6 docks to terminal 10
Plane 6 docked to terminal 10
  communication with plane 9
Plane 2 requests to undock from terminal 1
  end communication with plane 9
Plane 9 undocked from terminal 3
  communication with plane 7
  end communication with plane 7
Plane 7 docks to terminal 3
Plane 7 docked to terminal 3
  communication with plane 2
  end communication with plane 2
```

```
Plane 2 undocked from terminal 1
Plane 4 docks to terminal 1
Plane 4 docked to terminal 1
Plane 3 requests to dock to terminal 5
  communication with plane 3
  end communication with plane 3
Plane 3 docks to terminal 5
Plane 3 docked to terminal 5
Plane 4 requests to undock from terminal 1
Plane 2 requests to dock to terminal 2
  communication with plane 4
  end communication with plane 4
Plane 4 undocked from terminal 1
  communication with plane 2
  end communication with plane 2
Plane 2 docks to terminal 2
Plane 2 docked to terminal 2
^Czappa% exit
zappa%
script done on Wed Jan 04 20:13:45 2006
```

5. Summary

All the program showed in Part II works and satisfies all the assumptions made before I begun writing a code. I have also learned a lot about concurrent programming, i.e processes, synchronization methods and algorithms as well as about large amount of different problems linked to this type of programming.

The most I have achieved and of which I am very proud of is the ability to use monitors, message passing and semaphores what was also the aim of the project.

I have gained skills and experience in programming SR concurrent language and got familiar with Unix environment which I have hardly ever worked with.