

(1) Fair Readers-Writers problem with semaphores

```
1  var  q, mutex, wrt:  semaphore;
2      readcount:  integer;
3
4  begin {initialize}
5      readcount  := 0;
6      q          := 1;
7      mutex      := 1;
8      wrt        := 1;
9  end;
10
11 procedure entry Reader()
12     begin
13         repeat
14             wait(q);
15
16             wait(mutex);
17             readcount := readcount + 1;
18             if readcount = 1 then wait(wrt);
19             signal(mutex);
20
21             signal(q);
22
23             [READ FROM POOL]
24
25             wait(mutex);
26             readcount := readcount - 1;
27             if readcount = 0 then signal(wrt);
28             signal(mutex);
29
30             [REMAINDER SECTION - USE DATA]
31         until false;
32     end;
33
34 procedure entry writer()
35     begin
36         repeat
37             [REMAINDER SECTION - MAKE DATA]
38
39             wait(q);
40             wait(wrt);
41             signal(q);
42
43             [WRITE TO POOL]
44
45             signal(wrt);
46         until false;
47     end;
```

The main feature in this code is the addition of the "q" semaphore, which each reader and writer waits on. This is what guarantees the FIFO ordering for read/write requests, while maintaining the exclusive write access and shared read access.

Code explanation:

1-2 Variables

4-9 Initialization – all semaphores are non-binary and are initialized in the standard way, i.e. allow the first wait() statement to pass

- 14 The reader waits on the "q" semaphore, waiting to be allowed to enter the queue
- 16-19 Increment the reader count. If this is the first reader in the queue, we must wait on "wrt" in case a writer is writing. (Any readers immediately behind this one will be waiting on "q".)
- 21 Signal "q" to allow the next reader/writer to enter the queue.
- 23 Read the data from the pool/file/database/whatever.
- 25-28 Decrement the reader count. If this is the last reader, then release (wrt) for the next reader/writer.
- 30 The reader consumes the data in some application-specific way

- 37 The writer creates the data to be written in some application-specific way
- 39 Wait on "q" to enter the queue to read/write
- 40 Wait on "wrt" to be able to have exclusive access to the pool/file/database/whatever.
- 41 Once we have exclusive access to "wrt", then the writer can signal "q" and allow other readers/writers to enter the queue.
- 43 Write the data to the pool/file/database/whatever.
- 45 Release "wrt" for the next reader/writer.

(2) Fair Readers-Writers problem with conditional critical region statements

```
1  type reader_writer = class
2      var v:          shared record
3          nreaders:   integer;
4          busy:       boolean;
5          q:          integer;
6          next:       integer;
7      end;
8
9      begin {initialize}
10         v.busy           := false;
11         v.nreaders       := 0;
12         v.q              := 0;
13         v.next           := 0;
14     end;
15
16     procedure entry open_read()
17         var my_num: integer;
18         begin
19             region v do begin
20                 my_num = q;
21                 q := q + 1;
22
23                 await(my_num = next);
24
25                 await(not(busy));
26                 nreaders := nreaders + 1;
27
28                 next := next + 1;
29             end
30         end;
31
32     procedure entry close_read()
33         begin
34             region v do nreaders := nreaders - 1;
35         end;
36
37     procedure entry open_write()
38         var my_num: integer;
39         begin
40             region v do begin
41                 my_num = q;
42                 q := q + 1;
43
44                 await(my_num = next);
45
46                 await(not(busy) and nreaders = 0);
47                 busy = true;
48
49                 next := next + 1;
50             end
51         end;
52
53     procedure entry close_write()
54         begin
55             region v do busy = false;
56         end;
57
58 end class;
59
60
61 var rw: class reader_writer;
62 procedure entry writer()
```

```

63         begin
64             repeat
65                 [REMAINDER SECTION - MAKE DATA]
66                 rw.open_write();
67                 [WRITE TO POOL]
68                 rw.close_write();
69             until false
70         end;
71
72 procedure entry reader()
73     begin
74         repeat
75             rw.open_read();
76             [READ FROM POOL]
77             rw.close_read();
78             [REMAINDER SECTION - USE DATA]
79         until false
80     end;
81
82
83
84
85
86

```

I assume, in this code, that the “q” and “next” integer variables are allowed to become arbitrarily large. In the real world, one could add code to mod them if they become too large, and therefore wrap them around to zero.

Code explanation:

2-7 The variables of the critical region shared record.

9-14 Initialize the variables of the critical region shared record.

20-23 Get the next number in the queue (sort of the bakery number) and wait your turn in line.

25-26 Once it’s your turn, the reader waits until any writer is done, then increment the reader count.

28 Give the next reader/writer his turn. If the next guy is a reader, it will proceed to read, since busy = false. If the next guy is a writer, it will be stopped on await(not(busy) and nreaders=0).

34 To close the read, decrement the reader count.

41-44 Get the next number in the queue (sort of the bakery number) and wait your turn in line.

46-47 Once it’s your turn, the writer waits until any readers are done, then sets busy=true.

49 Give the next reader/writer his turn. If the next guy is a reader, it will proceed to read, since busy = false. If the next guy is a writer, he will be stopped on await(not(busy) and nreaders=0).

55 To close the write, set busy=false.

61-86 This is an implementation using the reader_writer class to illustrate how the methods are called.

(3) Ski lift problem with a monitor

```
type ski_lift = monitor
var   lt:      array [0..2] of condition;
      rt:      array [0..2] of condition;
      ls:      condition;
      rs:      condition;
      ltcount: integer;      // The number of people in line, not the number of triples
      rtcount: integer;      // The number of people in line, not the number of triples
      lscount: integer;
      rscount: integer;
      loadinglt: boolean;    // whether or not the left triple line has priority
      loadingls: boolean;    // whether or not the left single line has priority

begin {initialize}
      ltcount      := 0;
      rtcount      := 0;
      lscount      := 0;
      rscount      := 0;
      loadinglt    := true;
      loadingls    := true;
end;

procedure entry get_in_line()
begin
      if ( (lscount < 2 * ceiling(ltcount / 3))
          && (lscount < 2 * ceiling(rtcount / 3))
          && (lscount < rscount) )
      then begin
          //Load in the left single line
          lscount := lscount + 1;
          ls.wait();
      end
      else if ( (rscount < 2 * ceiling(ltcount / 3))
               && (rscount < 2 * ceiling(rtcount / 3))
               && (rscount <= lscount) )
      then begin
          //Load in the right single line
          rscount := rscount + 1;
          rs.wait();
      end
      else if (ltcount < rtcount)
      then begin
          //Load in the left triple line
          ltcount := ltcount + 1;
          ltcount[(ltcount-1) mod 3].wait();
      end
      else begin
          //Load in the right triple line
          rtcount := rtcount + 1;
          rtcount[(rtcount-1) mod 3].wait();
      end
      end;
end;

procedure entry load_lift() : boolean
var   ltloaded:  boolean;
      rtloaded:  boolean;
      lsloaded:  boolean;
      rsloaded:  boolean;

begin
      if ltcount >= 3 then
          ltloaded := true;
      else
          ltloaded := false;

      if rtcount >= 3 then
          rtloaded := true;
      else
          rtloaded := false;

      if lscount >= 1 then
          lsloaded := true;
```

```

else
    lsloaded := false;

if rscount >= 1 then
    rsloaded := true;
else
    rsloaded := false;

if loadinglt then begin
    // Left triple has priority
    if ltloaded then begin
        //Left triple is loaded
        if loadingls then begin
            //left single has priority
            if lsloaded then begin
                //Left single is loaded
                // Load left triple and left single
                ltcount := ltcount - 3;
                lscount := lscount - 1;
                lt[0].signal();
                lt[1].signal();
                lt[2].signal();
                ls.signal();
                loadinglt := false;
                loadingls := false;

                return true;
            end
        else begin
            //Left single is not loaded
            if rsloaded then begin
                //Right single is loaded
                // Load left triple and right single
                ltcount := ltcount - 3;
                rscount := rscount - 1;
                lt[0].signal();
                lt[1].signal();
                lt[2].signal();
                rs.signal();
                loadinglt := false;
                loadingls := true;

                return true;
            end
        else return false; //No singles loaded
        end
    end
end
else begin
    //right single has priority
    if rsloaded then begin
        //Right single is loaded
        // Load left triple and right single
        ltcount := ltcount - 3;
        rscount := rscount - 1;
        lt[0].signal();
        lt[1].signal();
        lt[2].signal();
        rs.signal();
        loadinglt := false;
        loadingls := true;

        return true;
    end
    else begin
        //Right single is not loaded
        if lsloaded then begin
            //Left single is loaded
            // Load left triple and left single
            ltcount := ltcount - 3;
            lscount := lscount - 1;
            lt[0].signal();
            lt[1].signal();
            lt[2].signal();
            ls.signal();
            loadinglt := false;
        end
    end
end

```

```

        loadingls := false;
        return true;
    end
    else return false; //No singles loaded
end
end
end
else begin
    //Left triple is not loaded
    if rtloaded then begin
        //Right triple is loaded
        if loadingls then begin
            //left single has priority
            if lsloaded then begin
                //Left single is loaded
                // Load right triple and left single
                rtcount := rtcount - 3;
                lscount := lscount - 1;
                rt[0].signal();
                rt[1].signal();
                rt[2].signal();
                ls.signal();
                loadinglt := true;
                loadingls := false;

                return true;
            end
            else begin
                //Left single is not loaded
                if rsloaded then begin
                    // Right single is loaded
                    // Load right triple and right single
                    rtcount := rtcount - 3;
                    rscount := rscount - 1;
                    rt[0].signal();
                    rt[1].signal();
                    rt[2].signal();
                    rs.signal();
                    loadinglt := true;
                    loadingls := true;

                    return true;
                end
                else return false; // No singles loaded
            end
        end
    end
    else begin
        //right single has priority
        if rsloaded then begin
            //Right single is loaded
            // Load right triple and right single
            rtcount := rtcount - 3;
            rscount := rscount - 1;
            rt[0].signal();
            rt[1].signal();
            rt[2].signal();
            rs.signal();
            loadinglt := true;
            loadingls := true;

            return true;
        end
        else begin
            //Right single is not loaded
            if lsloaded then begin
                //Left single is loaded
                // Load right triple and left single
                rtcount := rtcount - 3;
                lscount := lscount - 1;
                rt[0].signal();
                rt[1].signal();
                rt[2].signal();
                ls.signal();
                loadinglt := true;
            end
        end
    end
end

```

```

loadingls := false;
return true;
end
else return false; // No singles loaded
end
end
end
end
else return false; // No triples loaded
end
end
else begin
// Right triple has priority
if rtloaded then begin
//Right triple is loaded
if loadingls then begin
//left single has priority
if lsloaded then begin
//Left single is loaded
// Load right triple and left single
rtcount := rtcount - 3;
lscount := lscount - 1;
rt[0].signal();
rt[1].signal();
rt[2].signal();
ls.signal();
loadinglt := true;
loadingls := false;

return true;
end
else begin
//Left single is not loaded
if rsloaded then begin
//Right single is loaded
// Load right triple and right single
rtcount := rtcount - 3;
rscount := rscount - 1;
rt[0].signal();
rt[1].signal();
rt[2].signal();
rs.signal();
loadinglt := true;
loadingls := true;

return true;
end
else return false; // No singles loaded
end
end
end
else begin
//right single has priority
if rsloaded then begin
//Right single is loaded
// Load right triple and right single
rtcount := rtcount - 3;
rscount := rscount - 1;
rt[0].signal();
rt[1].signal();
rt[2].signal();
rs.signal();
loadinglt := true;
loadingls := true;

return true;
end
else begin
//Right single is not loaded
if lsloaded then begin
//Left single is loaded
// Load right triple and left single
rtcount := rtcount - 3;
lscount := lscount - 1;
rt[0].signal();
rt[1].signal();

```

```

        rt[2].signal();
        ls.signal();
        loadinglt := true;
        loadingls := false;

        return true;
    end
else return false; // No singles loaded
end
end
end
else begin
    //Right triple is not loaded
    if ltloaded then begin
        //Left triple is loaded
        if loadingls then begin
            //left single has priority
            if lsloaded then begin
                //Left single is loaded
                // Load left triple and left single
                ltcount := ltcount - 3;
                lscount := lscount - 1;
                lt[0].signal();
                lt[1].signal();
                lt[2].signal();
                ls.signal();
                loadinglt := false;
                loadingls := false;

                return true;
            end
        else begin
            //Left single is not loaded
            if rsloaded then begin
                //Right single is loaded
                // Load left triple and right single
                ltcount := ltcount - 3;
                rscount := rscount - 1;
                lt[0].signal();
                lt[1].signal();
                lt[2].signal();
                rs.signal();
                loadinglt := false;
                loadingls := true;

                return true;
            end
        else return false; // No singles loaded
        end
    end
end
end
else begin
    //right single has priority
    if rsloaded then begin
        //Right single is loaded
        // Load left triple and right single
        ltcount := ltcount - 3;
        rscount := rscount - 1;
        lt[0].signal();
        lt[1].signal();
        lt[2].signal();
        rs.signal();
        loadinglt := false;
        loadingls := true;

        return true;
    end
end
else begin
    //Right single is not loaded
    if lsloaded then begin
        //Left single is loaded
        // Load left triple and left single
        ltcount := ltcount - 3;
        lscount := lscount - 1;
        lt[0].signal();
        lt[1].signal();
    end
end

```

```

        lt[2].signal();
        ls.signal();
        loadinglt := false;
        loadingls := false;

        return true;
    end
    else return false; // No singles loaded
end
end
end
end
end
end;

end monitor;

type skier = class
    var sl: monitor ski_lift;

    procedure entry skier()
    begin
        [PRE SECTION]

        sl.get_in_line();

        [EXECUTED WITH THREE OTHER PROCESSES]
    end;
end;

type ski_lift_operator = class
    var sl: monitor ski_lift;

    procedure entry operator()
    begin
        repeat
            repeat no_op until sl.load_lift() = true;
        until false;
    end;
end;
end;

```

Some assumptions:

- This monitor implementation uses option 2: If P_i executes `wait()`, then P_j executes `signal()`, then P_i waits until P_j leaves the monitor before waking up.
- This monitor implementation assumes that the processes will be awoken (with `signal()`) in the same order in which they called `wait()`.
- I assume the monitor function `load_lift()` is allowed to return a boolean value, which tells the lift operator whether the load was successful or not. Although my implementation of the `ski_lift_operator` class does not use this information for anything, it could have value in some instances (e.g. if you wanted to perform only one `load_lift`, then do something, then perform another `load_lift`, etc.).

Additionally, although the `load_lift` function is very long, it is only long in order to cover all the cases of line priority vs. whether a line has anyone in it or not, etc. Hopefully the comments explain this.