

(1) Fair Readers-Writers problem with a monitor

```
1  type reader_writer = monitor
2      var    q:      condition;
3            w:      condition;
4            writerwait: boolean;
5            qsize:  integer;
6            readcount: integer;
7
8      begin {initialize}
9            writerwait := false;
10           qsize      := 0;
11           readcount  := 0;
12      end;
13
14     procedure entry open_read()
15     begin
16         if qsize > 0 then begin
17             qsize := qsize + 1;
18             q.wait();
19             qsize := qsize - 1;
20         end
21
22         readcount := readcount + 1;
23         if qsize > 0 then q.signal();
24     end;
25
26     procedure entry close_read()
27     begin
28         readcount := readcount - 1;
29         if (readcount = 0) and writerwait then w.signal();
30     end;
31
32     procedure entry open_write()
33     begin
34         qsize := qsize + 1;
35
36         if qsize > 1 then q.wait();
37
38         if readcount > 0 then begin
39             writerwait := true;
40             w.wait();
41             writerwait := false;
42         end
43     end;
44
45     procedure entry close_write()
46     begin
47         qsize := qsize - 1;
48         if qsize > 0 then q.signal();
49     end;
50
51 end monitor;
52
53
54 var rw: monitor reader_writer;
55 procedure entry reader()
56     begin
57         repeat
58             rw.open_read();
59
60             [READ DATA]
61
62             rw.close_read();
63
```

```

64             [USE DATA]
65         until false
66     end;
67
68 procedure entry writer()
69     begin
70         repeat
71             [MAKE DATA]
72
73             rw.open_write();
74
75             [WRITE DATA]
76
77             rw.close_write();
78         until false;
79     end;

```

This main features of this code are a condition variable "q" which all readers and writers wait on if there are writers ahead of them in line, and a condition variable "w" which writers wait on if readers are reading.

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()`.
- The integer variables (and thus the number of waiting processes) are allowed to become arbitrarily large.

Code explanation:

2-6 Variables

8-12 Variable initialization

16-20 If there are other readers/writers in line ahead of this reader, then increase the queue size and wait in line.

23 Increase the number of readers currently active

24 If there is a reader/writer in line behind, then release it. If it's a reader, it will go ahead and read; if it's a writer, it will wait on "w".

28-29 Decrement the number of readers. If this was the last reader and there's a writer waiting next in line, then release it.

34-36 Increase the queue size, and wait in line if there are readers/writers ahead of this one. Note: we don't want to decrease the queue size after we wait on q, because we want to ensure that if this is the only writer in the queue, if a reader enters next, it needs to wait on "q", which it will do if qsize is greater than zero.

38-42 When this writer is released in the queue, if there are readers reading, then wait on "w" and indicate that a writer is waiting.

47-48 Decrement the queue size. If there is something in the queue, release it.

54-79 This is an implementation using the reader_writer class to illustrate how the methods are called.

(2) Ski lift problem with semaphores

```
type ski_lift = class
var
  lt:      array [0..2] of semaphore;
  rt:      array [0..2] of semaphore;
  ls:      semaphore;
  rs:      semaphore;
  mutex:   semaphore;    // Mutual exclusion for the following variables
  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
  loadingrt: boolean;    // whether or not the right triple line has priority
  loadingls: boolean;
  loadingrs: boolean;

begin {initialize}
  lt[0] := 0;
  lt[1] := 0;
  lt[2] := 0;
  rt[0] := 0;
  rt[1] := 0;
  rt[2] := 0;
  ls := 0;
  rs := 0;
  mutex := 1;
  ltcount := 0;
  rtcount := 0;
  lscount := 0;
  rscount := 0;
  loadinglt := true;
  loadingrs := true;
end;

procedure entry get_in_line()
var i: integer;
begin
  wait(mutex);

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

```

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

  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;
          signal(lt[0]);
          signal(lt[1]);
          signal(lt[2]);
          signal(ls);
          loadinglt := false;
          loadingls := false;

          signal(mutex);
          return true;
        end
      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;
        signal(lt[0]);
        signal(lt[1]);
        signal(lt[2]);
        signal(rs);
        loadinglt := false;
        loadingls := true;

        signal(mutex);
        return true;
      end
    else begin //No singles loaded
      release(mutex);
      return false;
    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;
    signal(lt[0]);
    signal(lt[1]);
    signal(lt[2]);
    signal(rs);
    loadinglt := false;
    loadingls := true;

    signal(mutex);
    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;
        signal(lt[0]);
        signal(lt[1]);
        signal(lt[2]);
        signal(ls);
        loadinglt := false;
        loadingls := false;

        signal(mutex);
        return true;
    end
    else begin //No singles loaded
        release(mutex);
        return false;
    end
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;
                signal(rt[0]);
                signal(rt[1]);
                signal(rt[2]);
                signal(ls);
                loadinglt := true;
                loadingls := false;

                signal(mutex);
                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;
                    signal(rt[0]);
                    signal(rt[1]);
                    signal(rt[2]);
                    signal(rs);
                    loadinglt := true;
                    loadingls := true;

                    signal(mutex);

```

```

        return true;
    end
    else begin //No singles loaded
        release(mutex);
        return false;
    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;
        signal(rt[0]);
        signal(rt[1]);
        signal(rt[2]);
        signal(rs);
        loadinglt := true;
        loadingls := true;

        signal(mutex);
        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;
            signal(rt[0]);
            signal(rt[1]);
            signal(rt[2]);
            signal(ls);
            loadinglt := true;
            loadingls := false;

            signal(mutex);
            return true;
        end
        else begin //No singles loaded
            release(mutex);
            return false;
        end
    end
end
end
end
else begin //No triples loaded
    release(mutex);
    return false;
end
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;
                signal(rt[0]);
                signal(rt[1]);
                signal(rt[2]);
                signal(ls);
                loadinglt := true;
                loadingls := false;

                signal(mutex);
                return true;
            end
        end
    end
end
end
end

```

```

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;
        signal(rt[0]);
        signal(rt[1]);
        signal(rt[2]);
        signal(rs);
        loadinglt := true;
        loadingls := true;

        signal(mutex);
        return true;
    end
    else begin //No singles loaded
        release(mutex);
        return false;
    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;
        signal(rt[0]);
        signal(rt[1]);
        signal(rt[2]);
        signal(rs);
        loadinglt := true;
        loadingls := true;

        signal(mutex);
        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;
            signal(rt[0]);
            signal(rt[1]);
            signal(rt[2]);
            signal(ls);
            loadinglt := true;
            loadingls := false;

            signal(mutex);
            return true;
        end
        else begin //No singles loaded
            release(mutex);
            return false;
        end
    end
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;
            end
        end
    end
end

```

```

        lscount := lscount - 1;
        signal(lt[0]);
        signal(lt[1]);
        signal(lt[2]);
        signal(ls);
        loadinglt := false;
        loadingls := false;

        signal(mutex);
        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;
        signal(lt[0]);
        signal(lt[1]);
        signal(lt[2]);
        signal(rs);
        loadinglt := false;
        loadingls := true;

        signal(mutex);
        return true;
    end
    else begin //No singles loaded
        release(mutex);
        return false;
    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;
        signal(lt[0]);
        signal(lt[1]);
        signal(lt[2]);
        signal(rs);
        loadinglt := false;
        loadingls := true;

        signal(mutex);
        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;
            signal(lt[0]);
            signal(lt[1]);
            signal(lt[2]);
            signal(ls);
            loadinglt := false;
            loadingls := false;

            signal(mutex);
            return true;
        end
        else begin //No singles loaded
            release(mutex);
            return false;
        end
    end
end
end
end
end
end

```

```

else begin //No triples loaded
    release(mutex);
    return false;
end
end
end
end;
end class;

type skier = class
    var sl: class 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: class ski_lift;

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

```

I assume the class 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.

(3) Ski lift problem with conditional critical region statements

```
type ski_lift = class
  var v: shared record
    ltcoun: array[0..2] of integer; // The number of people, not the number of triples
    ltcurrent: array[0..2] of integer;
    rtcoun: array[0..2] of integer; // The number of people, not the number of triples
    rtcurrent: array[0..2] of integer;
    lscoun: integer;
    lscurrent: integer;
    rscoun: integer;
    rscurrent: integer;
    loadinglt: boolean; // whether or not the left triple line has priority
    loadingls: boolean; // whether or not the left single line has priority
  end;

begin {initialize}
  v.ltcoun[0] := 0;
  v.ltcoun[1] := 0;
  v.ltcoun[2] := 0;
  v.ltcurrent[0] := -1;
  v.ltcurrent[1] := -1;
  v.ltcurrent[2] := -1;
  v.rtcoun[0] := 0;
  v.rtcoun[1] := 0;
  v.rtcoun[2] := 0;
  v.rtcurrent[0] := -1;
  v.rtcurrent[1] := -1;
  v.rtcurrent[2] := -1;
  v.lscoun := 0;
  v.lscurrent := -1;
  v.rscoun := 0;
  v.rscurrent := -1;
  v.loadinglt := true;
  v.loadingls := true;
end;

procedure entry get_in_line()
  var i, j: integer;
  begin
    region v do begin
      if ( (lscoun-lscurrent) < 2 * (ltcoun[0]-ltcurrent[0]))
        && ((lscoun-lscurrent) < 2 * (rtcoun[0]-rtcurrent[0]))
        && ((lscoun-lscurrent) < (rscoun-rscurrent)) )
      then begin
        //Load in the left single line
        i := lscoun;
        lscoun := lscoun + 1;
        await(i = lscurrent);
      end
      else if ( (rscoun-rscurrent) < 2 * (ltcoun[0]-ltcurrent[0]))
        && ((rscoun-rscurrent) < 2 * (rtcoun[0]-rtcurrent[0]))
        && ((rscoun-rscurrent) <= (lscoun-lscurrent)) )
      then begin
        //Load in the right single line
        i := rscoun;
        rscoun := rscoun + 1;
        await(i = rscurrent);
      end
      else if ( ((ltcoun[0]-ltcurrent[0]) < (rtcoun[0]-rtcurrent[0]))
        || ((ltcoun[1]-ltcurrent[1]) < (rtcoun[1]-rtcurrent[1]))
        || ((ltcoun[2]-ltcurrent[2]) < (rtcoun[2]-rtcurrent[2])) )
      then begin
        //Load in the left triple line
        j := (ltcoun[0] + ltcoun[1] + ltcoun[2]) mod 3;
        i := ltcoun[j];
        ltcoun[j] := ltcoun[j] + 1;
        await(i = ltcurrent[j]);
      end
      else begin
        //Load in the right triple line
        j := (rtcoun[0] + rtcoun[1] + rtcoun[2]) mod 3;

```

```

        i := rtcoun[t[j];
        rtcoun[t[j] := rtcoun[t[j] + 1;
        await(i = rtcurren[t[j]);
    end
end
end;

procedure entry load_lift() : boolean
var   ltloaded:   boolean;
      rtloaded:   boolean;
      lsloaded:   boolean;
      rsloaded:   boolean;
begin
    region v do begin
        if ( (ltcount[0]-ltcurrent[0]) > 0
            && (ltcount[1]-ltcurrent[1]) > 0
            && (ltcount[2]-ltcurrent[2]) > 0
        then
            ltloaded := true;
        else
            ltloaded := false;

        if ( (rtcoun[t[0]-rtcurren[t[0]) > 0
            && (rtcoun[t[1]-rtcurren[t[1]) > 0
            && (rtcoun[t[2]-rtcurren[t[2]) > 0
        then
            rtloaded := true;
        else
            rtloaded := false;

        if (lscoun[t-lscurren[t] > 0 then
            lsloaded := true;
        else
            lsloaded := false;

        if (rscoun[t-rscurren[t] > 0 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
                        ltcurrent[0] := ltcurrent[0] + 1;
                        ltcurrent[1] := ltcurrent[1] + 1;
                        ltcurrent[2] := ltcurrent[2] + 1;
                        lscurren[t := lscurren[t + 1;

                        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
                        ltcurrent[0] := ltcurrent[0] + 1;
                        ltcurrent[1] := ltcurrent[1] + 1;
                        ltcurrent[2] := ltcurrent[2] + 1;
                        rscurren[t := rscurren[t + 1;

                        loadinglt := false;
                        loadingls := true;

                        return true;
                    end
                end
            end
        end
    end
end

```

```

        else return false; //No singles loaded
    end
end
else begin
    //right single has priority
    if rsloaded then begin
        //Right single is loaded
        // Load left triple and right single
        lcurrent[0] := lcurrent[0] + 1;
        lcurrent[1] := lcurrent[1] + 1;
        lcurrent[2] := lcurrent[2] + 1;
        rcurrent := rcurrent + 1;

        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
            lcurrent[0] := lcurrent[0] + 1;
            lcurrent[1] := lcurrent[1] + 1;
            lcurrent[2] := lcurrent[2] + 1;
            lcurrent := lcurrent + 1;

            loadinglt := false;
            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
                rcurrent[0] := rcurrent[0] + 1;
                rcurrent[1] := rcurrent[1] + 1;
                rcurrent[2] := rcurrent[2] + 1;
                lcurrent := lcurrent + 1;

                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
                    rcurrent[0] := rcurrent[0] + 1;
                    rcurrent[1] := rcurrent[1] + 1;
                    rcurrent[2] := rcurrent[2] + 1;
                    rcurrent := rcurrent + 1;

                    loadinglt := true;
                    loadingls := true;

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

```

```

//right single has priority
if rsloaded then begin
    //Right single is loaded
    // Load right triple and right single
    rcurrent[0] := rcurrent[0] + 1;
    rcurrent[1] := rcurrent[1] + 1;
    rcurrent[2] := rcurrent[2] + 1;
    rscurrent := rscurrent + 1;

    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
        rcurrent[0] := rcurrent[0] + 1;
        rcurrent[1] := rcurrent[1] + 1;
        rcurrent[2] := rcurrent[2] + 1;
        lscurrent := lscurrent + 1;

        loadinglt := true;
        loadingls := false;

        return true;
    end
    else return false; // No singles loaded
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
                rcurrent[0] := rcurrent[0] + 1;
                rcurrent[1] := rcurrent[1] + 1;
                rcurrent[2] := rcurrent[2] + 1;
                lscurrent := lscurrent + 1;

                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
                    rcurrent[0] := rcurrent[0] + 1;
                    rcurrent[1] := rcurrent[1] + 1;
                    rcurrent[2] := rcurrent[2] + 1;
                    rscurrent := rscurrent + 1;

                    loadinglt := true;
                    loadingls := true;

                    return true;
                end
                else return false; // No singles loaded
            end
        end
    end
end
end
else begin
    //right single has priority

```

```

if rsloaded then begin
    //Right single is loaded
    // Load right triple and right single
    rcurrent[0] := rcurrent[0] + 1;
    rcurrent[1] := rcurrent[1] + 1;
    rcurrent[2] := rcurrent[2] + 1;
    rscurrent := rscurrent + 1;

    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
        rcurrent[0] := rcurrent[0] + 1;
        rcurrent[1] := rcurrent[1] + 1;
        rcurrent[2] := rcurrent[2] + 1;
        lscurrent := lscurrent + 1;

        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
                lcurrent[0] := lcurrent[0] + 1;
                lcurrent[1] := lcurrent[1] + 1;
                lcurrent[2] := lcurrent[2] + 1;
                lscurrent := lscurrent + 1;

                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
                    lcurrent[0] := lcurrent[0] + 1;
                    lcurrent[1] := lcurrent[1] + 1;
                    lcurrent[2] := lcurrent[2] + 1;
                    rscurrent := rscurrent + 1;

                    loadinglt := false;
                    loadingls := true;

                    return true;
                end
                else return false; // No singles loaded
            end
        end
    end
end
end
else begin
    //right single has priority
    if rsloaded then begin
        //Right single is loaded
        // Load left triple and right single
        lcurrent[0] := lcurrent[0] + 1;

```

```

        ltcurent[1] := ltcurent[1] + 1;
        ltcurent[2] := ltcurent[2] + 1;
        rscurrent := rscurrent + 1;

        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
        ltcurent[0] := ltcurent[0] + 1;
        ltcurent[1] := ltcurent[1] + 1;
        ltcurent[2] := ltcurent[2] + 1;
        lscurrent := lscurrent + 1;

        loadinglt := false;
        loadingls := false;

        return true;
    end
    else return false; // No singles loaded
end
end
end
end
else return false; // No triples loaded
end
end
end region
end;
end monitor;

type skier = class
    var sl: class 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: class ski_lift;

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

```

Some assumptions:

- The integer variables are allowed to become arbitrarily large. In the real world, we could add a lot of messy code to mod both the xxcount and xxcurrent variables.
- I assume the class 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.