Semaphores

Dekker's algorithm solves the mutual exclusion problem on a shared memory machine with no support from the hardware or software. *Semaphores* are a higher level concept than atomic instructions. They are atomic actions and usually implemented within the operating system.

A semaphore S is a non-negative integer variable that has exactly two operations defined for it.

P(S) If $S > 0$ then $S = S-1$, otherwise suspend the process.

V(S) If there are processes suspended on this semaphore wake one of them, else $S = S + 1$.

An important point is that V(S), as it is currently defined, does not specify which of the suspended processes to wake.

Semaphore Invariants

The following invariants are true for semaphores.

$$S \geq 0$$

$$S = S0 + \#V - \#P$$

where $S0$ is the initial value of semaphore S.

Mutual Exclusion

```
sem mutex := 1
const N := 20

process p(i := 1 to N)
    do true ->
        Non_critical_Section
        P(mutex)
        Critical_Section
        V(mutex)
    od
end
```

Theorem Mutual exclusion is satisfied.

Proof: Let #CS be the number of processes in their critical sections. We need to prove that the following is an invariant.

$$\#CS + mutex = 1$$

i)	$\#CS = \#P - \#V$	;from the program structure
ii)	$mutex = 1 + \#V - \#P$	;semaphore invariant
iii)	$mutex = 1 - \#CS$	;from i) and ii)
iv)	$\#CS + mutex = 1$	; from iii)

QED

Theorem The program cannot deadlock.

Proof: This would require all processes to be suspended in their P(mutex) operations. Then mutex = 0 and #CS = 0 since no process is in its critical section. The critical section invariant just proven is :

$$\#CS + mutex = 1 \Rightarrow 0 + 0 = 1$$

which is impossible.

Types of Semaphores

What we defined earlier is a general semaphore. A *binary semaphore* is a semaphore that can only take the values 0 and 1.

The choice of suspended process to wake gives the following definition.

Blocked-set semaphore Awakens any one of the suspended processes.

Blocked-queue semaphore The suspended processes are kept in FIFO and are awakened in the same order they were suspended. This is the type of semaphore implemented in the SR language.

Busy-wait semaphore The value of the semaphore is tested in a busy wait loop, with the test being atomic. There may be interleavings between cycles of the loop.

Theorem With busy-wait semaphores, starvation is possible.

Proof: Consider the following execution sequence for 2 processes.

i) P1 executes P(mutex) and enters its critical section.

ii) P2 executes P(mutex), finds mutex = 0 and loops.

iii) P1 finishes critical section, executes V(mutex) loops back and execute P(mutex) and enters its critical section.

iv) P2 test mutex, finds mutex = 0, and loops.

Theorem With blocked-queue semaphores, starvation is impossible.

Proof: If P1 is blocked on mutex there will be at most N-1 processes ahead of P1 in the queue. Therefore after N-1 V(mutex) P1 will enter its critical section.

Theorem With blocked-set semaphores, starvation is possible for $N \geq 3$.

Proof: If there are 3 processes it is possible to construct an execution sequence such that there are always 2 processes blocked on a semaphore. V(mutex) is required to only wake one of them, so it could always ignore one and leave that process starved.

The Producer-Consumer Problem

This type of problem has two types of processes:

Producers processes that, due to some internal activity, produce data to be sent to consumers.

Consumers processes that on receipt of a data element consume the data in some internal computation.

We could connect these processes in a synchronous manner, such that data is only transmitted when a producer is ready to send it and a consumer is ready to receive it. A more flexible method is to connect the producers and the consumers by a buffer which is a *queue*.

If this was an infinite buffer (wow!) then the following invariants should hold true for the buffer.

$$\#elements \geq 0$$

$$\#elements = 0 + in\_pointer - out\_pointer$$

These invariants are exactly the same as the semaphore invariants with a semaphore called *elements* and an initial value 0.

```
var buffer [?]:int
var in_pointer:int := 0, out_pointer:int := 0
sem elements := 0

process producer
    do true ->
        buffer [in_pointer] := produce ()
        in_pointer := in_pointer + 1
        V(elements)
    od
end

process consumer
var i:int
    do true ->
        P (elements)
        i := buffer [out_pointer]
        out_pointer := out_pointer + 1
        consume (i)
end
```

This can be modified for real bounded circular buffers by using another semaphore to count the empty places in the buffer.

As an exercise prove the following:

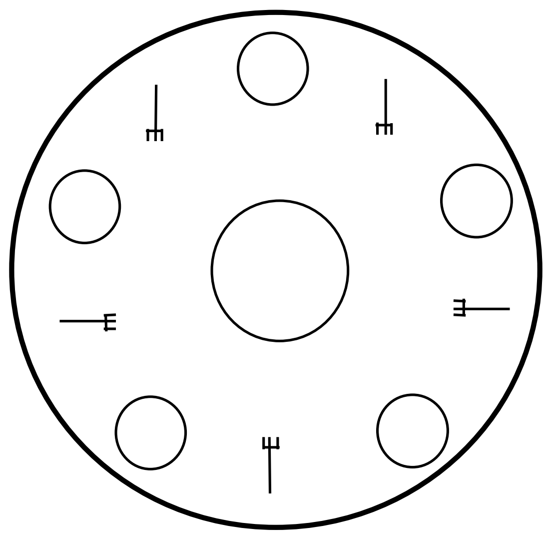
(i) the program cannot deadlock, (ii) neither process is starved and (iii) the program never removes data from an empty buffer or appends data to a full buffer.

```
const N := 100
var buffer [N]:int
var in_pointer:int := 0, out_pointer:int := 0
sem elements := 0
sem spaces := N

process producer
var i:int
    do true ->
        i := produce ()
        P (spaces)
        buffer [in_pointer] := i
        in_pointer := (in_pointer + 1) mod N
        V (elements)
    od
end

process consumer
var i:int
    do true ->
        P (elements)
        i := buffer [out_pointer]
        out_pointer := (out_pointer + 1) mod N
        V (spaces)
        consume (i)
end
```

The Dining Philosophers Problem

The Problem

An institution hires five philosophers to solve a very difficult problem. Each philosopher only engages in two activities - *thinking* and *eating*. Meals are taken in the diningroom which has a table set with five plates and five forks. In the centre of the table is a bowl of spaghetti that is endlessly replenished. The philosophers, not being the most dextrous of individuals, requires two forks to eat; and may only pick up the forks immediately to his left and right.



For this system to operate correctly it is required that:

a) A philosopher eats only if he has two forks.

b) No two philosophers can hold the same fork simultaneously.

c) No deadlock.

d) No individual starvation.

e) Efficient behaviour under the absence of contention.

This problem is a generalisation of multiple processes accessing a set of shared resources; e.g. a network of computers accessing a bank of printers.

First Attempted Solution

Model each fork as a semaphore. Then each philosopher must wait (execute a P operation) on both the left and right forks before eating.

```
sem fork [5] := ([5] 1)
process philosopher (i := 1 to 5)
    do true ->
        Think ( )
        P(fork [i])
        P(fork [(i+1) mod 5]
        Eat ( )
        V(fork [i])
        V(fork [(i+1) mod 5]
    od
end
```

This is called a *symmetric solution* since each task is identical. Symmetric solutions have many advantages, particularly when it comes to load-balancing.

We can prove that no fork is ever held by two philosophers since `Eat ( )` is the critical section of each fork. If $\#P_i$ is the number of philosophers holding fork *i* then we have:

$$Fork(i) + \# P_i = 1$$

Since a semaphore is non-negative then $\#P_i \leq 1$.

However, this system can deadlock under an interleaving in which all five philosophers pick up their left forks together; i.e. all processes execute `P(fork [i])` before `P(fork [(i+1) mod 5]`. There are two solutions. One is to make one of the philosophers pick up the right fork before the left fork (asymmetric solution); the other is to only allow four philosophers into the room at any one time.

A Symmetric Solution

```
sem Room := 4
sem fork [5] := ([5] 1)
```

```
process philosopher (i := 1 to 5)
    do true ->
        Think ( )
        P (Room)
        P(fork [i])
        P(fork [(i+1) mod 5]
        Eat ( )
        V(fork [i])
        V(fork [(i+1) mod 5]
        V (Room)
    od
end
```

This solves the deadlock problem.

Theorem Individual starvation cannot occur.

Proof For a process to starve it must be forever blocked on one of the three semaphores, Room, fork [i] or fork [(i+1) mod 5].

Room semaphore

If the semaphore is a blocked-queue semaphore then process *i* is blocked only if Room is 0 indefinitely. This would require the other four philosophers to be blocked on their left forks, since if one of them can get two forks he will finish, put down the forks and signal Room (V(Room)). So this case will follow from the fork [i] case.

fork [i]

If philosopher *i* is blocked on his left fork, then philosopher *i-1* must be hold his right fork. Therefore he is either eating or signalling he is finished with his left fork, and will eventually put down his right fork which is philosopher *i*'s left fork.

fork [(i+1) mod 5]

If philosopher *i* is block on his right fork, this means that philosopher *(i+1)* has taken his left fork and never released it. Since eating and signalling cannot block, philosopher *(i+1)* must be waiting for his right fork, and must all the other philosophers by induction: $i+j$, $0 \leq j \leq 4$. However the Room semaphore invariant only 4 philosophers can be in the room, so philosopher *i* cannot be blocked on his right fork.

Readers-Writers Problem

The Problem

Two kinds of processes, readers and writers, share a database. Readers execute transactions that examine the database, writers execute transactions that examine and update the database. Given that the database is initially consistent, then to ensure that the database remains consistent, a writer process must have exclusive access to the database. Any number of readers may concurrently examine the database.

Obviously, as far as a writer process is concerned, updating the database is a critical section that cannot be interleaved with any other process.

```
const M:int := 20, N:int := 5
var nr:int :=0
sem mutexR := 1
sem rw := 1

process reader (i:= 1 to M)
    do true ->
        P (mutexR)
        nr := nr + 1
        if nr = 1 -> P (rw) fi
        V (mutexR)
        Read_Database ( )
        P (mutexR)
        nr := nr - 1
        if nr = 0 -> V (rw) fi
        V (mutexR)
    od
end
```

```
process writer (i:= 1 to N)
    do true ->
        P (rw)
        Update_Database ( )
        V (rw)
    od
end
```

This is called the *readers' preference* solution since if some reader process is accessing the database and a reader process and a writer process arrive at their entry protocols then the reader process will always have preference over the writer process.

This is not a fair solution since this solution always gives readers precedence over writers, a continual stream of readers will block any writer process from updating the database.

To make the solution fair we need to use a *split binary semaphore*, that is several semaphores who together have the property that their sum is 0 or 1. We also need to count the number of suspended reader processes and suspended writer processes.

This technique is called *passing the baton*.

```
const M:int := 20, N:int := 5
var nr:int :=0, nw:int := 0
var sr:int := 0, sw:int := 0   # count of suspended
                               # readers and writers
sem e := 1, r := 0, w := 0     # 0 ≤ (e+r+w) ≤ 1


process reader (i:= 1 to M)
    do true ->
        P (e)
        if nw > 0 ->
            sr:= sr + 1; V (e); P (r)
        fi
        nr := nr + 1
        if sr > 0 ->
            sr := sr - 1; V (r)
        [] sr = 0 -> V(e)
        fi

        Read_Database ( )

        P (e)
        nr := nr - 1
        if nr = 0 and sw > 0 -> sw := sw - 1; V (w)
        [] nr >0 or sw = 0 -> V (e)
        fi
    od
end
```

```
process writer (i:= 1 to N)
    do true ->
        P (e)
        if nr > 0 or nw > 0 ->
            sw := sw + 1; V (e); P (w)
        fi
        nw := nw + 1
        V (e)

        Update_Database ( )

        P (e)
        nw := nw - 1
        if sr > 0 -> sr := sr -1; V (r)
        [] sw > 0 -> sw := sw - 1; V (w)
        [] sr = 0 and sw = 0 -> V(e)
        fi
    od
end
```

Monitors

The main disadvantage with semaphores is that they are a low level programming construct. In a large project, with many programmers, if one programmer forgets to do V( ) operation on a semaphore after a critical section, then the whole system can deadlock.

What is required is a higher level construct that groups the responsibility for correctness into a few modules.

Monitors are such a construct. These are an extension of the monolithic found in operating systems. They *encapsulate* a set of procedures, and the data they operate on, into single modules, called monitors, and guarantee that only one process can execute a procedure in the monitor at any given time (mutual exclusion). Of course different processes can execute procedures from different monitors at the same time.

Condition Variables

Synchronisation is achieved by using *condition variables*. These are data structures that have 3 operations defined for them.

wait (C)	The process that called the monitor containing this operation is suspended in a FIFO queue associated with C. Mutual exclusion on the monitor is released.
signal (C)	If the queue associated with C is non-empty, then wake the process at the head of the queue.
non-empty (C)	Returns true if the queue associated with C is non-empty.

If a monitor guarantees mutual exclusion, and a process uses the *signal* operation and awakens another process suspended in the monitor, is there not two processes in the same monitor at the same time? Yes.

To solve this problem, several signalling mechanisms can be implemented, the simplest of which is the *signal and continue mechanism*. Under these rules the procedure in the monitor that signals a condition variable is allowed to continue to completion, so the *signal* operation should be at the end of the procedure. The process which was suspended on the condition variable, but is now awoken, is scheduled for *immediate resumption* on the exiting of the procedure which signalled the condition variable.

Readers/Writers Problem

```
_monitor (RW_controller)
    op request_read ( ), release_read ( )
    op request_write ( ), release_write ( )
_body (RW_controller)
    var nr:int := 0, nw:int := 0
    _condvar (ok_to_read)
    _condvar (ok_to_write)

    _proc (request_read ( ))
        do nw > 0 -> _wait (ok_to_read) od
        nr := nr + 1
    _proc_end

    _proc (release_read ( ))
        nr := nr - 1
        if nr = 0 -> _signal (ok_to_write) fi
    _proc_end

    _proc (request_write ( ))
        do nr > 0 or nw > 0 -> _wait (ok_to_write) od
        nw := nw + 1
    _proc_end

    _proc (release_write ( ))
        nw := nw -1
        _signal (ok_to_write)
        _signal_all (ok_to_read)
    _proc_end
_monitor_end
```

File rw_controller.m

```
resource main ( )
    import RW_controller

    process reader (i := 1 to 20)
        RW_controller.request_read ( )
        Read_Database ( )
        RW_controller.release_read ( )
    end

    process writer (i := 1 to 5)
        RW_controller.request_write ( )
        Update_Database ( )
        RW_controller.release_write ( )
    end
end
```

File reader_writer.sr

Emulating Semaphores with Monitors

Both semaphores and monitors are concurrent programming primitives of equal power. Monitors are just a higher level construct.

```
_monitor semaphore
    op p ( ), v ( )
_body semaphore
    var s:int := 0
    _condvar (not_zero)
```

```
    _proc (p ( ))
        if s=0 -> _wait (not_zero) fi
        s := s - 1
    _proc_end

    _proc (v ( ))
        if _not_empty (not_zero) = true ->
            _signal (not_zero)
        [] else ->
            s := s + 1
        fi
    _proc_end

_monitor_end
```

Emulating Monitors by Semaphores

Firstly we need blocked-queue semaphores (SR is OK) and secondly we need to implement the *signal and continue* mechanism. We do this with a variable c_count, one semaphore, s, to ensure mutual exclusion and another semaphore, c_semaphore, to act as the condition variable. Then we translate _wait (s) to:

```
c_count := c_count + 1
V (s)
P (c_semaphore)
c_count := c_count - 1
```

and _signal (s) to:

```
if c_count > 0 -> V (c_semaphore)
```

```
[] else -> V (s)
fi
```

The Dining Philosophers Problem with Monitors

Using monitors yields a nice solution, since with semaphores you cannot test two semaphores simultaneously. The monitor solution maintains an array fork which counts the number of free forks available to each philosopher.

```
_monitor (fork_monitor)
    op take_fork (i:int), release_fork (i:int)
_body (fork_monitor)
    var fork [5]:int := ([5] 2)
    _condvarl (ok_to_eat, 5)    # define an array of
                                # condition variables
    _proc (take_fork (i))
        if fork [i] != 2 -> _wait (ok_to_eat [i]) fi
        fork [(i-1) mod 5] := fork [(i-1) mod 5] - 1
        fork [(i+1) mod 5] := fork [(i+1) mod 5] - 1
    _proc_end

    _proc (release_fork (i))
        fork [(i-1) mod 5] := fork [(i-1) mod 5] - 1
        fork [(i+1) mod 5] := fork [(i+1) mod 5] - 1
        if fork [(i+1) mod 5] = 2 ->
            _signal (ok_to_eat [(i+1) mod 5] )
        fi
```

```
        if fork [(i-1) mod 5] = 2 ->
            _signal (ok_to_eat [(i-1) mod 5] )
        fi
    _proc_end

_monitor_end


resource main ( )
    import fork_monitor

    process philosopher (i:= 1 to 5)
        do true ->
            Think ( )
            fork_monitor.take_fork (i)
            Eat ( )
            fork_monitor.release_fork (i)
        od
    end
end
```

<u>Theorem</u> Solution doesn't deadlock.

Proof: Let #E be the number of philosophers who are eating, and have therefore taken both forks. Then the following invariants are true from the program.

$$Non-empty(\mathrm{ok\_to\_eat}[i]) \Rightarrow \mathrm{fork}[i] < 2$$

$$\sum_{i=1}^{5} \mathrm{fork}[i] = 10 - 2(\#\,\mathrm{E})$$

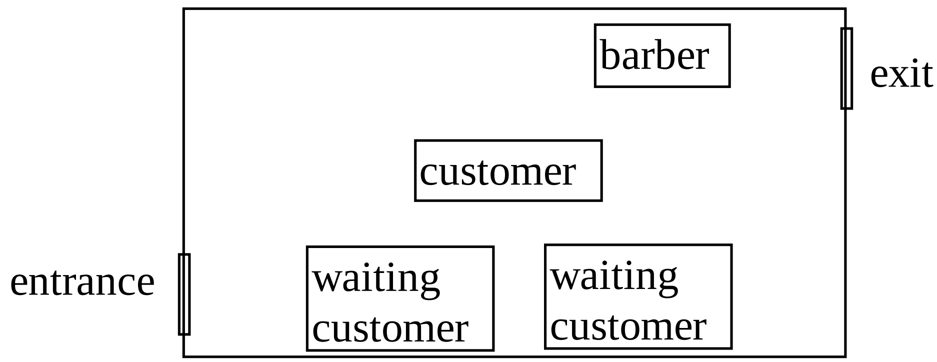
Deadlock implies #E=0 and all philosophers are enqueued on ok_to_eat. If they are all enqueued then the first equation implies $\sum \text{fork}[i] \leq 5$ and if no philosopher is eating, then the second equation implies $\sum \text{fork}[i] = 10$. The contradiction implies that the solution does not deadlock.

However, individual starvation can occur. How? What is the solution?

The Sleeping Barber Problem

This is a generalisation of *rendezvous* synchronisation in *client/server* architectures.

Problem A small barber shop has two doors, an entrance and an exit. Inside is a barber who spends all his life serving customers, one at a time. When there are none in the shop, he sleeps in his chair. If a customer arrives and finds the barber asleep he awakens the barber, sits in the customer's chair and sleeps while his hair is being cut. If a customer arrives and the barber is busy cutting hair, the customer goes asleep in one of the two waiting chairs. When the barber finishes cutting a customer's hair, he awakens the customer and holds the exit door open for him. If there are any waiting customers, he awakens one and waits for the customer to sit in the customer chair, otherwise he goes to sleep.



The barber and customers are interacting processes, and the barber shop is the monitor in which they react.

```
_monitor (barber_shop)
    op get_haircut ( ), get_next_customer ( )
    op finish_cut ( )
_body (barber_shop)
    var barber:int := 0, chair:int := 0, open:int := 0
    _condvar (barber_available)  # when barber > 0
    _condvar (chair_occupied)    # when chair > 0
    _condvar (door_open)         # when open > 0
    _condvar (customer_left)     # when open = 0

    # called by customer
    _proc (get_haircut ( ))
        do barber = 0 -> _wait (barber_available) od
        barber := barber -1
        chair := chair + 1
        _signal (chair_occupied)
        do open = 0 -> _wait (door_open) od
```

```
        open := open - 1
        _signal (customer_left)
    _proc_end

    # called by barber
    _proc (get_next_customer ( ))
        barber : barber + 1
        _signal (barber_available)
        do chair = 0 -> _wait (chair_occupied) od
        chair := chair - 1
    _proc_end

    #called by barber
    _proc (finished_cut ( ))
        open := open + 1
        _signal (door_open)
        do open > 0 -> _wait (customer_left) od
    _proc_end

_monitor_end
```

In general, when many different processes have to *rendezvous*, the solution technique is to use a monitor to create an “environment” in which the processes can *rendezvous*.

Java: Threads and Monitors

Java supports concurrency via ***threads***. These threads are lightweight processes and they synchronise via monitors.

Threads

A Java thread is a lightweight process that has its own stack and execution context, but also has access to all variables in its scope. Threads are programmed by either extending the **Thread** class or by implementing the **Runnable** interface. Both of these are part of the standard **java.lang** package.

An instance of a thread is created by:

```
Thread myProcess = new Thread ( );
```

To start the new thread running you simply execute:

```
MyProcess.start ( );
```

The **start** method invokes a **run** method in the thread. Since we haven't defined the **run** method, the code above does nothing. We can define the **run** method by extending the **Thread** class.

```
class myProcess extends Thread
{
    public void run ( )
    {
        System.out.println (“Hello from the thread”);
    }
}

myProcess p = new myProcess ( );
p.start ( );
```

If you don’t need to keep a reference to the new thread we can do away with **p** and simply write:

```
new myProcess ( ).start ( );
```

As well as extending the **Thread** class, we can create lightweight processes by implementing the **Runnable** interface. The advantage of this approach is that you can make one of your own classes, or a system-defined class, into a process. You cannot do this with threads since Java only allows you to extend one class at a time.

Using the **Runnable** interface the previous example becomes:

```
Class myProcess implements Runnable
{
    public void run ( )
    {
        System.out.println ("Hello from the thread");
    }
}

Runnable p1 = new myProcess  ( );
New Thread (p1). start ( );
```

The preferred way to terminate a thread is to allow the **run** method to terminate. There is a **stop** method that terminates a thread no matter what it is doing, but his has been deprecated in Java2 (for very good reasons).

If a thread has nothing immediate to do (for example it updates the screen every second) then you should suspend the thread by putting it to sleep for a specified time. There are two sleep methods, **static void sleep (long milliseconds)** and **static void sleep (long milliseconds, int nanoseconds)**. Be careful because most operating systems have a minimum time resolution greater than 1 millisecond, so the JVM rounds the requested sleep period to the smallest time resolution the operating system supports.

The **join** method waits for the completion of the specified thread and provides some basic synchronisation. It can be supplied with a timeout period (like the **sleep** methods).

```
…
    try
    {
        otherThread.join (1000);     // wait for 1 sec
    }
    catch (InterruptedEception e ) {}
…
```

Synchronisation in Threads

Conceptually threads in Java execute concurrently and therefore could simultaneously access shared variables. To prevent two threads from interfering with each other when updating a shared variable, Java provides synchronisation via a slimed-down monitor.

Java’s keyword **synchronized** provides mutual exclusion and can be used with a group of statements or with an entire method.

The following class will potentially have problems if its **update** method is executed by several threads concurrently.

```
class  Problematic
{
    private int data = 0;
    public void update ( )
    {
        data++;
    }
}
```

An executing instance of the **update** method needs exclusive access to the variable **data**.

```
class ExclusionByMethod
{
    private int data = 0;
    public synchronized void update ( )
    {
    data++;
    }
}
```

This is a simple monitor where the monitor's permanent variables are private variables in the class; and the monitor procedures are implemented as **synchronized** methods. There is only one lock per object in Java so when a synchronized method is invoked it waits to obtain the lock, execute the method, and then releases the lock.

Another way to implement mutual exclusion is to use the **synchronize** statement within the body of a method.

```
class ExclusionByGroup
{
    private int data  = 0;
    public void update ( )
    {
        synchronized (this)    // lock this object for the
        {                              // following group of statements
            data++;
        }
    }
}
```

The keyword **this** refers to the object invoking the **update** method. The lock is obtained on the invoking object.

A **synchronized** statement specifies that the following group of statements is executed as an atomic, non-interruptible, action. A **synchronized** method is equivalent to a monitor procedure.

While Java does not explicitly support condition variables, there is one implicitly declared for each synchronised object. The **wait** and **notify** methods are similar to the **wait** and **signal** operations in SR but they can only be executed within **synchronized** portions of the code (i.e. when an object is locked).

The **wait** method releases the lock on an object and suspends the executing thread. There is a single delay queue per object (usually a FIFO but not necessarily).

The **notify** method awakens the thread at the front of the object's delay queue. **notify** has *signal and continue* semantics, so the thread invoking **notify** continues to hold the lock on the object. The awakened thread will execute at some future time when it can reacquire the lock on the object. Java also has a **notifyAll** method that is similar to **signal_all** in SR.

Readers/Writers Problem in Java

```
class ReadersWriters
{
    private int data = 0;        // our database
    private int nr = 0;


    private synchronized void startRead ( )
    {
        nr++;
    }


    private synchronized void endRead ( )
    {
        nr--;
        if (nr == 0)
            notify ( );          // awaken a waiting writer
    }
```

```
    public void read ( )
    {
        startRead ( );
        System.out.println (“read: ” + data);
        endRead ( );
    }


    public synchronized void write ( )
    {
        while (nr > 0)
            try
            {
                wait ( );                  // delay any active readers
            }
            catch (InterruptedException ex)
            {
                return;
            }
        data++;
        System.out.println (“write: ” + data);
        notify ( );                        // awaken a waiting writer
    }
}


class Reader extends Thread
{
    int rounds;
    ReadersWriters RW;
```

```
    Reader (int rounds, ReadersWriters RW)
    {
        this.rounds = rounds;
        this.RW = RW;
    }


    public void run ( )
    {
        for (int i = 0; i < rounds; i++)
            RW.read ( );
    }
}


class Writer extends Thread
{
    int rounds;
    ReadersWriters RW;


    Reader (int rounds, ReadersWriters RW)
    {
        this.rounds = rounds;
        this.RW = RW;
    }


    public void run ( )
    {
        for (int i = 0; i < rounds; i++)
            RW.write ( );
    }
}
```

```
class RWProblem
{
    static ReadersWriters RW = new ReadersWriters ( );

    public static void main (String[ ] args)
    {
        int rounds = Integer.parseInt (args[0], 10);
        new Reader (rounds, RW).start ( );
        new Writer (rounds, RW).start ( );
    }
}
```

This piece of Java code gives reader preference over writers. As an exercise can you fgiure out how to make it fair?