Class Notes on Multi-Threading

A. Udaya Shankar
shankar@cs.umd.edu

November 9, 2012

Contents

1 Multi-threaded programs

- Multiple interacting threads arise because
  - application is inherently distributed (data at different locations), or
  - need for better performance via parallel hardware
- Threads interact via shared memory or message passing (e.g., signals, send/receive).
  - Shared memory can be implemented using message passing, and vice versa.
  - At lower levels (hardware/physical), it's necessarily message passing
- A code chunk execution x **interferes** with a code chunk execution y if one writes to a memory location that the other reads or writes.
- **Atomicity:** The execution of a code chunk S by a thread t is defined to be **atomic** if while t is executing S (i.e., has started and not yet finished), no other thread influences or is influenced by t's execution of S (i.e., executes an interfering statement).

  Atomicity is achieved by exclusion in time or space.
- **Ordering:** Code chunk S is executed only after code chunk T is executed.

  One easy (but not efficient) way to achieve this on a shared-memory machine: set a flag at the end of T and have S busy wait for the flag to be set.
- **Atomicity and ordering:** S is atomically executed only after T has completed execution.
- **Achieving atomicity:** On single-CPU system, atomicity of S can be achieved by disabling external interrupts to the CPU.
  - If interrupts are enabled when S starts, they should be disabled at that point and enabled when S ends.
  - If interrupts are disabled when S starts, nothing should be done. (This happens if S is being called by some code chunk T which is to be executed atomically. So interrupts should not be enabled at the end of S.)
  - So is the following a solution?

    ```
    1: if interrupts enabled
    2:    flag ← true     // local variable
    3:    disable interrupts
    4: S
    5: if flag
    6:    enable interrupts
    ```

    Suppose interrupts are enabled when the thread is at 1, and the thread gets switched out just before 3 (after `flag` is updated). Can the thread be resumed with interrupts disabled? Would things go wrong if this happens?

- But disabling interrupts is not a good approach in general.
  - Can cause more important activities to be slowed or dropped (e.g., clock ticks).
  - Does not work in multi-processor system.
  - Not system-friendly.
  - Programming-friendly?
- Want a better synchronization construct.
  - Allow `S` to be atomic but interruptible by non-interfering `T`.
  - Allow ordering without busy waiting.
  - Higher-level programming language construct.
- **Typical programming-language synchronization constructs**
  - reads and writes of shared memory locations.
  - semaphores
  - locks and condition variables (aka monitors)
- **Typical synchronization problem to be solved:**
  - Given shared data structure X,
    boolean conditions $B_1, B_2, \cdots, B_n$ in X,
    code chunks $U_1, U_2, \cdots, U_n$ in X,
  - To ensure that U_i is atomically executed only when X satisfies B_i.

2 Locks

At any time, a lock is either available (aka free) or acquired (aka held) by some thread.

Lock constructs

- Initialization: `Lock lck` // Creates available lock `lck`.
- Acquire lock: `lck.acq()`
  - Callable only if thread does not hold the lock.
  - Thread gets past this statement only if lock is available; simultaneously acquires the lock. So thread blocks if `lck` is not available.
- Release lock: `lck.rel()`
  - Callable only if thread holds the lock.
  - Releases the lock. Non blocking.
- Progress:
  - Weak lock: thread at `lck.acq()` eventually gets past if lock is continuously free.
  - Strong lock: thread at `lck.acq()` eventually gets past if lock is repeatedly free (but not necessarily continuously free).

Achieving mutual exclusion

To have code chunks U_1, U_2, $\cdots$, U_n be executed mutually exclusively,

- Define lock `lck`.
- Surround each U_i by `lck.acq()` and `lck.rel()`.

Achieving ordering

Using locks alone, this can be achieved only with busy waiting.

3 Lock implementation for single-CPU system

Here is an implementation of a lock for a single-CPU system. It uses interrupt disabling and accesses PCBs.

```
Lock lck:
   lck.free ← true
   lck.queue ← []

lck.acq():
   // here on trap with interrupts disabled
   currentPcb ← current thread state with return address after lck.acq() call
   if lck.free
      lck.free ← false
      interrupt return
   else // lck not free
      move currentPcb to lck.queue
      Scheduler

lck.rel():
   // here on trap with interrupts disabled
   if lck.queue ≠ []
      move a pcb from lck.queue to runnable queue
   else
      lck.free ← true
   interrupt return
```

A strong lock can be achieved simply by using a fair queueing discipline (e.g., fifo) when removing a PCB from `lck.queue`.

4 Spin lock implementation using critical-section solution

The **critical section** problem is as follows. A program is executed by `N` threads, with ids `0`, $\cdots$, `N-1`. Sections of program by the threads are marked as "critical". The problem is to come up with `entry` and `exit` code chunks to surround each critical section so that:

- **Safety:** At any time at most one thread is in a critical section.
- **Progress:** Any thread that enters an `entry` code chunk eventually enters the associated critical section, provided no thread stays in a critical section forever.
- **Atomicity assumption:** Atomic reads and writes of memory words are the only atomicity that can be assumed by the `entry` and `exit` code chunks.

Clearly this problem is equivalent to that of implementing locks, with `entry` corresponding to acquire and `exit` to release. Thus a solution yields a lock implementation that requires only atomic reads and writes of memory words (and hence is applicable in a multi-CPU shared-memory system).

Because these lock implementations employ busy-waiting, the locks are referred to as **spin locks**.

4.1 2-process spin lock implementation

Here is a spin lock implementation using Peterson's critical section solution for the case of two processes, i.e., `N` equals `2`, thread ids are `0` and `1`. Thus this lock is a "2-process" lock, i.e., a lock that is accessed by only two threads.

Below, `i` and `j` range over `0` and `1`, such that `i` $\neq$ `j`. Thread `i` calls `acq(i)` to acquire the lock, and `rel(i)` to release the lock. The threads share three 1-bit variables: `flag[0]`, indicating if thread `0` is acquiring or holding the lock; `flag[1]`, just like `flag[0]` for thread `1`; and `turn`, indicating which thread has priority in case of contention. To acquire the lock, thread `i` sets `flag[i]` to true and `turn` to `j`, and then waits for `flag[j]` to be false or `turn` to equal `i`.

```
Shared variables:
   boolean flag[2] ← [false, false]
   turn ← 0     // can also be 1
```

```
acq(i):
 s1: flag[i] ← true
 s2: turn ← j
 s3: while (flag[j] and turn = j) skip;
     return
```

```
rel(i):
   flag[i] ← false
   return
```

Here is a proof that it works.

Safety: Suppose `i` leaves `s3` (in `acq(i)`). We need to show that `j` does not hold the lock at that time.

- Suppose `i` leaves `s3` because `flag[j]` is false. Then at that time thread `j` is neither in `acq(j)` nor does it hold the lock.
- Suppose `i` leaves `s3` because `flag[j]` is true and `turn` equals `i`, say at time t_0. Let `i`'s last execution of `s2` be at time t_1, where $t_1 < t_0$. Just after t_1, `flag[i]` and `turn` = `j` hold. Because `turn` = `i` holds at t_0, thread `j` must have executed `s2` at some time t_2 where $t_1 < t_2 < t_0$. Hence `flag[i]` and `turn` = `i` hold during $[t_2, t_0]$. Hence thread `j` is still in `s3`.

Progress: Suppose `i` calls `acq(i)` and reaches `s3` at time t_0. We need to show that `i` eventually leaves `s3`.

1. Suppose `flag[j]` stays false after some point t_1 where $t_0 \leq t_1$. Then `i` eventually leaves `s3` and returns.
2. Otherwise `flag[j]` is true at t_0 or becomes set at some later point (while `i` is in `s3`).
   - 2a. Suppose `flag[j]` is set (by `j`) at some point after t_0. Then at some later point `j` sets `turn` to `i`. After this `turn` stays `i` (because `j` is stuck in `s3`), and so `i` eventually leaves `s3`.
   - 2b. Suppose `flag[j]` is true at t_0 (i.e., it was set before t_0), and `j` has not yet executed `s2`. Then this reduces to case 2a.
   - 2c. Suppose `flag[j]` is true and `turn` is `i` before t_0 (i.e., `j` did `s2` before t_0). Then `i` is stuck in `s3` but `j` will eventually leave `s3`. After this it will eventually release the lock (otherwise progress holds vacuously), after which this reduces to case 1 or 2a.

4.2 N-process spin lock implementation

Here is a lock implementation using Lamport's bakery algorithm solution to the `N`-process critical section problem.

TBD

5 Lock implementation using spin locks and PCB queues

Our lock implementation for single-CPU systems involves disabling interrupts and manipulating PCB queues. Our lock implementations for multi-CPU shared memory systems do not involve disabling interrupts or manipulating PCB queues, but they do involve busy waiting. A busy waiting solution is acceptable only if the lock is to be held by threads for short durations. If that is not the case, it makes sense to have a lock implementation that manipulates PCB queues (as in the single-CPU case) but where atomicity of `acq` and `rel` are achieved using spin locks (instead of disabling interrupts).

```
Lock lck:
   lck.free ← true
   lck.queue ← []
   lck.spinLck

lck.acq():
   lck.spinLck.acq()
   currentPcb ← current thread state with return address after lck.acq() call
   if lck.free
      lck.free ← false
      lck.spinLck.rel()
      interrupt return
   else // lck not free
      move currentPcb to lck.queue
      lck.spinLck.rel()
      Scheduler

lck.rel():
   lck.spinLck.acq()
   if lck.queue ≠ []
      move a pcb from lck.queue to runnable queue
   else
      lck.free ← true
   lck.spinLck.rel()
   return
```

The overhead of busy waiting is now acceptable because the spin lock, namely `lck.spinLck`, is held for only short durations (specifically, the time to execute the body of `acq` or `rel`.

6 Spin lock implementation using test-and-set instruction

A test-and-set(x) instruction returns the old value of x and simultaneously sets x to true. Locks can be easily implemented if the hardware provides this instruction. Below we assume N threads, with ids 0, · · ·, N−1.

6.1 Safety-only solution

Here is a partial solution. It ensures that at most one thread holds the lock. It does not ensure that a thread waiting for the lock eventually acquires it if the lock becomes repeatedly free.

```
Shared variables:
   // false iff lock is free
   boolean lckd ← false
```

```
acq(i):   // i not used
   while test-and-set(lckd) skip;
   return
```

```
rel(i):  // i not used
   lckd ← false
   return
```

6.2 Safety and progress solution

Here is a complete solution. In addition to lckd, the threads share boolean variables waiting[0], · · ·, waiting[N−1], where waiting[i] indicates if thread i is waiting for the lock. When a thread j releases the lock, it looks in increasing (modulo-N) order for a waiting thread. If it finds one, it “passes” the lock to it; otherwise, it makes the lock free.

```
Shared variables:
   boolean lckd ← false
   boolean waiting[N] ← [false, · · ·, false]
```

```
acq(i):
   waiting[i] ← true
   key ← true
   while (waiting[i] and key)
      key ← test-and-set(lckd)
   waiting[i] ← false
   return
```

```
rel(i):
   j ← (i+1) mod N
   while (j ≠ i and not waiting[j])
      j ← (j+1) mod N
   if j = i
      lckd ← false
   else
      waiting[j] ← false
   return
```

7 Semaphores

Semaphores (aka Counting Semaphores) are the earliest synchronization construct.

At any time, a semaphore has a non-negative integer value. Unlike with a lock, there is no notion of a thread holding a semaphore (except when a semaphore is being used as a lock).

Semaphore operations

- Initialization: `Semaphore(N) sem`
  Creates semaphore `sem` with initial value `N`.
- `sem.P()` // text uses `acquire(sem)` or `wait(sem)`
  Thread gets past this statement only if `sem.value > 0` holds; simultaneously decreases `sem.value` by 1.
  So thread blocks until `sem.value > 0` holds.
- `sem.V()` // text uses `release(sem)` or `signal(sem)`
  Increments `sem.value` by 1.
  Non blocking.
- Progress:
  Weak: thread at `sem.P()` eventually gets if `sem.value > 0` holds continuously.
  Strong: thread at `sem.P()` eventually gets if `sem.value > 0` holds repeatedly.

Achieving mutual exclusion

To have code chunks U_1, U_2, $\cdots$, U_n be executed mutually exclusively,

- Define semaphore `mutex` initialized to `1`.
- Surround each U_i by `mutex.P()` and `mutex.V()`.

Achieving ordering (without busy waiting)

To have `S` execute only after `U` completes:

- Define semaphore `gate` initialized to `0`.
- Insert `gate.V()` at the end of `U`.
- Insert `gate.P()` at the start of `S`.

8 Semaphore implementation for single-CPU system

```
Semaphore(N) sem:
   sem.value ← N
   sem.queue ← []

sem.P():
   // here on trap with interrupts disabled
   currentPcb ← current thread state with return address after sem.P() call

   if (sem.val > 0)
      sem.val ← sem.val - 1
      interrupt return
   else // sem.val = 0
      move currentPcb to sem.queue
      Scheduler  // enables interrupts

sem.V():
   // here on trap with interrupts disabled
   if sem.queue ≠ []
      move a pcb from sem.queue to runnable queue
   else
      sem.val ← sem.val + 1
   interrupt return
```

9 Semaphore implementation using spin locks

To implement semaphores on shared memory systems without disabling interrupts, we can have the P and V operations do the same manipulations of PCB queues as in the single-CPU case, but use spin locks to achieve atomicity of these operations. (This is the same approach as in section 5.)

10 Bounded-buffer

Given buffer buff of max size N, and non-blocking code chunks append and remove, to obtain functions enQ and deQ such that:

- enQ and deQ can be called by multiple threads simultaneously.
- enQ(x) calls append(x) exactly once, waiting if buff.size = N holds.
- deQ calls remove exactly once. waiting if buff.size = 0 holds.

10.1 Solution BB1

```
Shared variables:
   Semaphore(1) mutex
   Semaphore(0) gateE  // enQ thread waits here if buff full
   int nE ← 0    // tracks number of enQ threads waiting on gateE
   Semaphore(0) gateD  // deQ thread waits here if buff empty
   int nD ← 0    // tracks number of deQ threads waiting on gateD
```

```
enQ(x):
   mutex.P()
   if buff.size = N
      nE ← nE + 1
      mutex.V()
  e0: gateE.P()
      nE ← nE - 1
   buff.append(x)
   if nD > 0
      gateD.V()
   else
      mutex.V()
```

```
deQ():
   mutex.P()
   if buff.size = 0
      nD ← nD + 1
      mutex.V()
  d0: gateD.P()
      nD ← nD - 1
   x ← buff.remove
   if nE > 0
      gateE.V()
   else
      mutex.V()
   return x
```

We now show that this works. Below, for brevity, we use the name of a semaphore to denote its value.

Atomic steps of program

First, note that there are six possible atomic steps in the program:

e1: from enQ-start to enQ-end.
e2: from enQ-start to e0.
e3: from e0 to enQ-end.
d1: from deQ-start to deQ-end.
d2: from deQ-start to d0.
d3: from d0 to deQ-end.

[Explanation: Let α be the number of threads in `enQ` or `deQ` but not at `gateE.P()` or `gateD.P()`. Then following holds initially and is preserved by each atomic step:

- (α = 1 and mutex + gateE + gateD = 0) or (α = 0 and mutex + gateE + gateD = 1)

Hence α is either 0 or 1. So at most one of e1, e2, e3, d1, d2, d3 can be executing at any time. End of Explanation]

No overflow or underflow

The following holds initially and is preserved by each atomic step:

1. mutex + gateE + gateD = 1
2. number of threads at `gateE.P()` = nE
3. number of threads at `gateD.P()` = nD
4. (nE = gateE = 0) or
   (nE > 0 and gateE = 0 and buff.size = N) or
   (nE > 0 and gateE = 1 and buff.size = N - 1)
5. (nD = gateD = 0) or
   (nD > 0 and gateD = 0 and buff.size = 0) or
   (nD > 0 and gateD = 1 and buff.size = 1)

Step e1 calls `buff.append` only when `buff.size < N` holds.
Predicate 4 above ensures that e2 calls `buff.append` only when `buff.size < N` holds.
Hence no overflow.

Step d1 calls `buff.remove` only when `buff.size > 1` holds.
Predicate 5 above ensures that d2 calls `buff.remove` only when `buff.size > 1` holds.
Hence no underflow.

10.2 Solution BB2

```
Shared variables:
   Semaphore(1) mutex
   Semaphore(N) spaces  // enQ thread waits here if buff full
   Semaphore(0) items   // deQ thread waits here if buff empty
```

```
enQ(x):
   spaces.P()
   mutex.P()
   buff.append(x)

   mutex.V()
   items.V()
```

```
deQ():
   items.P()
   mutex.P()
   x ← buff.remove()

   mutex.V()
   spaces.V()
   return x
```

11 Awaits

Awaits are powerful synchronization constructs that are very convenient for programmers. Awaits are not provided by programming languages (because they are too powerful). However a program using awaits can be methodically transformed to one using semaphores (or locks and conditions).

An **await** construct has the form `await(B)S`, where `B` is a boolean expression (without side effects) and `S` is a non-blocking statement. A thread at `await(B)S` executes `S` only if `B` holds, all in one atomic step. The thread waits if `B` does not hold. `B` is referred to as a "guard" and `S` as the "action" of the await.

Regarding progress, an await can be "weak" or "strong" (just as in the case of semaphores):

- A thread at a **weak** await eventually gets past if the guard holds continuously.
- A thread at a **strong** await eventually gets past if the guard holds repeatedly or continuously.

The construct `await(true)S` is just a convenient way of specifying that `S` is atomically executed.

11.1 Implementing await-based programs using semaphores

An **await-based program** is a program such that awaits are its only synchronization constructs and there is no conflicting code outside its awaits.

An await-based progam X can be transformed to an equivalent semaphore-based program Y. First introduce the following variables in Y:

- A "gate" semaphore for every *distinct* await guard in X. A thread waits on this if it would wait on a corresponding await in X.
- A "gate" counter for every gate semaphore, indicating the number of threads waiting on the semaphore.
- A "mutex" semaphore to achieve atomicity of each await's implementation in Y.

Next, for each `await(B)S` in X, do the following in Y:

1. Do P on the mutex semaphore.
2. If `B` holds then skip 3 and go to 4.
3. Increment `B`'s gate counter, do V on the mutex semaphore, do P on `B`'s gate semaphore. Upon being awakened, decrement `B`'s gate counter and go to 4.
4. do `S`
5. Look for a guard (which can be `B` or any other) in X such that the guard holds and its gate counter is non-zero. If there is such a guard, then do V on the guard's gate semaphore and exit. If there is no such a guard, do V on the mutex semaphore and exit.

12 Readers-writers problem

Given functions `read()` and `write()`, to obtain functions `concRead()` and `concWrite()` that can be called simultaneously by multiple threads such that:

1. `concRead()` calls `read()` exactly once.
2. `concWrite()` calls `write()` exactly once.
3. The following holds at any time (exactly one holds at any time):
   - (no thread in `read()` and no thread in `write()`), or
   - (no thread in `read()` and 1 thread in `write()`), or
   - (1 or more threads in `read()` and no thread in `write()`).
4. If every `read` call returns and every `write` call returns then:
   a. every call to `concRead` eventually enters `read`, and
   b. every call to `concWrite` eventually enters `write`.
5. while a read is ongoing, every call to `concRead` eventually enters `read`.
   // informally: multiple simultaneous reads should be allowed.

Note that every evolution is a sequence of read intervals and write intervals separated by idle intervals (during which no read or write is ongoing). A write interval has exactly one write. A read interval has one or more reads; it starts when the first read of the interval starts, and ends when the last read of the interval ends.

We now give several solutions, named RW1, RW2, RW3 and RW4. RW1 uses awaits and does not satisfy (progress) requirement 4. RW2 implements RW1 using semaphores; it has the same deficiency. RW3 is another semaphore-based solution that does not satisfy requirement 4; it is the solution usually shown in textbooks. RW4 is a refinement of RW1; it satisfies all the requirements.

12.1 Solution RW1

```
Shared variables:
   int ongR ← 0;    // number of ongoing reads
   int ongW ← 0;    // number of ongoing writes
```

```
concRead():
r1: await (ongW = 0)
       ongR ++
    read()
r2: await (true)
       ongR --
```

```
concWrite():
w1: await (ongW = ongR = 0)
       ongW ++
    write()
w2: await (true)
       ongW --
```

If the awaits are weak, RW1 satisfies requirement 5 but not 4. (A thread can wait forever at `r1` or `w1` while other threads complete reads and writes.)

If the awaits are strong, RW1 satisfies 5 and 4a but not 4b. (A thread can wait forever at `w1` while a never-ending stream of readers ensures that there is always one ongoing read.)

12.2 Solution RW2

Solutin RW2 implements the awaits in RW1 using semaphores. We follow the standard procedure (given in section 11.1). In RW1, blocking occurs in only two places: await `r1` and await `w1`. The guards at `r1` and `w1` are not the same. Hence we introduce two "gate" semaphores (for threads to wait on), two gate counters (tracking how many threads are waiting on each gate), and a "mutex" semaphore. Here is the resulting program.

```
Shared variables:
   int ongR ← 0;         // number of ongoing reads
   int ongW ← 0;         // number of ongoing writes
   Semaphore(0) gateR;  // readers wait here
   Semaphore(0) gateW;  // writers wait here
   int atgR ← 0;         // number of threads waiting on gateR
   int atgW ← 0;         // number of threads waiting on gateW
   Semaphore(1) mutex;  // for atomicity of each await
```

```
concRead():
   mutex.P()           // await r1 start
   if (not ongW = 0)
      atgR ++
      mutex.V()
      gateR.P()
      atgR --
   ongR ++
   if (atgR > 0)
      gateR.V()
   else
      mutex.V()         // await r1 end

   read()

   mutex.P()           // await r2 start
   ongR --
   if (atgW > 0 and ongR = 0)
      gateW.V()
   else
      mutex.V()         // await r2 end
```

```
concWrite():
   mutex.P()           // await w1 start
   if (not ongW = ongR = 0)
      atgW ++
      mutex.V()
      gateW.P()
      atgW --
   ongW ++
1: mutex.V()            // await w1 end

   write()

2: mutex.P()           // await w2 start
   ongW --
3: if (atgW > 0)
      gateW.V()
4: else if (atgR > 0)
      gateR.V()
   else
      mutex.V()         // await w2 end
```

If `mutex` is weak, RW2 satisfies requirement 5 but not 4.

If `mutex` is strong, RW2 satisfies requirement 5 but not 4. It will satisfy 4a if the updates in lines 3 and 5 (in `concWrite`) are interchanged.

Lines 1 and 2 (in `concWrite`) can be deleted (because no other process can execute them while a write is ongoing.

12.3 Solution RW3

Here is the solution that one usually finds in textbooks. It satisfies all requirements except 4b.

A semaphore wrt protects every read interval and write interval; i.e., P is done at the start of the interval and V is done at the end of the interval. (In case of a read interval with more than one read, the P and V are done in different invocations of concRead.)

To detect the start and end of a read interval, a counter nr keeps track of the number of threads in concRead.

A semaphore mutex protects the counter.

```
Shared variables:
   int nr ← 0;             // number of threads in concRead
   Semaphore(1) wrt;       // protects read and write
   Semaphore(0) mutex;     // number of ongoing writes
```

```
concRead():
   mutex.P()
   nr ++
   if (nr = 1)
      wrt.P()
   mutex.V()

   read()

   mutex.P()
   nr --
   if (nr = 0)
      wrt.V()
   mutex.V()
```

```
concWrite():
   wrt.P()

   write()

   wrt.V()
```

Note that if reader threads are blocked (because of any ongoing write), the first reader thread to be blocked is waiting on wrt and all other reader threads are blocked on mutex.

Very slick. But not easily modified to satisfy requirement 4b.

12.4 Solution RW4

A natural way to make solution RW1 satisfy requirement 4b is to impose a limit, say `N`, on the number of consecutive reads while a writer is waiting. To detect this condition it suffices to maintain the following variables:

- `consecR`: number of consecutive reads in the current read interval; 0 if no ongoing read interval.
- `waitngW`: number of waiting writers.

For `consecR` to maintain the above stated meaning, it is incremented when a read starts and zeroed when a write starts. For `waitngW` to maintain the above stated meaning, it is incremented when a thread enters `concWrite` and decremented when the thread starts to write.

Here is the resulting solution, expressed using awaits.

```
Shared variables:
   int ongR ← 0;     // number of ongoing reads
   int ongW ← 0;     // number of ongoing writes
   int consecR ← 0;  // number of consecutive reads currently
   int waitngW ← 0;  // number of waiting writers
```

```
concRead():
   await (ongW = 0 and
          (consecR < N or waitngW = 0)
         )
      ongR ++
      consecR ++
   read()
   await (true)
      ongR --
```

```
concWrite():
   await (true)
      waitngW ++
   await (ongW = ongR = 0)
      ongW ++
      waitngW --
   write()
   await (true)
      ongW --
```

12.5 Solution RW5: semaphore-based implementation of RW4

Left as an exercise.