VU Advanced Multiprocessor Programming
SS 2013
Exercises Batch 1

Jakob Gruber, 0203440

January 27, 2019

Contents

1 Specifications

Select any 12 of $\{6, 7, 9, 10, 11, 12, 13, 14, 15, 34, 35, 36, 38, 39, 40, 41, 42, 43\}$ from *Maurice Herlihy, Nir Shavit: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008. Revised 1st Edition, 2012.*

2 Solutions

2.1 Exercise 6

Suppose a computer program has a method M that cannot be parallelized, and that this method accounts for 40% of the program's execution time. What is the limit for the overall speedup that can be achieved by running the program on a p-processor multiprocessor machine?

Amdahl's law states the following: if a program A consists of a perfectly parallelizable fraction $r, r \in [0, 1]$, and a sequential fraction $1 - r$, the maximum achievable speedup for a problem size of n and p processes is

$$Speedup(p, n) = \frac{T_{seq}(n)}{T_{par}(p, n)} = \frac{T_{seq}(n)}{(1 - r) \cdot T_{seq}(n) + r \cdot \frac{T_{seq}(n)}{p}}$$

Plugging in $r = 0.6$, we get:

$$Speedup(p, n) = \frac{T_{seq}(n)}{0.4 \cdot T_{seq}(n) + 0.6 \cdot \frac{T_{seq}(n)}{p}} = \frac{1}{0.4 + \frac{0.6}{p}}$$

Suppose the method M accounts for 30% of the program's computation time. What should be the speedup of M so that the overall execution time improves by a factor of 2?

Assume that M still refers to the sequential fraction of A, the time required by the parallelizable section does not change and achieves perfect speedup. Let $s = (1-r)$ equal the fraction of $T_{seq}(n)$ required by M and s', r' be the sequential and parallel fraction such that speedup is doubled:

$$\begin{aligned}
Speedup'(p, n) &= \frac{1}{s' + \frac{1-s'}{p}} = \frac{p}{ps' + 1 - s'} = \frac{p}{s'(p-1) + 1} && (1)\\
&= 2 \cdot \frac{p}{s(p-1) + 1} = 2 \cdot Speedup(p, n) && (2)\\
s(p-1) + 1 &= 2s'(p-1) + 2 && (3)\\
s' &= \frac{s(p-1) - 1}{2(p-1)} && (4)
\end{aligned}$$

To transform the relative fraction s into an absolute value S, we set $s = \frac{S}{S+R}$ and $s' = \frac{S'}{S'+R}$. Note that the same R is referenced in both equations since the parallelizable section does not change.

$$\frac{S'}{S'+R} = \frac{\frac{S}{S+R}(p-1)-1}{2(p-1)} \tag{5}$$

$$S' = \frac{S(p-1)-1}{S(p-1)+2R(p-1)+1} \tag{6}$$

The desired speedup of the sequential fraction method M is therefore:

$$S(p) = \frac{S}{S'} = S \cdot \frac{S(p-1)+2R(p-1)+1}{S(p-1)-1} \tag{7}$$

Since this assignment specifies $s = \frac{3}{10}$, we set $R = \frac{7}{3}S$:

$$S(p) = S \cdot \frac{\frac{17}{3}S(p-1)+1}{S(p-1)-1} \tag{8}$$

For a random assignment $p = 7, S = 1, S' = \frac{1}{6}$ results in a the desired overall speedup of 2 (the speedup of M is obviously 6).

Suppose the method M can be sped up three-fold. What fraction of the overall execution time must M account for in order to double the overall speedup of the program?

Under the same assumptions as for the previous point, we can reuse the formula for $S(p)$.

$$S(p) = \frac{S}{S'} = S \cdot \frac{S(p-1)+2R(p-1)+1}{S(p-1)-1} = 3 \tag{9}$$

$$\tag{10}$$

Which gives us:

$$R = \frac{3S(p-1)-3-S^2(p-1)-S}{2S(p-1)} \tag{11}$$

With $s = \frac{S}{S+R}$, we end up with:

$$s = \frac{S}{S+R} = \frac{2S^2(p-1)}{(5S-S^2)(p-1)-3-S} \tag{12}$$

Again, for $p = 7, S = 1$, the initial fraction s evaluates to $\frac{3}{5}$.

Note: I initially solved this assignment by looking only at the relative sequential fractions s, s', which made the involved formulas much simpler but did not take into account that the parallel portion of the program did not change.

```
class Flaky implements Lock {
    private int turn;
    private boolean busy = false;
    public void lock() {
        int me = ThreadID.get();
        do {
            do {
                turn = me;
            } while (busy);
            busy = true;
        } while (turn != me);
    }
    public void unlock() {
        busy = false;
    }
}
```

Figure 1: The Flaky lock used in Exercise 11.

2.2 Exercise 11

Programmers at the Flaky Computer Corporation designed the pro- tocol shown in Figure 1 to achieve n-thread mutual exclusion. For each question, either sketch a proof, or display an execution where it fails.

Does this protocol satisfy mutual exclusion?

Mutual Exclusion Critical sections of different threads do not overlap. For threads A and B, and integers j and k, either $CS_A^j \rightarrow CS_B^k$ or $CS_B^k \rightarrow CS_A^j$.

Suppose not. Then there exist integers j, k such that $CS_A^j \nrightarrow CS_B^k$ and $CS_B^k \nrightarrow CS_A^j$. Inspecting the code, we see that

$$w_A(turn = A) \rightarrow r_A(busy == false) \rightarrow w_A(busy = true) \rightarrow r_A(turn == A)$$
$$w_B(turn = B) \rightarrow r_B(busy == false) \rightarrow w_B(busy = true) \rightarrow r_B(turn == B)$$
$$r_A(turn == A) \rightarrow w_B(turn = B)$$

If we only look at the last loop iteration, in which A successfully enters the critical section, then the last equation must hold. However, this implies $w_A(busy = true) \rightarrow r_A(turn == A) \rightarrow w_B(turn = B) \rightarrow r_B(busy == false)$.

Since busy is never reset to false, this is a contradiction. Therefore the protocol must satisfy mutual exclusion.

Is this protocol starvation-free? Is this protocol deadlock-free?

The Flaky lock is neither starvation-free nor deadlock free. The definitions are as follows:

Freedom from Deadlock: If some thread attempts to acquire the lock, then some thread will succeed in acquiring the lock. If thread A calls lock()

```java
class Filter implements Lock {
    int[] level;
    int[] victim;
    public Filter(int n) {
        level = new int[n];
        victim = new int[n]; // use 1..n-1
        for (int i = 0; i < n; i++) {
            level[i] = 0;
        }
    }
    public void lock() {
        int me = ThreadID.get();
        for (int i = 1; i < n; i++) {
            level[me] = i;
            victim[i] = me;
            while ((exists k != me) such that (level[k] >= i && victim[i] == me)) {};
        }
    }
    public void unlock() {
        int me = ThreadID.get();
        level[me] = 0;
    }
}
```

Figure 2: The Filter lock used in Exercise 12.

but never acquires the lock, then other threads must be completing an infinite number of critical sections.

Freedom from Starvation: Every thread that attempts to acquire the lock even- tually succeeds. Every call to `lock()` eventually returns. This property is some- times called lockout freedom.

Suppose two threads A and B call `lock()` and run in lock-step until they both reach line 10. Without loss of generality, we can assume that B went last, and `turn` is therefore set to B. If A now continues, it reads `turn != me` and therefore enters the loop at line 8. If we let B reach line 11 at this moment, it also reads `turn != me`.

We have ended up with both threads stuck in an endless loop, and neither thread ever aquiring the lock. All other threads are also blocked since `busy` is set to true.

2.3 Exercise 12

Show that the Filter lock allows some threads to overtake others an arbitrary number of times.

It is not possible to overtake a thread within a loop iteration since that would imply changing `victim`, which immediately releases the waiting thread

```
 1 class FastPath implements Lock {
 2     private static ThreadLocal<Integer> myIndex;
 3     private Lock lock;
 4     private int x, y = -1;
 5     public void lock() {
 6         int i = myIndex.get();
 7         x = i;
 8         while (y != -1) {}
 9         y = i;
10         if (x != i)
11             lock.lock();
12     }
13     public void unlock() {
14         y = -1;
15         lock.unlock();
16     }
17 }
```

Figure 3: The FastPath lock used in Exercise 15.

from line 16.

However, between levels it is very possible for other threads to skip ahead. Let's construct a situation in which thread A is overtaken by other threads. Suppose thread A has just successfully entered level 1 (it has completed the first `for` iteration and is about to enter the next iteration with `i = 2`). If A is now paused for some reason (for example scheduling), every other thread which has successfully completed level 1 may overtake A.

2.4 Exercise 15

In practice, almost all lock acquisitions are uncontended, so the most practical measure of a lock's performance is the number of steps needed for a thread to acquire a lock when no other thread is concurrently trying to acquire the lock. Scientists at Cantaloupe-Melon University have devised the following "wrapper" for an arbitrary lock, shown in Figure 3. They claim that if the base Lock class provides mutual exclusion and is starvation-free, so does the FastPath lock, but it can be acquired in a constant number of steps in the absence of contention. Sketch an argument why they are right, or give a counterexample.

We will provide a counterexample, in which both threads A and B are able to enter the critical section.

Suppose both A and B run until line 8. Both are able to pass line 8, because neither have set `y = i` yet. Without loss of generality, assume $w_A(x = A) \rightarrow w_B(x = B) \rightarrow r_B(x == B)$. B is therefore able to enter the critical section without consulting the inner lock.

Now we will allow A to continue. Since $x == B$, `lock.lock()` is called. Since the inner lock has not been locked by B, A is also able to enter the critical section.

```java
public class SafeBooleanMRSWRegister implements
   Register<Boolean> {
    boolean[] s_table; // array of safe SRSW registers
    public SafeBooleanMRSWRegister(int capacity) {
        s_table = new boolean[capacity];
    }
    public Boolean read() {
        return s_table[ThreadID.get()];
    }
    public void write(Boolean x) {
        for (int i = 0; i < s_table.length; i++)
            s_table[i] = x;
    }
}
```

Figure 4: The safe Boolean MRSW construction used in Exercise 34, 38 and 43.

With two threads in the critical section, the property of mutual exclusion is contradicted.

2.5 Exercise 34

Consider the safe Boolean MRSW construction shown in Figure 4. True or false: if we replace the safe Boolean SRSW register array with an array of safe M-valued SRSW registers, then the construction yields a safe M-valued MRSW register. Justify your answer.

A single-writer, multi-reader register implementation is safe if:

- A `read()` call that does not overlap a `write()` call returns the value written by the most recent `write()` call.
- Otherwise, if a `read()` call overlaps a `write()` call, then the `read()` call may return any value within the register's allowed range of values (for example, 0 to $M - 1$ for an M-valued register).

Yes, the construction yields a safe M-valued MRSW register. If `read()` does not overlap `write()`, then all registers in the array have been written and will return the most recent value (remember, these are safe SRSW registers).

If `read()` overlaps `write()`, all we need to show is that any value within the domain is returned. Since that is already guaranteed by the properties of the safe SRSW registers in the array, we do not even need to go into the different cases.

2.6 Exercise 36

Consider the atomic MRSW construction shown in Figure 5. True or false: if we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register. Justify your answer.

```java
public class AtomicMRSWRegister<T> implements Register<T> {
    ThreadLocal<Long> lastStamp;
    private StampedValue<T>[][] a_table; // each entry is SRSW atomic
    public AtomicMRSWRegister(T init, int readers) {
        lastStamp = new ThreadLocal<Long>() {
            protected Long initialValue() { return 0; };
        };
        a_table = (StampedValue<T>[][]) new StampedValue[readers][readers];
        StampedValue<T> value = new StampedValue<T>(init);
        for (int i = 0; i < readers; i++) {
            for (int j = 0; j < readers; j++) {
                a_table[i][j] = value;
            }
        }
    }
    public T read() {
        int me = ThreadID.get();
        StampedValue<T> value = a_table[me][me];
        for (int i = 0; i < a_table.length; i++) {
            value = StampedValue.max(value, a_table[i][me]);
        }
        for (int i = 0; i < a_table.length; i++) {
            a_table[me][i] = value;
        }
        return value;
    }
    public void write(T v) {
        long stamp = lastStamp.get() + 1;
        lastStamp.set(stamp);
        StampedValue<T> value = new StampedValue<T>(stamp, v);
        for (int i = 0; i < a_table.length; i++) {
            a_table[i][i] = value;
        }
    }
}
```

Figure 5: The atomic MRSW construction used in Exercise 36.

Atomic registers satisfy the following conditions, regular registers only the first two:

$$\text{it is never the case that } R^i \rightarrow W^i \tag{13}$$

$$\text{it is never the case that for some } j : W^i \rightarrow W^j \rightarrow R^i \tag{14}$$

$$\text{if } R^i \rightarrow R^j \text{ then } i \leq j \tag{15}$$

The construction which we arrive at by substituting the array of atomic registers `a_table` with an array of regular registers `r_table` yields an atomic MRSW register.

Regular and atomic registers differ only in their behavior for overlapping reads and writes. Since the second property concerns non-overlapping writes, and the original construction from the figure is an atomic MRSW register, the altered construction also satisfies the second property.

It remains to show the third condition, which specifies the behavior of non-overlapping reads. To violate this condition, a read must return an earlier value than a preceding, non-overlapping read. Since every read writes its value into an entire row, this is impossible. The later read will pick up the latest value read by any earlier read by fetching the maximum `StampedValue` over a column and so this condition is also satisfied.

The linearization point is either when the write completes, or when a read (which has already seen the latest value) finishes writing its row.

2.7 Exercise 38

Consider the safe Boolean MRSW construction shown in Figure 4. True or false: if we replace the safe Boolean SRSW register array with an array of regular M-valued SRSW registers, then the construction yields a regular M-valued MRSW register. Justify your answer.

Regular registers satisfy the following conditions:

$$\text{it is never the case that } R^i \rightarrow W^i \tag{16}$$

$$\text{it is never the case that for some } j : W^i \rightarrow W^j \rightarrow R^i \tag{17}$$

$$\text{if a read overlaps a write, it returns either the old or the new value} \tag{18}$$

We have already shown that replacing the array of safe boolean registers with safe M-valued registers results in a safe M-valued MRSW register. `read()` never returns a value from the future, so it remains to examine the second and third conditions.

If `read()` and `write()` calls do not overlap, `read()` returns the most recent value. If they do overlap, we can distinguish between three cases:

1. If `r_table[ThreadID.get()]` has not been written yet, the old value is returned.
2. If `r_table[ThreadID.get()]` has already been written, the new value is returned.

```java
public class RegBooleanMRSWRegister implements
    Register<Boolean> {
     ThreadLocal<Boolean> last;
     boolean s_value; // safe MRSW register
     RegBooleanMRSWRegister(int capacity) {
         last = new ThreadLocal<Boolean>() {
             protected Boolean initialValue() { return
                 false; };
         };
     }
     public void write(Boolean x) {
         if (x != last.get()) {
             last.set(x);
             s_value = x;
         }
     }
     public Boolean read() {
         return s_value;
     }
}
```

Figure 6: The regular boolean MRSW construction used in Exercise 39.

3. If `r_table[ThreadID.get()]` is being written as it is being read, it can return either the old value or the new value (as `r_table` is composed of regular registers).

Both conditions are satisfied, therefore the construction yields a regular M-valued MRSW register.

2.8 Exercise 39

Consider the regular Boolean MRSW construction shown in Figure 6. True or false: if we replace the safe Boolean MRSW register with a safe M-valued MRSW register, then the construction yields a regular M-valued MRSW register. Justify your answer.

If this construction were a regular M-valued MRSW register, it would need to return either the old or the new value in case of a `read()` overlapping a `write()`. In the original boolean register, this is the case: if $new \neq old$, the backing safe register can read any value of the domain; since the domain only consists of two values, it must equal either new or old, and the property is satisfied.

This cannot be extended to M-valued registers. In the case of an overlapping write, the backing safe M-valued register can return any element $\in \mathbb{D}$. Since this value is not guaranteed to equal either the old or the new value, this construction is not regular.

```java
 1  class Peterson implements Lock {
 2      // thread-local index, 0 or 1
 3      private volatile boolean[] flag = new boolean[2];
 4      private volatile int victim;
 5      public void lock() {
 6          int i = ThreadID.get();
 7          int j = 1 - i;
 8          flag[i] = true;
 9          victim = i;
10          while (flag[j] && victim == i) {}; // wait
11      }
12      public void unlock() {
13          int i = ThreadID.get();
14          flag[i] = false;
15      }
16  }
```

Figure 7: The Peterson lock used in Exercise 40.

2.9 Exercise 40

Does Peterson's two-thread mutual exclusion algorithm in Figure 7 work if we replace shared atomic registers with regular registers?

The difference between regular and atomic registers is that atomic registers can't switch between old and new values when read during a `write()`.

In this algorithm, registers are only read in line 10. In the case of a thread running solo, no reads and writes overlap and the algorithm is correct. If threads A and B are trying to acquire the lock at the same time, overlaps can occur only in lines 8 and 9.

If the overlap occurs with B in line 8 (A is spinning in line 10), we have two cases:

1. A reads the old value of flag[i] and enters the critical section.
2. A reads the new value of flag[i] continues spinning.

This leads to two further cases with B in line 9:

1. A reads the old value of victim and spins until B finishes writing.
2. A reads the new value of victim and enters the critical section.

In all cases, B will spin at line 10 because the value of `flag[i]` is never changed and the B has completed the write to victim before reaching line 10.

The algorithm is therefore correct even if regular registers are used.

2.10 Exercise 41

Consider the following implementation of a Register in a distributed, message-passing system. There are n processors $P_0, ..., P_{n-1}$ arranged in a ring, where

P_i can send messages only to $P_{i+1 \bmod n}$. Messages are delivered in FIFO order along each link. Each processor keeps a copy of the shared register. To read a register, the processor reads the copy in its local memory. A processor P_i starts a `write()` call of value v to register x, by sending the message "P_i : write v to x" to $P_{i+1 \bmod n}$. If P_i receives a message "P_j : write v to x," for $i \neq j$, then it writes v to its local copy of x, and forwards the message to $P_{i+1 \bmod n}$. If P_i receives a message "P_i : write v to x," then it writes v to its local copy of x, and discards the message. The `write()` call is now complete.

Give a short justification or counterexample. If write() calls never overlap, is this register implementation regular? Is it atomic?

At any time during the write, reads can only return either the old or the new value; therefore this construction is a regular register.

However, there is no single commit point, making this register non-atomic. Consider a write started by P_0 at $t = 0$ in a ring of three processors. A read at processor $P_1, t = 1$ returns the new value, while a read at processr $P_0, t = 2$ returns the old value.

If multiple processors call write(), is this register implementation atomic?

No. Suppose a write is started by processor P_2 at $t = 0$ with $v = 2$. At $t = 1$, P_0 starts a write with $v = 0$. The ring consists of the processors $P_i, i \in 0, 1, 2, 3$.

Once both writes have completed P_1, P_2 will read $v = 2$, while P_3, P_0 see $v = 0$. This already violates the requirement that reads without overlapping writes return the value of the most recent write.

2.11 Exercise 42

You learn that your competitor, the Acme Atomic Register Company, has developed a way to use Boolean (single-bit) atomic registers to construct an efficient write-once single-reader single-writer atomic register. Through your spies, you acquire the code fragment shown in Figure 8, which is unfortunately missing the code for `read()`. Your job is to devise a `read()` method that works for this class, and to justify (informally) why it works. (Remember that the register is write-once, meaning that your read will overlap at most one write.)

This answer assumes that N is actually the number of bits of the value stored in `AcmeRegister`, not the number of threads as stated in the book.

To satisfy the conditions of atomic registers, `read()` must return either the old value or the new value (if a `read()` overlaps the single `write()` call), and write must have a linearization point.

The solution is given in Figure 9.

Informally, `write()` constructs a local copy of `b`, reading from back to front. It then returns a reconstruction of the integer from a specific subsection of the local copy, depending on whether the first two sections are equal or not.

If a read does not overlap a write, this register returns the latest value.

Since this register is write-once, there may only be at most one $i \in [1, 3N-1]$ such that $c_i = b_{i_{old}}$ and $c_{i-1} = b_{i-1_{new}}$. If $n < 2N$, the old value $c[2N..3N-1]$ is returned. As soon as $n \geq 2N$, only the new value can be returned. The linearization point is therefore located at line 13 in Figure 8 (or more specifically, when `b[2N-1]` is written).

```java
class AcmeRegister implements Register {
    // N is the total number of threads
    // Atomic multi-reader single-writer registers
    private BoolRegister[] b = new BoolMRSWRegister[3
        * N];
    public void write(int x) {
        boolean[] v = intToBooleanArray(x);
        // copy v[i] to b[i] in ascending order of i
        for (int i = 0; i < N; i++)
            b[i].write(v[i]);
        // copy v[i] to b[N+i] in ascending order of i
        for (int i = 0; i < N; i++)
            b[N+i].write(v[i]);
        // copy v[i] to b[2N+i] in ascending order of
            i
        for (int i = 0; i < N; i++)
            b[(2*N)+i].write(v[i]);
    }
    public int read() {
        // missing code
    }
}
```

Figure 8: Partial acme register implementation used in Exercise 42.

```java
class AcmeRegister implements Register{
    /* ... */
    public int read() {
        BoolRegister[] c = new BoolMRSWRegister[3 * N
            ];
        for (int i = 3 * N - 1; i >= 0; i--) {
            c[i] = b[i];
        }
        if (c[0..N-1] is equal to c[N..2N-1]) {
            return booleanArrayToInt(c[0..N-1]);
        } else {
            return booleanArrayToInt(c[2N..3N-1]);
        }
    }
}
```

Figure 9: Solution to Exercise 42.

2.12 Exercise 43

Prove that the safe Boolean MRSW register construction from safe Boolean SRSW registers illustrated in Figure 4 is a correct implementation of a regular MRSW register if the component registers are regular SRSW registers.

See the solution to exercise 38.