Parallel Programming - Exercise Solutions

August 13, 2010

Important note: You might find the following exercises and control questions useful, while learning for the exam. Please note that this is inofficial material, which is not meant as a guideline for problems you might encounter during the exam. I do not provide any guarantees regarding completeness or correctness of the solutions.

Not so important note: I have not tested any of the code written in this document. Most snippets probably wont even compile because of small typos or syntax errors.

Feel free to send me a mail in case you find an incorrect or unclear solution.

Contents

1 Exceptions

1.1 Explain the difference between checked and unchecked exceptions.

Checked exceptions must always have a matching catch block. If a method does not want to handle a checked exception it must announce that it could throw the exception ("void m() throws Exception"). In that case the caller must handle the checked exception. Unchecked exceptions may occur anywhere an may be unhandled. Most of the time unchecked exceptions are thrown as side effects of instructions other than throw.

1.2 What is the output of the run() method in the following examples?

1.
```
A
finally
IOException
finally2
end
```

2.
```
IO
done
```

1.3 Will the following classes compile? If not: Why not? (Ignore errors related to unreachable code. Focus on Exception related errors)

1. Does not compile. A throws announcement is needed even though the condition can never be true.
2. $\surd$
3. Does not compile. A announces that it can throw any exception. B must handle all exceptions announced by A. Not just those actually thrown.
4. Does not compile.
5. Does not compile. A announces that it can throw any exception. B must handle all exceptions announced by A. Not just those actually thrown.
6. $\surd$
7. Does not compile. InterruptedException is unhandled.

2 Java Threads

2.1 Which of the following snippets do what their comment says?

1. No. run() just executes the run method in the current thread. No new thread is created.

2. No. run() just executes the run method in the current thread. No new thread is created. You cannot pass parameters to a thread this way. Parameters have to be passed via the constructor.

3. No. After each thread has been started join is immediately called. Join blocks until that thread has finished. Therefore the ten threads do not run in parallel.

4. $\surd$ This actually works. Thread implements Runnable. So a MyThread object is a valid argument to the constructor of another thread. Note that only the outer Thread is started, which then calls run on the MyThread object. (This is very bad coding style!)

5. No. Instead of overwriting (Eiffel: "redefine") the run method, MyThread just defines a new method run(string). MyThread now has two (overloaded) run methods: run() and run(string). Calling start() creates a new thread that will execute the run() method, not the newly defined run(string) method.

2.2 Draw a diagram containing all thread states and their transitions.

- NEW $\xrightarrow{start}$ RUNNABLE
- RUNNABLE:
  - $\xrightarrow{exit}$ TERMINATED
  - $\xrightarrow[join]{wait}$ WAITING
  - $\xrightarrow{synchronized\, contented}$ BLOCKED
- WAITING:
  - $\xrightarrow[InterruptedException]{join()\, returns}$ RUNNABLE
  - $\xrightarrow{wait()\, returns}$ BLOCKED (notify received)
- BLOCKED $\xrightarrow{synchronized\, free}$ RUNNABLE
- TERMINATED

3 Synchronization

3.1 Which of the following classes is equivalent to the first one?

3.1.1 Set 1

1. No
2. $\surd$
3. No
4. No
5. No
6. No (get() and addAndGet() are atomic. But both together are not)
7. $\surd$
8. $\surd$
9. $\surd$ If newValue is less then zero we throw an exception. If compareAndSet fails we know that someone modified the counter between the call to get() and compareAndSet() $\Rightarrow$ retry. This is an example of optimistic concurrency. Instead of acquiring a lock we assume that no one else is modifying the counter. If that turns out to be wrong, then we have to redo our work.

   General pattern with compareAndSet()

   - Copy state (atomic)
   - Update copy
   - If state unchanged: Copy back using compareAndSet. (atomic)

3.1.2 Set 2

1. No. A thread calling getValue() might not read the last value, which is guaranteed in the fully synchronized version.
2. $\surd$
3. No. increment() is not atomic!

The issue with 1 is not simply that the value will be read “sometime later” (which could maybe be tolerated), but more subtile:

```
//Assume all counters are set to 0
//Thread A
c1.increment();
c2.increment();

//Thread B
int a = c1.getValue();
int b = c2.getValue();

//In the synchronized example a==b or a==b+1.
//In Version 1 this is not guaranteed due to visibility issues
//In Version 2 it is, because all reads and writes are volatile.
```

3.2 Static methods

1. Method A locks on ’this’. (one lock per instance of the class)
2. Method B is static and locks on the ’Test.class’ a static object representing the Test class. (one global lock!)
3. Mutual exclusion is not achieved. Both methods lock on different objects.

3.3 This Buffer is used to transfer objects between multiple producers and consumers. Does it work? If not why?

The Buffer is incorrect. It locks on ’data’ which changes between multiple invocations. This means that different threads will use different locks. Therefore mutual exclusion is not guaranteed. (Also data could be null.)

3.3.1 The student who wrote the buffer class claims that a slide from the lecture reads “lock data not code” and that therefore his code is correct. How do you respond?

The idea is that your locks should be associated with the data you are modifying, not with the code that modifies the data. get() and set() modify the buffer, not the object stored in data. Therefore the buffer should be locked. Locking code would mean to associate one global lock with set() and get(). Then no operations (even on different Buffer objects) could be done in parallel, which would be bad.

3.4 Correct all errors you find in the following program:

```
class SynchronizedBuffer {
  private boolean full;
  private Object object;

  public synchronized void write(Object obj) {
    while(full) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    object = obj;
    full = true;
    notifyAll();
  }

  public synchronized Object read() {
    while(!full) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    notifyAll();
    full = false;
    return object;
  }
}
```

3.5 Monitors and Semaphores

3.5.1 A student notes that the following code will deadlock because the lock is acquired in "incrementTenTimes()" and again in "increment()", when called from the loop. Is that a problem? Why/why not?

Java's synchronized is reentrant. This means that a lock can be acquired multiple times by one thread without deadlocking.

3.5.2 What happens if you convert the SharedObject class to semaphores using the technique discussed in the lecture? Does it work now? Does it no longer work?

Semaphores are not reentrant. If we acquire a semaphore in incrementTenTimes() and then reacquire it in increment() a deadlock will occur.

3.5.3 How would you solve the problems encountered in the previous two questions (if any)? (Note that both methods belong to the public interface of the class)

Either use x++ directly in incrementTenTimes() or use a private method that is used by both public methods:

```java
class SharedObject {
  int x = 0;
  Semaphore sem = new Semaphore(1);
  public void increment() {
    sem.acquire();
    doIncrement();
    sem.release();
  }

  public void incrementTenTimes() {
    sem.acquire();
    for (int i=0;i<10;i++)
      doIncrement();
    sem.release();
  }

  //require: sem already acquired
  private void doIncrement() {
    x++;
  }
}
```

4 Deadlocks

4.1 What is a deadlock?

A deadlock is a situation wherein two or more competing actions are each waiting for the other to finish, and thus neither ever does. http://en.wikipedia.org/wiki/Deadlock

4.2 Name two techniques/approaches that can be used to deal with deadlocks.

1. Order all locks and acquire them always in the same order.
2. Detect deadlocks (either through analysis (A waits for B, B waits for C, C waits for A) or use a timeout) and recover. (let one thread back off, abort and restart one thread)

4.3 Which of the following programs could deadlock? Give a small argument on why a deadlock can / cannot occur.

1. The following code might deadlock:

```
//Thread A:
a.transfer(b,10);
//Thread B:
b.transfer(a,10);
```

2. $\surd$ Only one thread acquires more than one lock.
3. $\surd$ All lock operations are ordered. Therefore no deadlock can occur.
4. $\surd$
5. A deadlock can occur. (acquire({0,1}) vs acquire({1,0})

5 Data races, visibility, volatile

5.1 What is a data race?

If the output of a program or method depends on the ordering of instructions the program is said to have a race condition (or data race). For example if two threads write to the same variable a "race" occurs. The output depends on who finishes its write first. In some cases races can be part of a normal execution. If a queue contains the values 1 and 2 and two threads call dequeue(), then it is not clear which thread will receive which value. As long es the overall output does not depend on this distinction no harm is done.

Definition from the slides (for memory accesses): Two accesses are involved in a race condition if:

- at least one of them is a write
- they are not ordered

5.2 How can data races be prevented?

synchronized, semaphores...

5.3 What could be the value of x after the following program has finished?

$x \in [2, 150]$

The following execution leads to $x = 2$:
A: read $x_a = 0$
B: increment 49 times: $x = 49$
A: increment one time: $x = 1$
B: read $x_b = 1$ (B is in its last iteration now)
A: increment 99 times: $x = 100$
B: increment one time: $x = 2$

5.4 How does the situation change if x is defined as a volatile variable?

Nothing changes. Volatile reads and writes are already atomic in the previous version. Volatile does not guarantee atomicity for the whole operation (read, increment, write).

5.5 Which of the following operations are guaranteed to be atomic?

(int i, long l, float f, double d, volatile int vi, volatile long vl, Integer ii, Object o,oo)

Type	Atomic	Not atomic
i = 5	[✓]	[ ]
l = 5	[ ]	[✓]
ii = 5	[✓]	[ ]
f = 5	[✓]	[ ]
d = 5	[ ]	[✓]
o = oo*	[✓]	[✓]
vi = 5	[✓]	[ ]
vl = 5	[✓]	[ ]
vi++	[ ]	[✓]
vl++	[ ]	[✓]

*Depends on your point of view. The assignment is atomic. Meaning that if two threads write at the same time, one of them will win (race condition) and o will contain either the value of thread A or the one from thread B. (Contrast this with an assignment to a long value, where the result can be a mix of two writes.) On the other hand, the statement consists of a read and a write. If another thread executes oo = o concurrently then a possible outcome is that o and oo are swapped. This exposes the non atomicity of the statement.

5.6 Visibility/Ordering

What are the possible outputs of the following programs? Do they always print 5? If not what can go wrong?

1. $\surd$
2. No. A volatile write only guarantees that all previous writes are visible. Not that all later writes are invisible! ⇒ ready == true can be seen before the volatile write. (This is somewhat advanced)
3. No. volatile int[] is a volatile reference to an array of normal integers. It is not possible to define an array of volatile ints. Therefore data[0] = 5 can be executed after ready = true. Which means that thread B can see a wrong value. (There is an AtomicIntegerArray class, but that is a normal object with get and set methods. Not a "real" Java array.)
4. $\surd$ (Here a new array is created, and then assigned to data. The assignment is volatile and therefore ordered (array creation happens first, then the array is filled with data, then the (volatile) assignment to data)
5. No. Object creation does not guarantee visibility. The assignment to mgc can become visible before the assignment to data.

6 Programming

6.1 Implement a parallel method *int sum(int[] data)* that sums all elements of the data array.

```
int sum(int[] data) {
  int sum = 0;
  //omp parallel for reduction(+:sum)
  for(int i=0;i<data.length;i++)
    sum+=data[i];
  return sum;
}
```

6.2 Write a program that increases all values in an array by 20. The update should be done in parallel:

```
int[] data;
//omp parallel for
for(int i=0;i<data.length;i++)
  data[i]+=20;
```

6.3 Implement a counter class with the following interface:

```
//Initial value should be zero.
class Counter {
   int counter = 0;

   //increment stored value
   public synchronized void increment() {
      counter++;
      notifyAll();
   }

   //wait until the counter reaches value
   public synchronized void waitForValue(int value) {
      while ( counter < value ) {
         try {
            wait();
         } catch (InterruptedException e) { }
      }
   }
}
```

6.4 Implement a ThreadPool

Creating threads has some overhead. If a lot of small tasks have to be started this overhead becomes significant. The overhead can be avoided by using a threadpool. A threadpool starts a number of threads at startup and then accepts a tasks in form of Runnable objects (not necessarily Threads) which are assigned to threads in a FIFO manner. No new threads are created.

```
//This code has not been tested :)
class WorkerThread extends Thread {
  private ThreadPool pool;
  public WorkerThread(ThreadPool pool) {
    this.pool = pool;
  }
  public void run() {
    while (true) {
      Runnable task;
      synchronized(pool) {
        while (queue.size() == 0) {
          try {
            pool.wait();
          } catch (Exception e) {}
        }
        task = (Runnable) pool.removeFirst();
      }
      task.run();
    }
  }
}
```

```
class ThreadPool {
  public LinktedList queue = new LinkedList();

  public ThreadPool(int threadCount) {
    for (int i=0; i<threadCount; i++)
      new WorkerThread(this).start();
  }

  public void synchronized addTask(Runnable runnable) {
    queue.add(runnable);
    notify();
  }
}
```

7 Questions

7.1 Fairness

7.1.1 Name 4 fairness models and describe their properties.

Weak fairness: continuous request → eventually granted

Strong fairness: requests infinitely often → eventually granted

Linear waiting: request is granted before any other request got granted twice

FIFO waiting: a request is granted before any later request is granted.

7.1.2 Which of the following constructs are fair? Write down any additional assumptions you make.

Type	Not fair	fair
synchronized {	[√]	[ ]
sem.acquire()*	[√]	[√]
notify()	[√]	[ ]
notifyAll()	[√]	[ ]

* depends (fifo setting in semaphore constructor)

7.2 Performance

7.2.1 Whats the difference between cpu time and elapsed time ("wall-clock time")?

CPU time: The time spend running the time on the CPU.

Wallclock time: Total time that the program takes to execute. Includes waiting for IO and other programs that share the same cpu.

7.2.2 What is the k-best measurement scheme? Why is it used? Are there disadvantages to using the scheme?

Take the best k measurements from a series. With this measurement scheme the influence of external factors is reduced. The program never runs in isolation and other programs may interfere with your measurements. Of course this just gives a best case for your program and hides random effects from your program. In reality you might be more interested in average or worst case performance.

7.2.3 A program is executed on two processors. To everyone's surprise it executed 2.5 times faster than on one processor. Give a possible explanation.

Running on two processors means that the program can use twice as much cache. If data partitioning is used each processor only has to work with half the data. Those effects can additionally improve performance.

7.3 Parallelism and scaling

7.3.1 Whats the difference between task and data parallelism?

7.3.2 How would you parallelize:

1. Matrix multiplication
2. Sorting

7.3.3 What scales better with the number of processors. Task or data parallelism?

Data parallelism. Normally you have a fixed number of tasks. New processors only help as long as $\#tasks > \#processors$. With data parallelism you can exploit more processors. (Assuming you have a lot of data.)

8 Amdahl's law

$S = \frac{1}{1-P+\frac{P}{n}}$

8.1 What can you say about the parallel part of the following programs:

1. A program takes 20 seconds on one processor and 12 seconds on two processors.

   $S = \frac{20}{12} = \frac{1}{1-P+\frac{P}{2}}$

   $\Rightarrow P = \frac{4}{5}$

2. A program takes 30 seconds on one processor, 13 on two and 5 seconds on four processors.

   The speedup cannot be explained with Amdahl's law. Also see question 7.2.3.

8.2 Whats the maximum expected speedup if 60% of your program can run in parallel

1. On a two processor machine?

   $S = \frac{1}{1-0.6+\frac{0.6}{2}} = \frac{10}{7}$

2. On a three processor machine?

   $S = \frac{1}{1-0.6+\frac{0.6}{3}} = \frac{5}{3}$

3. With an infinite number of processors?

   $S = \frac{1}{1-0.6} = 2.5$

8.3 Two thirds of a method M can be parallelized. How many processors are needed to...

1. ...achieve a speedup of 2

   $S = \frac{1}{1-\frac{2}{3}+\frac{2}{3n}} \overset{!}{=} 2$

   $\Rightarrow n = 4$

2. ...achieve a speedup of 4

   $S = \frac{1}{1-\frac{2}{3}+\frac{2}{3n}} \overset{!}{=} 4$

   $\frac{1}{\frac{1}{3}+\frac{2}{3n}} = 4 \Rightarrow 1 = \frac{4}{3} + \frac{8}{3n}$

   $\Rightarrow -\frac{1}{3} = \frac{8}{3n} \Rightarrow n = -8$

   It is impossible to achieve a speedup of 4.

8.4 A clever optimization speeds up a method G by a factor of ten. What is the speedup of the whole program?

This question cannot be answered without knowing how much time is spend in G. If G makes up only 1% of the whole program, then the optimization will not be noticeable. If G took 50% of the time then the speedup will be $S = \frac{1}{0.5+\frac{0.5}{10}} = \frac{20}{11} \approx 1.8$

9 OpenMP

9.1 Parallelize the following programs using OpenMP. Do not modify the programs besides adding "//omp ..." lines.

1.

```
int[][] data;
int result = 0;
int i;
int j;

//omp parallel for private(j) reduction(+:result)
for (i=0; i<data.length; i++) {
  ________________________________________
  for (j=0; j<data[i].length; j++)
      ________________________________________
      result += data[i][j];
}
```

2.

```
int[] data;
int i;
int j;
int a;
________________________________________
for (i=0; i<data.length;i++) {
  ________________________________________
  a = 0;
  //omp parallel for lastprivate(a)
  for (j=0;j<data.length;j++) {
    if (i != j)
      data[j] += data[i];
    a = data[j]*i*j;
  }
  ________________________________________
    data[i] = a;
}
```

3.
```
int[] src;
int[] dst;
int[] res;

int i;
int j;
int k;
_______________________________________
for (i=0; i<src.length;i++)
//omp parallel private(j,k)
{
  //omp for nowait
  for (j=0;j<src.length;j++) {
    _______________________________________
    dst[j] += src[i]+j;
  }

  //omp single
  for(j=k;k<src.length;k++) {
    _______________________________________
    res[k] += res[k-1];
  }
}
```

4.
```
int[][] data;

int i;
int j;
int tmp;
//omp parallel for(j,tmp)
for (i=1; i<data.length; i++) {
  _______________________________________
  for (j=0; j<data[i].length; j++) {
    _______________________________________
    tmp = data[i][0] + data[i][2] + data[i][j];
    _______________________________________
    if (tmp < 0)
    _______________________________________
    {
      _______________________________________
      data [i][0] = tmp;
    }
    _______________________________________
    if (tmp == 0)
    //omp critical
    {
      _______________________________________
      data[0][0]++;
    }
    _______________________________________
  }
}
```

10 Mutual exclusion

10.1 Lamport's bakery algorithm

10.1.1 Explain the asymmetry in the version for two threads. Why is it needed? How is this problem solved in the version that works for N threads?

The asymmetry is used as a tie breaker in case both threads get the same number (line 2 is not atomic!). The N thread version uses the index into the number array (thread id) as a tie breaker.

10.1.2 Is starvation possible with Lamport's Bakery algorithm? Why / Why not?

Starvation is not possible. Each thread acquires a number when entering. The thread now waits until all threads with a lower number have finished their work. If a new thread wants to enter (or reenter) it has to get a new number which will be higher.

10.1.3 Show that the algorithm (for two threads) achieves mutual exclusion.

Invariants:

1. $A2 - A3 \Rightarrow Entering_A$
2. $B2 - B3 \Rightarrow Entering_B$
3. $A3 - A8 \Rightarrow Number_A \neq 0$
4. $B3 - B8 \Rightarrow Number_B \neq 0$
5. $A7 \implies Number_B = 0 \vee Number_B >= Number_A$
6. $B7 \implies Number_A = 0 \vee Number_A > Number_B$
7. $\neg (A7 \wedge B7)$ - Mutual Exclusion

Proof of (1), (2): $Entering_X$ is only modified by thread X. It is set to true at $X1$ and to false at $X3$. Therefore the invariant holds.

Proof of (3), (4): $Number_X$ is only modified by thread X. It is set to a value greater zero at $X2$ and to zero at $X8$. Therefore the invariant holds. ($Number_X$ is always non negative $\Rightarrow Number_X + 1 > 0$)

Proof of (5): Invariant holds initially.

Transitions for thread A

$A5 \rightarrow A7$: invariant has to hold for the transition to take place

$A6 \rightarrow A7$: invariant has to hold for the transition to take place

No other way to reach $A7$. Invariant holds

Transitions for thread B

$B2 \to B3$:

Note that the statement $B2$ is not atomic! We cannot assume that thread A does nothing while we execute $B2$.

Thread A cannot go from $A4 \to A5$ (because $Entering_B$ is true while we are at $B2$ (invariant 2))

Case 1: Assume Thread A is at $A7$ after we execute $B2 \to B3$. Then Thread A must have been at $A5 - A7$ when we started. Therefore Thread A did not modify $Number_A$ and the invariant holds (because $Number_B = Number_A + 1$)

Case 2: Assume Thread A is not at $A7$ after we execute $B2 \to B3$. Then the invariant holds.

$B8 \to B1 : \surd$

No other transitions modifies $Number_B$.

$\Rightarrow$ Invariant holds

Proof of (6): Analogous to (5).

Proof of (7): Assume that (7) is true and derive a contradiction using 3-6.

10.2 Claudio's solution

10.2.1 Is starvation possible with Claudio's solution? Why / Why not?

Yes, starvation is possible.

```
Thread A lock() t=1,b=0
Thread B lock(), two iterations: n_B=2, t=3, b=0
Thread A unlock() n_B=2, t=3, b=1
```

Now thread B loops forever. Since b will always be smaller than t.
A possible solution is call getAndIncrement() only once per call to lock!

10.2.2 Either give an execution in which two threads enter the critical section or show that the algorithm (for two threads) achieves mutual exclusion.

All access are to atomic variables and therefore ordered and immediately visible. Since getAndIncrement in (2) is atomic, each call will return a new value. Two calls will never return the same value (overflows are ignored). b is only changed in (4). Assume two threads (A and B) are in the critical section. Let n_A and n_B be the values they used to leave the loop in (3). Assume A entered first at time t_A and that b entered at time $t_B > t_A$. Then at t_A it must have been the case that $b == n_A$. Also at time t_B the term $b == n_B$ must have been true. Since

A was still in its critical section b cannot have been modified (only modified at (4)) and therefore $n_A == n_B$. But as stated previously $n_A \neq n_B$. The case that B entered before A is symmetric. Therefore the algorithm guarantees mutual exclusion.