Research Group for Parallel Computing
Faculty of Informatics
TU Wien

Advanced Multiprocessor Programming

Summer term 2021
Batch 1

1: Daniel, Schloms, 11701253

Exercise 6

Answer 1

Sequential Part $s = 1 - p = 0.4$
Parallelizable Part $p = 0.6$
Limit $SU = \lim_{n\to\infty} \frac{1}{0.4+\frac{0.6}{n}} = 2.5$

Answer 2

Sequential Part $s = 0.3$, S_n is speedup on n processes
Find k so that if the sequential part runs k times faster, the speedup is doubled.

- $\frac{0.3}{k} + \frac{1-\frac{0.3}{k}}{n} \leq \frac{1}{2} \times (0.3 + \frac{0.7}{n})$
- $\frac{0.3n+k-0.3}{kn} \leq \frac{1}{2} * (0.3 + \frac{0.7}{n})$
- $0.6n + 2k - 0.6 \leq kn * (0.3 + \frac{0.7}{n})$
- $0.6n - 0.6 \leq k * (0.3n + 0.7 - 2)$
- $k \geq \frac{0.6n-0.6}{0.3n-1.3}$
- Since k must be greater than 0, n must be at least 5 ($n \in \mathbb{N} \setminus \{0\}$).

Answer 3

Sequential part s sped up three-fold causes half the original time on n processors, what fraction did the sequential part account for?

- $s + \frac{1-s}{n} = 2 * (\frac{s}{3} + \frac{1-\frac{s}{3}}{n})$
- $s + \frac{1}{n} - \frac{s}{n} = \frac{2}{3}s + \frac{2}{n} - \frac{\frac{2}{3}s}{n}$
- $\frac{1}{3}s = \frac{1}{n} + \frac{\frac{s}{3}}{n} = \frac{1+\frac{s}{3}}{n}$
- $\frac{\frac{1}{3}s}{1+\frac{s}{3}} = \frac{1}{n} \rightarrow n = \frac{1+\frac{s}{3}}{\frac{s}{3}} = \frac{1}{\frac{s}{3}} + 1$
- $\frac{3}{s} = n - 1 \rightarrow \frac{s}{3} = \frac{1}{n-1} \rightarrow s = \frac{3}{n-1}$

Exercise 7

Running application with sequential part s with 2 processors yields speedup S_2. Derive formula for S_n (speedup with n processors) in terms of n and S_2.

- $S_2 = \frac{1}{s+\frac{1-s}{2}}$
- $\frac{1}{S_2} = s + \frac{1-s}{n} = \frac{2s+1-s}{2} = \frac{s+1}{2}$
- $\frac{2}{S_2} = s + 1 \rightarrow s = \frac{2}{S_2} - 1$

Then insert s into Amdahl's law for n processors:
$S_n = \frac{1}{(\frac{2}{S_2}-1)+\frac{1-(\frac{2}{S_2}-1)}{n}}$

Exercise 8

Choice between one uniprocessor with five zillion IPS or ten-processor multiprocessor with one zillion IPS each? Decide with Amdahl's.

- Basically check where multiprocessor gives a speedup ≥ 5 since then it matches and surpasses the 5 zillion IPS processor. s is the sequential part.
- $\frac{1}{s+\frac{1-s}{10}} \geq 5$
- $s + \frac{1-s}{10} \leq \frac{1}{5} \rightarrow \frac{10s+1-s}{10} \leq \frac{1}{5}$
- $9s + 1 \leq 2 \rightarrow 9s \leq 1 \rightarrow s \leq \frac{1}{9}$
- This means that if sequential part is less than $\frac{1}{9}$ use the multiprocessor, if its more use the uniprocessor.

Exercise 9

The Peterson algorithm provides 0-bounded waiting if the doorway is defined as the two lines before the while loop.

- The Peterson lock is used for 2 threads only.
- As soon as both threads wait for entry to the critical section, no one thread can lap the other, Peterson is fair/first-come-first-served, which means the waiting is $0-bounded$ in this context.
- Assuming that after both threads have waited for entry into the CS one (e.g. thread 1) succeeds, if thread 1 finishes up the CS, completes the doorway again and waits for entry, it must have set the victim to itself and will read that the other thread awaits entry, therefore the other thread 0 will succeed with no possibility of one thread lapping the other.

Exercise 10

Why is a doorway section needed? Argue for each case: read first, write to seperate or same location.

- Read first: Assuming that the first read instruction can tell a thread its position in a queue or similar, there has to be a write as well since otherwise future threads can't know that there was this one before them, so no ordering based on this instruction alone is possible.
- Write to different locations: Similar problem as before, 2 writes happening in sequence can't determine order, since the threads would have to read the other threads' writes first.
- Write to same location for 2 threads: Writing to the same location is basically half of the Peterson lock (write to victim variable). As mentioned in the slides, this can lead to the problem that if only one thread is left, this thread will be blocked since the latest write is the thread that came in last and thus will not be given entry to the CS.
- Write to same location for $n > 2$ threads: Only information about the last write is preserved, thus no statement about the order of previous threads can be made.

Exercise 11

Answer 1

Does the Flaky lock satisfy mutex?

Yes:
Case 1: One thread is already in the critical section.
If one thread is already in the critical section, then any other thread trying to enter the CS would need to read `busy == false`. This leads to a contradiction since the thread in the CS must have set `busy = true` when before entering and will only set it to `false` when leaving the CS.

Case 2: Threads racing to enter the CS.
It is possible for multiple threads to pass the `while(busy)` check, however, mutex is still satisfied since only one thread will be able to pass the `while(turn != me)` check. For another thread to enter the CS as well it would need loop again, set `turn` to itself and pass the `while(busy)` check again, which leads to a contradiction, since `busy` was set to `true` by that thread and the one that entered the CS and will only be set to `false` when no thread remains in the CS.

Answer 2

Starvation- & Deadlock-free?

The Flaky lock is not deadlock-free and thus not starvation-free. Assuming 2 threads, if one thread reaches `busy = true` and then the other thread sets `turn = me` before the first thread enters the CS, both threads will be stuck. The first one will fail the `while(turn != me)` check, will repeat the outer loop and get stuck on `while(busy)`, since it has set `busy = true` before. The other thread will never have made it past `while(busy)` in the first place. Now both threads can't continue since `busy` will never be set to `false` again.

Exercise 12

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

This can be shown with 3 threads:
Assume 3 threads (t_1, t_2, t_3) enter the first level and t_1 goes to sleep for an arbitrary amount of time. If t_3 enters last, it will be the victim thread and the other 2 can enter the next level. If t_2 now completes the lock and following CS and enters the lock again, t_2 will be the victim and t_1 and t_3 are free to enter the next level. t_3 can now complete the lock and CS and enter the lock again, then we end up in the starting situation. In this scenario, there is nothing stopping t_2 and t_3 from lapping t_1 an arbitrary amount of times.

Exercise 13

Binary tree Peterson lock.

Answer 1

Does it satisfy mutex?

Assume the parent nodes as critical sections of 2 underlying threads/nodes. Since the Peterson lock is correct, the parent nodes (critical sections) of 2 leaf locks are guaranteed to only contain one thread. If we now assume that the following locks higher up in the tree are also correct if they only have 2 threads fighting for the lock and each parent only has 2 child nodes we can guarantee correct locking up to the root node, where only one thread will be able to acquire the lock for the actual critical section.

Proof by contradiction:
For the final lock to work incorrectly, a node lock/CS fighting for the final lock needs to allow 2 or more threads to acquire its lock at the same time. This is only possible, if one of the 2 locks leading to the root lock failed, which means that one of their child locks failed, and so on. This means that the blame can be pushed down to the leaf locks, which are guaranteed to work with 2 threads, thus leading to a contradiction.

Answer 2

Deadlock freedom?

Proof by contradiction:
For a deadlock to occur there must be a scenario where no thread ever acquires the root lock. If any thread from the roots children tries to acquire it, it must work, since the Peterson lock itself is deadlock-free, which implies that the both children don't let any thread acquire their locks. However, these lock nodes again are deadlock-free Peterson locks, so we can again push down the blame to the leaf locks. Since those are also deadlock-free Peterson locks, we have a contradiction.

Answer 3

Starvation freedom?

Proof by contradiction:
For starvation to occur, there needs to be a scenario where a thread will not progress and get lapped by other threads infinitely often. At the leaf locks this could only happen if the parent of a leaf lock only gives the lock to one child since the leaf locks are Peterson locks which are starvation-free (and even fair). On the other hand, each tree note excluding the leaf locks could be starved if the unlocking mechanism was from leaf to root instead of the other way around since then a child could switch threads forever and never give up the parent lock. However, since unlocking works from root to leaf, starvation would require that a lock allowed a child thread to overtake another thread waiting on its lock. Since any Peterson lock itself is correct and starvation free, this is a contradiction.

Answer 4

Upper bound for acquire and release of the tree lock before a thread will succeed?

If no thread ever goes to sleep:
Each time a node/lock in the tree is active, it has to give the lock to the other child the next time it is active. This means that our worst case is that in which we have to do the maximum amount of "switching" between 2 threads, i.e. that there is always a thread waiting on the other side of each node/lock in the path to our victim thread. We can model the nodes in the path to the victim as a binary number, where the bit values indicate if we choose the left or right child. For the worst case we have to iterate through every possible combination of that number before getting to our thread, which means that for a tree with depth n, a thread to wait at most $2^n - 1$ turns before it is guaranteed the lock.

If threads can go to sleep, they can lap each other an arbitrary amount of times.
Consider 2 threads t_0 and t_1 at the same leaf lock and some other thread t_2. If t_0 sets its busy flag and victim variable and goes to sleep before entering the CS, thread t_1 would set the victim to itself, see that t_0 is busy and wait. However, since neither t_0 nor t_1 actually acquire the lock, that node never actually "forwards" any of those threads, and any other thread t_2 could acquire any of the locks in the path of t_0 and t_1 (except their leaf lock obviously) an arbitrary amount of times.

Exercise 14

Filter implementation for reference:

```
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++) { //attempt level 1
      level[me] = i;
      victim[i] = me;
      // spin while conflicts exist
      while ((∃k != me) (level[k] >= i && victim[i] == me)) {};
    }
  }
  public void unlock() {
    int me = ThreadID.get();
    level[me] = 0;
  }
}
```

Turn the Filter mutex lock into an ℓ-exclusion algorithm.

To make the Filter lock ℓ-exclusive, we have to change the `while` check to make sure that a thread does only get held up if ℓ threads are ahead (with ahead meaning at a level greater or equal). Therefore I introduced a variable `num_ahead[]` which counts up the amount of threads that are on the same or an above level as this thread. If a thread is not the victim at the current level or the allowed amount (ℓ) of threads is not already ahead of a thread, it can continue. Furthermore, while this alone would already work, one could also adjust the amount of levels from $n-1$ to $n-(\ell-1)-1$, i.e. change n to $n-(\ell-1)$ in the code, since $n-(n-(\ell-1)-1)=\ell$. This improvement sets the amount of levels to the maximum amount of threads we need to hold up.

```
class Filter implements Lock {
  int[] level;
  int[] victim;
  int[] num_ahead;
  public Filter(int n-(ℓ-1)) {
    level = new int[n-(ℓ-1)];
    victim = new int[n-(ℓ-1)];
    for (int i = 0; i < n-(ℓ-1); i++) {
      level[i] = 0;
    }
  }
  public void lock() {
    int me = ThreadID.get();
    for (int i = 1; i < n-(ℓ-1); i++) { //attempt level 1
      level[me] = i;
      victim[i] = me;
      // spin while conflicts exist
      do {
        num_ahead[me] = 0;
        for (int k = 0; k < n-(ℓ-1); k++) {
          if (k == me) continue;
          if (level[k] >= i) num_ahead[me]++;
        }
      } while (num_ahead[me] >= ℓ && victim[i] == me);
    }
  }
  public void unlock() {
    int me = ThreadID.get();
    level[me] = 0;
  }
}
```

Exercise 15

Is the FastPath implementation correct?

No, we can construct a scenario where 2 threads compete for the CS, one gets the FastPath lock and one still acquires the regular lock (`lock.lock()`). This can happen if both threads make the `while(y != -1)` check before a thread writes `y = i`. Now one thread will see `x != i` and acquire the regular lock while the other thread will see `x == i` and also make it to the critical section with the Fastpath lock.

Exercise 34

Does the mentioned construct yield a safe M-valued MRSW register?

True. Each thread only reads a single register, if reads are not overlapping with writes, they will return the most recently written value ("no future or distant past values"). If a read overlaps with a write, it can return any value anyway since it only has to satisfy the safe requirement.

Exercise 35

Does the mentioned construct yield a regular boolean MRSW register?

True. Each thread only reads one register, if reads are not overlapping with write, they will read the most recently written value ("no future or distant past values"). If reads are overlapping with a write, they will either return the old or the new value, since the single registers the threads are reading from are regular.

Exercise 36

Does the mentioned construct yield an atomic MRSW register?

False, because we can construct a scenario where the condition that once an updated value is read (during a write), subsequent reads must also return that new value is violated. Assume 2 threads t_0 and t_1, t_0 is the writer and t_1 a reader. t_0 now wants to update the value from x to y and let's say that it is currently writing to the second diagonal element (`r_table[1][1]`) and then falls asleep (for the duration of this scenario), which means that a read from that regular element could either return the new or the old value. Let's assume that t_1 now reads and returns y (new), it compares that timestamp to its matrix column, sees that it is a new value and so it updates its row with y(but not the `r_table[1][1]` element itself!) and returns y. Now t_1 reads again and reading `r_table[1][1]` specifically can return either x or y again. If we now assume that the second time x (old) is returned, then a comparison with the table column will show that that value is still new (no other thread has updated its row with any new values), and thus the full read will return x, which violates the aforementioned condition.

Exercise 37

Give an example of a quiescently-consistent register execution that is not regular.

We can construct an example where a reader and a writer thread always overlap, thus the w/r object is never quiescent.
Writer: | w(1) | | w(2) | | w(3) | ...
Reader: | r(1) | | r(1) | | r(1) | ...
We can see that the execution is quiescently-consistent (the object doesn't go quiescent for multiple writes/reads so no problem here), and the reads return non-regular values.

Exercise 38

Does the mentioned construct yield a regular M-valued MRSW register?

True. For one, a thread will not read any future values, nor will it return an old value once a write has completed with a new value. The critical condition is that overlapping reads and writes either return the old or the new value, and not just any value from the domain. This condition holds, because each thread only reads from a single register and those underlying registers are regular, meaning that a concurrent read/write to a register will behave regularly.

Exercise 39

Does the mentioned construct yield a regular M-valued MRSW register?

False. The regular boolean MRSW register uses the binary property of the boolean to satisfy the regularity condition. `if (x != last.get())` covers the case where the new value matches the old value in a write. This works in the boolean and the M-valued case, since it avoids a (potentially concurrent) write if the value stays the same. In the other case however where the new value does not match the old one, the boolean approach works because the domain of the boolean only consists of 1 and 0, so either value is correct (read can return either the old or the new value). In the M-valued case, this can cause a read concurrent to a write to return any value from the value domain which violates the mentioned condition.

Exercise 40

Does Peterson's two-thread mutex algorithm work if the shared atomic `flag` registers are replaced by regular registers?

Yes, it still works. Otherwise we would have to construct a scenario where now the Lock works incorrectly (no mutex, deadlock, ...). Since the change only concerns the acceptable read returns in the case of concurrency, we can isolate those cases.
In short, nothing bad can happen, since when one thread writes to the flag, the other can still enter the CS without bad consequences, and as soon as the second or both threads actually want to enter the CS at the same time (danger), every write is already settled or the victim variable acts as a safeguard and everything is correct.
The first possible concurrent read/write happens when thread 1 (t_1) wants to read the flag in the while-loop while thread 2 (t_2) writes to its flag element. We also only have to consider the case where the read doesn't behave atomically, so in this case t_1 reads `true`, then `false` and thus enters the CS. This is still ok, since at this point, there is still no possibility of violation of mutex, deadlock, or starvation. If t_2 hypothetically writes for ever, t_1 will just enter the CS each time the busy flag is `false`, otherwise it will wait. At any moment where the 2 threads actually compete for the lock, the critical write must already be complete and the algorithm works as intended. The most dangerous situation would be if t_1 reads that t_2 is not busy and before t_1 enters the CS, t_2 goes to the while-loop. At this point however, t_2 is the victim and t_1 can enter anyway.

The other possible case is when t_1 unlocks and t_2 awaits entry to the CS at the while-loop. Here also nothing bad can happen. As long as t_2 reads `true`, it cannot enter the CS anyway and everything works as expected. If t_2 reads `false` once (before the write is finished), nothing bad can happen since t_1 is not in the CS anymore. The most critical situation would be similar as above, when t_2 reads `false` and t_1 races to the while-loop before t_2 actually enters the CS. Then t_1 must set the victim variable to itself and t_2 can legally acquire the lock anyway.

Exercise 41

Answer 1

`write()` calls don't overlap.
Is this register implementation regular?

Yes. At processor P_n there are only sequential reads and writes to its local register, so if a `write` round was started and a read happens, either the new value has already reached that processor and the values was copied to the register (returns new value), or the message has either not reached that proccessor yet or the local write has not happened yet, in which case the old value will be returned. Since in this example `write()` calls don't overlap, this holds for all processors and the construct is regular.

Answer 2

`write()` calls don't overlap.
Is it atomic?

The construct is not atomic, since there is no way to distribute a new value to all local registers before the `write()` call is finished. As an example, P_0 wants to write some value x. Let's assume the message reaches P_i so if we now read the value stored in P_{i-1} and then the value in some P_j, $j > i$ where the message has not arrived yet, we will first read the new value and then the old one, which violates the atomicity condition.

Answer 3

Multiple processors can call `write()`.
Is this register implementation safe?

No. Imagine 4 processors $P_0, \ldots, P_3$ and P_0 and P_2 receive write calls x and y at the same time. If we assume for the sake of simplicity that all processors work in lockstep, P_0 will pass the message x to P_2 and P_2 will pass its message y to P_0 without problems. Afterwards however, the messages will overwrite each other, so that at the end P_1 and P_2 will have stored y, and P_3 and P_0 will have stored x. Now that both write operations are complete, reads from P_1 and P_2 will yield a different result than reads from P_0 and P_3, thus violating the condition that a read will return the most recent write value.

Exercise 43

Prove that the mentioned construct is a correct regular MRSW register.

For the construct not to be regular, it would need to violate the condition that a read during a write would neither yield the old nor the new value (the other conditions trivially hold since they already hold for underlying safe registers). In the `read()` call, a thread only reads from a single regular register, which means that if the read during a write yields an incorrect result, the underlying register could not have been regular, which leads to a contradiction. See also exercises 35 and 38.