Programming Languages

Dr. Michael Petter, Raphaela Palenta WS 2018/19

Exercise Sheet 4

Assignment 4.1 Memory Consistency

1. Given an execution path for each thread, what property does the hardware (or the model) have if only a single interleaving is possible?

   ☒ strict consistency
   ☐ sequential consistency
   ☐ weak consistency

2. What consistency guarantee does a system with a MESI cache but without store or invalidate buffers give?

   ☐ strict consistency
   ☒ sequential consistency
   ☐ weak consistency

3. A program reaching a state S (declared variables, values of variables, etc.) on weakly consistent hardware can always reach the same state S on sequentially consistent hardware. ☐yes ☒no

Assignment 4.2 Semaphores, Locks, and Monitors

Are the following statements true or false?

true false

1. A semaphore can be used to implement a mutex. ☒ ☐

   A mutex is a special kind of semaphore, thus: yes

2. A mutex is always re-entrant. ☐ ☒

   No, the monitor is a variant of the mutex, which is re-entrant

3. A monitor can be used as a mutex. ☒ ☐

   Use the monitor to protect the semaphore counter `s`, and use a condition variable for `wait()` and `signal()`

4. Any deadlock-free program must acquire locks in a fixed order. ☐ ☒

   No, this is only a condition that is sufficient to ensure that deadlocks do not occur.

5. When acquiring locks in a fixed order to ensure deadlock-freedom, there is no advantage in releasing them in the opposite order. ☐ ☒

   No, releasing them in opposite order has a performance advantage. Releasing them in the same order as they were acquired may be less efficient: Suppose thread A acquires three locks in the order l_1, l_2, l_3. A second thread B requires locks l_2 and l_3

and will therefore try to acquire l_2 before l_3. If thread A releases them in the sequence l_1, l_2, l_3 then as soon as l_2 is unlocked, the OS might schedule thread B which then immediately blocks again waiting for l_3. Thread A must now be scheduled in order to release l_3 before the OS can re-schedule B to acquire lock l_3. Thus, releasing lock in any order that is not the reverse of the locking order can incur a performance penalty.

6. The use of which concurrency construct may lead to starvation?

   ☐ a wait-free algorithm
   by definition never waits, nor fails, thus eventually completes

   ☒ a lock-free algorithm
   might fail and start over, thus might get trapped in an infinite loop

   ☐ a lock where blocking threads are put into a queue
   given enough signals, the critical section will eventually be executed

   ☐ a signal-and-urgent-wait monitor where all waiting threads are tracked in queues
   same here: given enough signals, the critical section will eventually be executed

7. Consider all program points p with the statement `lock(`a_p`)` and a lock set L_p. Which statement is true?

   ☐ The program is free of deadlocks if a_p is a lock and $a_p \in L_p$.
   contrarily, this would rather indicate, that there may be a deadlock

   ☒ The program may have a deadlock if a_p is a lock and $a_p \in L_p$.
   depending on a_p being either a monitor or semaphore, the program maybe or definitely has a deadlock – however, we can rule out that it is definitely free from deadlocks

   ☐ The program will deadlock if a_p is a lock and $a_p \in L_p$.
   would rely on a_p being either a semaphore or a monitor. However, with a_p a lock only, a definitely occuring deadlock is to strong

   ☐ The program is free of deadlocks if $a_p \in L_p$ implies that a_p is a monitor.
   for freedom of deadlocks, the ordering between different monitors needs to be globally irreflexive – which is not guaranteed just from this local property.

8. Consider the program P whose synchronization between its two threads is given by the following two program fragments. According to the definition of a deadlock

```
wait(A);
if (rnd()) {
  wait(B);
  if (rnd()) {
    wait(C);
    // compute
    signal(C)
  }
  signal(B);
}
signal(A);
```

```
wait(B);
if (rnd()) {
  wait(C);
  if (rnd()) {
    wait(D);
    // compute
  }
}
signal(B);
signal(C);
signal(D);
```

- ☐ P may deadlock. There exists a lock order between the locks.
- ☐ P may deadlock. There exists no lock order between the locks.
- ☒ P cannot deadlock. There exists a lock order between the locks.
  all locksets at `wait` instructions together with their locksets contribute to the lockorder $A < B < C < D$. Thus the freedom of deadlock theorem holds.
- ☐ P cannot deadlock. There exists no lock order between the locks.

9. By recording an interleaving of a program at runtime, we observe the following: Thread 1 releasing a lock A is descheduled and another Thread 2 is scheduled that then executes holding the same lock A.

   - ☐ This behavior should never happen since it violates the mutual exclusion property, so there must be an error in the program.
   - ☒ The lock is a signal-and-urgent-wait monitor.
   - ☐ The lock must be a signal-and-continue monitor.

Assignment 4.3 Deadlocks

Consider the following four functions:

```
f() {
  ...
  wait(A);
    u();
  signal(A);
  ...
}
```

```
g() {
  ...
  wait(A);
    v();
  signal(A);
  ...
}
```

```
u() {
  ...
  wait(B);
    wait(C);
      ...
    signal(C);
  signal(B);
  ...
}
```

```
v() {
  ...
  wait(C);
    wait(B);
      ...
    signal(B);
  signal(C);
  ...
}
```

1. Additionally, we are given a main function that runs `f` and `g` in parallel:

```
main() {
  f(); || g();
}
```

   Can this possibly cause a deadlock? If not, try to prove it using the *freedom of deadlock* theorem.

2. Assuming there is no possible deadlock, how can we change the main function in a simple way to render a deadlock possible?

3. Finally, we change the main function so that it runs `f` and `g` sequentially:

```
main() {
  f();
  g();
}
```

   Obviously, no deadlock can occur (no parallelism and no lock is acquired multiple times without releasing it in between). Again try to prove this using the *freedom of deadlock* theorem.

Suggested Solution 4.3

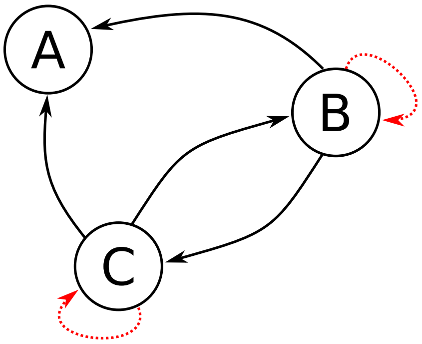
1. The parallel execution cannot cause a deadlock because both threads need to hold the lock A before entering `u` or `v`, respectively: `u` and `v` are executed sequentially. Additionally, it is obvious that in any of the two sequential executions of `u` and `v` no individual lock is reacquired before releasing it. In order to prove deadlock freedom, we first calculate the lock sets $\lambda(p)$ for each program point p (identified by its line number) and the respective new elements added to the lock order:

   - $\lambda(2) = \emptyset$
   - $\lambda(3) = \{A\}$, new lock order elements: $\emptyset$
   - $\lambda(4) = \{A\}$
   - $\lambda(5) = \emptyset$
   - $\lambda(6) = \emptyset$
   - $\lambda(9) = \emptyset$
   - $\lambda(10) = \{A\}$, new lock order elements: $\emptyset$
   - $\lambda(11) = \{A\}$
   - $\lambda(12) = \emptyset$
   - $\lambda(13) = \emptyset$
   - $\lambda(16) = \{A\}$
   - $\lambda(17) = \{A, B\}$, new lock order elements: $\{A \lhd B\}$
   - $\lambda(18) = \{A, B, C\}$, new lock order elements: $\{A \lhd C, B \lhd C\}$
   - $\lambda(19) = \{A, B, C\}$
   - $\lambda(20) = \{A, B\}$
   - $\lambda(21) = \{A\}$
   - $\lambda(22) = \{A\}$
   - $\lambda(25) = \{A\}$
   - $\lambda(26) = \{A, C\}$, new lock order elements: $\{A \lhd C\}$
   - $\lambda(27) = \{A, B, C\}$, new lock order elements: $\{A \lhd B, C \lhd B\}$
   - $\lambda(28) = \{A, B, C\}$
   - $\lambda(29) = \{A, C\}$
   - $\lambda(30) = \{A\}$
   - $\lambda(31) = \{A\}$
   - $\lambda(33) = \emptyset$
   - $\lambda(34) = \emptyset$

   Altogether, we obtain the following set of lock order elements:

   $$\{A \lhd B, A \lhd C, B \lhd C, C \lhd B\}$$

   This results in the following graph representation of the $\prec$ relation (the dashed red arrows show additional elements contained in the transitive closure of $\lhd$ only):



2. A deadlock is possible as soon as we refrain from always obtaining the *protective* lock A, e.g. by changing the `main` function as follows:

```
main() {
  f(); || v();
}
```

3. The proof does not change and, thus, fails to ascertain freedom of deadlocks, again. We thereby notice that even in very obvious cases one cannot rely on the *freedom of deadlock* theorem to indicate the presence of deadlocks if it fails to prove the freedom of deadlocks.