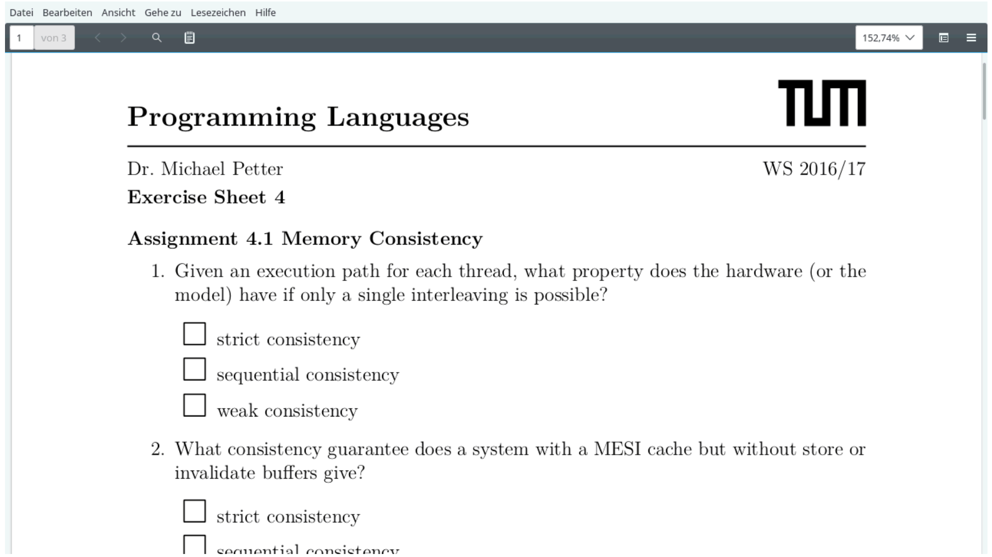
Script **generated by TTT**

Title: Petter: Programmiersprachen_Uebung (18.11.2016)

Date: Fri Nov 18 08:37:11 CET 2016

Duration: 66:57 min

Pages: 18



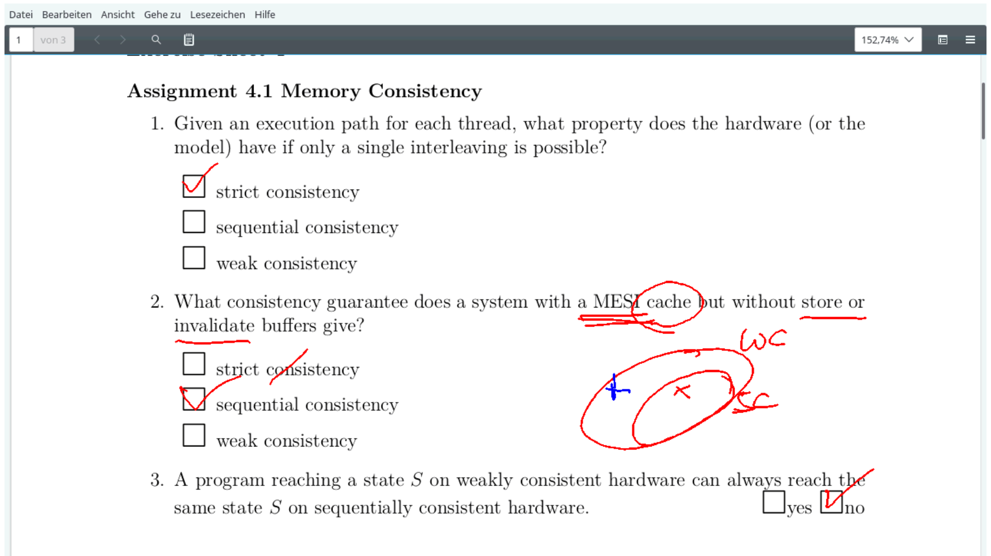
Programming Languages

Dr. Michael Petter WS 2016/17

Exercise Sheet 4

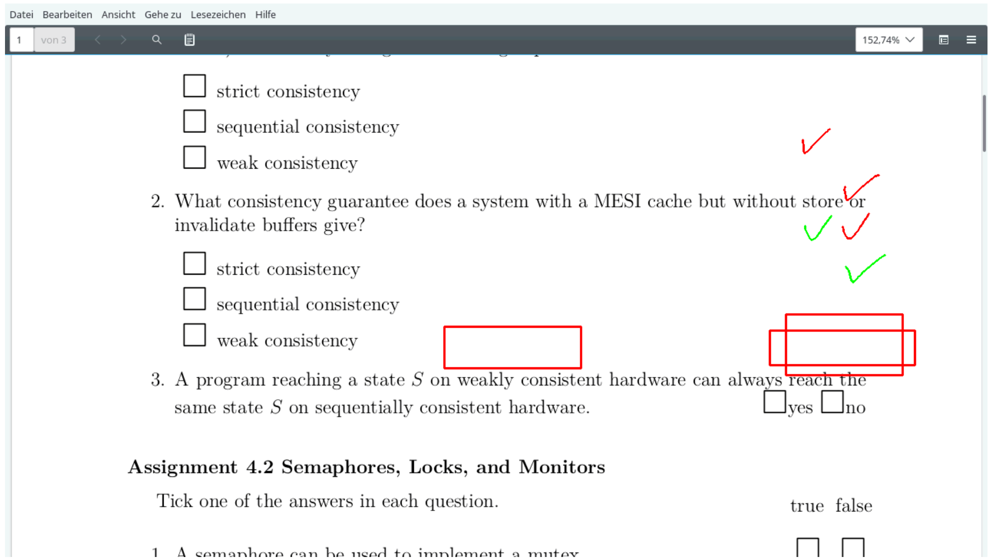
Assignment 4.1 Memory Consistency

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



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 on weakly consistent hardware can always reach the same state S on sequentially consistent hardware. ☐yes ☐no



- ☐ 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 on weakly consistent hardware can always reach the same state S on sequentially consistent hardware. ☐yes ☐no

Assignment 4.2 Semaphores, Locks, and Monitors

Tick one of the answers in each question. true false

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

1. A semaphore can be used to implement a mutex.
2. A mutex is always re-entrant. ☐ ☐
3. A monitor can be used as a mutex. ☐ ☐
4. Any deadlock-free program must acquire locks in a fixed order. ☐ ☐
5. When acquiring locks in a fixed order to ensure deadlock-freedom, there is no advantage in releasing them in the opposite order. ☐ ☐
6. The use of which concurrency construct may lead to starvation, that is, a thread that never manages to execute the critical section to completion, given arbitrary many chances?

- ☐ a wait-free algorithm
- ☐ a lock-free algorithm
- ☐ a lock where blocking threads are put into a queue
- ☐ a signal-and-urgent-wait monitor where all waiting threads are tracked in queues

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe
1 von 3 152,74%

3. A monitor can be used as a mutex. ☐ ☐
4. Any deadlock-free program must acquire locks in a fixed order. ☐ ☐
5. When acquiring locks in a fixed order to ensure deadlock-freedom, there is no advantage in releasing them in the opposite order. ☐ ☐
6. The use of which concurrency construct may lead to starvation, that is, a thread that never manages to execute the critical section to completion, given arbitrary many chances?

- ☐ a wait-free algorithm
- ☐ a lock-free algorithm
- ☐ a lock where blocking threads are put into a queue
- ☐ a signal-and-urgent-wait monitor where all waiting threads are tracked in queues

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe

- ☐ The program may have a deadlock if a_p is a lock and $a_p \in L_p$.
- ☐ The program will deadlock if a_p is a lock and $a_p \in L_p$.
- ☐ The program is free of deadlocks if $a_p \in L_p$ implies that a_p is a monitor.

8. Suppose that a program was shown to be deadlock free using the lock-order argument. This approach to dealing with deadlocks is called

- ☐ deadlock detection.
- ☐ deadlock prevention.
- ☐ deadlock avoidance.
- ☐ ignoring deadlock.

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

```
wait(A);                    wait(B);
if (rnd()) {                if (rnd()) {
  wait(B);                    wait(C);
  if (rnd()) {                if (rnd()) {
    wait(C);                    wait(D);
```

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe
2 von 3 152,74%

```
if (rnd()) {                if (rnd()) {
  wait(B);                    wait(C);
  if (rnd()) {                if (rnd()) {
    wait(C);                    wait(D);
    // compute                  // compute
    signal(C)                 }
  }                         }
  signal(B);                signal(B);
}                           signal(C);
signal(A);                  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.
- ☐ P cannot deadlock. There exists no lock order between the locks.

10. By recording an interleaving of a program at runtime, we observe the following: A thread that holds a lock is descheduled and another thread is scheduled that then executes holding the same lock.

- ☐ 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.
- ☐ P cannot deadlock. There exists no lock order between the locks.

10. By recording an interleaving of a program at runtime, we observe the following: A thread that holds a lock is descheduled and another thread is scheduled that then executes holding the same lock.

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

11. The `enter` operation of a monitor is called `notify` in Java. ☐ ☐


```
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() {
```

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe
3 von 3 152,74%

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?

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe
3 von 3 152,74%

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() {
```

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe

3 von 3 152,74%

```
    u();
  signal(A);
  ...
}
```

```
    v();
  signal(A);
  ...
}
```

```
    ...
    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.

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe

3 von 3 152,74%

```
  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() {
```

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe

3 von 3 152,74%

```
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() {
```

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe

3 von 3 152,74%

```
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.

```
  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() {
```

Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe
3 von 3 152,74%

```
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() {
```


Datei Bearbeiten Ansicht Gehe zu Lesezeichen Hilfe
3 von 3 152,74%

```
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.