Script generated by TTT

Title: Petter: Programmiersprachenh (14.11.2018)

Date: Wed Nov 14 14:11:50 CET 2018

Duration: 86:35 min

Pages: 27

Deadlocks

Deadlocks with Monitors

Definition (Deadlock)

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Deadlocks with Monitors

Definition (Deadlock)

A deadlock is a situation in which two processes are waiting for the respective other to finish, and thus neither ever does.

(The definition generalizes to a set of actions with a cyclic dependency.)

Consider this Java class:

```
class Foo {
  public Foo other = null;
  public synchronized void bar() {
    ... if (*) other.bar(); ...
  }
}
```

and two instances:

```
  Foo a = new Foo();
  Foo b = new Foo();
  a.other = b; b.other = a;
  // in parallel:
  a.bar() || b.bar();
```

Sequence leading to a deadlock:

- threads A and B execute `a.bar()` and `b.bar()`
- `a.bar()` acquires the monitor of `a`
- `b.bar()` acquires the monitor of `b`
- A happens to execute `other.bar()`
- A blocks on the monitor of b
- B happens to execute `other.bar()`
- $\rightsquigarrow$ both *block* indefinitely

Treatment of Deadlocks

Observation: Deadlocks occur if the following four conditions hold [Coffman et al.(1971)Coffman, Elphick, and Shoshani]:

1. *mutual exclusion*: processes require exclusive access
2. *wait for*: a process holds resources while waiting for more
3. *no preemption*: resources cannot be taken away form processes
4. *circular wait*: waiting processes form a cycle

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks are *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, that is, the set of locks that may be in the "acquired" state at program point p.

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks are *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, that is, the set of locks that may be in the "acquired" state at program point p.

We require the transitive closure σ^+ of a relation σ:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

$$\begin{array}{rcl} \sigma^0 & = & \sigma \\ \sigma^{i+1} & = & \{\langle x_1, x_3\rangle \mid \exists x_2 \in X . \langle x_1, x_2\rangle \in \sigma^i \wedge \langle x_2, x_3\rangle \in \sigma^i\} \end{array}$$

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks are *partially ordered*.

Definition (lock sets)

Let L denote the set of locks. We call $\lambda(p) \subseteq L$ the lock set at p, that is, the set of locks that may be in the "acquired" state at program point p.

We require the transitive closure σ^+ of a relation σ:

Definition (transitive closure)

Let $\sigma \subseteq X \times X$ be a relation. Its transitive closure is $\sigma^+ = \bigcup_{i \in \mathbb{N}} \sigma^i$ where

$$\begin{array}{rcl} \sigma^0 & = & \sigma \\ \sigma^{i+1} & = & \{\langle x_1, x_3\rangle \mid \exists x_2 \in X . \langle x_1, x_2\rangle \in \sigma^i \wedge \langle x_2, x_3\rangle \in \sigma^i\} \end{array}$$

Each time a lock is acquired, we track the lock set at p:

Definition (lock order)

Define $\lhd \subseteq L \times L$ such that $l \lhd l'$ iff $l \in \lambda(p)$ and the statement at p is of the form `wait(l')` or `monitor_enter(l')`. Define the strict lock order $\prec = \lhd^+$.

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.

Freedom of Deadlock

The following holds for a program with mutexes and monitors:

Theorem (freedom of deadlock)

If there exists no $a \in L$ with $a \prec a$ then the program is free of deadlocks.

Suppose a program blocks on semaphores (mutexes) L_S and on monitors L_M such that $L = L_S \cup L_M$.

Theorem (freedom of deadlock for monitors)

If $\forall a \in L_S \, . \, a \not\prec a$ and $\forall a \in L_M, b \in L \, . \, a \prec b \wedge b \prec a \Rightarrow a = b$ then the program is free of deadlocks.

Avoiding Deadlocks in Practice

How can we verify that a program contains no deadlocks?

1. identify mutex locks L_S and summarized monitor locks $L^s_M \subseteq L_M$
2. identify non-summary monitor locks $L^n_M = L_M \setminus L^s_M$
3. sort locks into ascending order according to lock sets
4. check that no cycles exist except for self-cycles of non-summary monitors

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

stack: removal

```
void pop() {
  ...
  wait(&q->t);
  ...
  if (*) { signal(&q->t); return; }
  ...
  if (c) wait(&q->s);
  ...
  if (c) signal(&q->s);
  signal(&q->t);
}
```

Outlook

Writing `atomic` annotations around sequences of statements is a convenient way of programming.

Outlook

Writing `atomic` annotations around sequences of statements is a convenient way of programming.

Idea of mutexes: Implement `atomic` sections with locks:

- a single lock could be used to protect all `atomic` blocks
- more concurrency is possible by using several locks
- some statements might modify variables that are never read by other threads $\rightsquigarrow$ no lock required
- statements in one `atomic` block might access variables in a different order to another `atomic` block $\rightsquigarrow$ deadlock possible with locks implementation
- creating too many locks can decrease the performance, especially when required to release locks in $\lambda(l)$ when acquiring l

Concurrency across Languages

In most systems programming languages (C,C++) we have

- the ability to use *atomic* operations

$\rightsquigarrow$ we can implement *wait-free* algorithms

Concurrency across Languages

In most systems programming languages (C,C++) we have

- the ability to use *atomic* operations

$\rightsquigarrow$ we can implement *wait-free* algorithms

In Java, C# and other higher-level languages

- provide monitors and possibly other concepts
- often simplify the programming but incur the same problems

language	barriers	wait-/lock-free	semaphore	mutex	monitor
C,C++	✓	✓	✓	✓	(a)
Java,C#	-	(b)	(c)	✓	✓

(a) some pthread implementations allow a *reentrant* attribute

(b) newer API extensions (`java.util.concurrent.atomic.*` and `System.Threading.Interlocked` resp.)

(c) simulate semaphores using an object with two `synchronized` methods

Summary

Classification of concurrency algorithms:

- wait-free, lock-free, locked
- next on the agenda: transactional

Wait-free algorithms:

- never block, always succeed, never deadlock, no starvation ✓
- very limited in expressivity ✗

Lock-free algorithms:

- never block, may fail, never deadlock, may starve ✓
- invariant may only span a few bytes (8 on Intel) ✗

Locking algorithms:

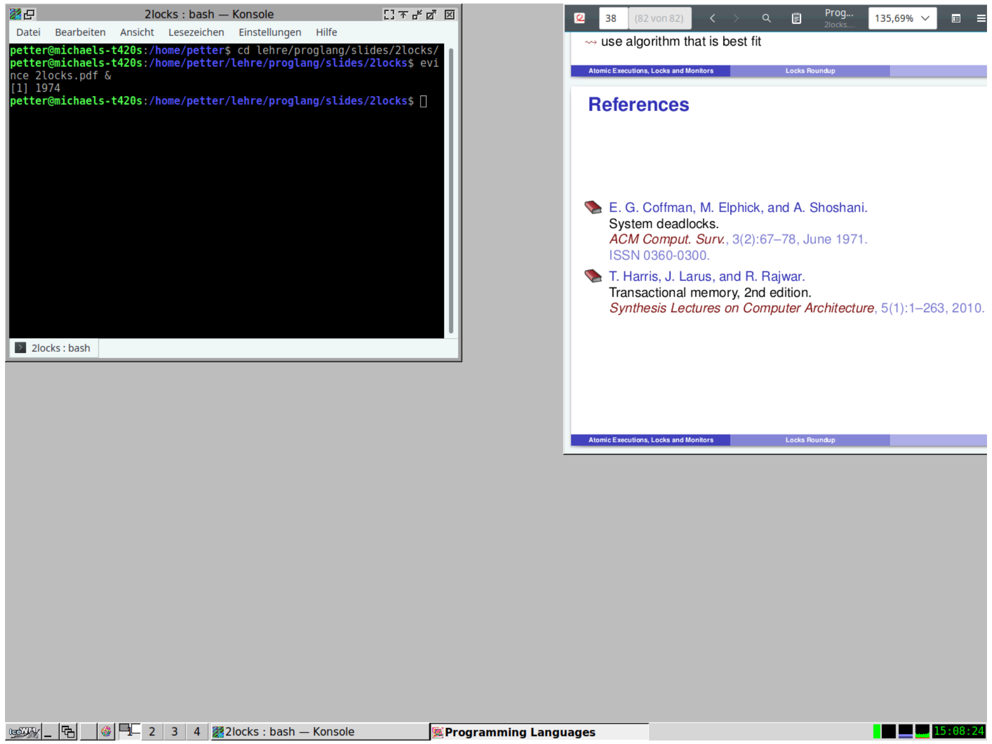
- can guard arbitrary code ✓
- can use several locks to enable more fine grained concurrency ✓
- may deadlock ✗
- semaphores are not re-entrant, monitors are

⇝ use algorithm that is best fit

References

E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ACM Comput. Surv., 3(2):67–78, June 1971.
ISSN 0360-0300.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.


2locks : bash — Konsole
Datei Bearbeiten Ansicht Lesezeichen Einstellungen Hilfe
2locks : bash
38 (82 von 82) Prog... 135,69%
2 3 4 2locks : bash — Konsole Programming Languages 15:08:24

```
petter@michaels-t420s:/home/petter$ cd lehre/proglang/slides/2locks/
petter@michaels-t420s:/home/petter/lehre/proglang/slides/2locks$ evi
nce 2locks.pdf &
[1] 1974
petter@michaels-t420s:/home/petter/lehre/proglang/slides/2locks$ 
```

⇝ use algorithm that is best fit

Atomic Executions, Locks and Monitors | Locks Roundup

References

E. G. Coffman, M. Elphick, and A. Shoshani.
System deadlocks.
ACM Comput. Surv., 3(2):67–78, June 1971.
ISSN 0360-0300.

T. Harris, J. Larus, and R. Rajwar.
Transactional memory, 2nd edition.
Synthesis Lectures on Computer Architecture, 5(1):1–263, 2010.

Atomic Executions, Locks and Monitors | Locks Roundup

Abstraction and Concurrency

Two fundamental concepts to build larger software are:

abstraction: an object storing certain data and providing certain functionality may be used without reference to its internals

composition: several objects can be combined to a new object without interference

Both, *abstraction* and *composition* are closely related, since the ability to compose depends on the ability to abstract from details.

Transactional Memory [2]

Idea: automatically convert `atomic` blocks into code that ensures atomic execution of the statements.

```
atomic {
  // code
  if (cond) retry;
  atomic {
    // more code
  }
  // code
}
```

Semantics of Transactions

The goal is to use transactions to specify *atomic executions*.
Transactions are rooted in databases where they have the *ACID* properties:

Semantics of Transactions

The goal is to use transactions to specify *atomic executions*.
Transactions are rooted in databases where they have the *ACID* properties:

atomicity : a transaction completes or seems not to have run
- ⇝ we call this *failure atomicity* to distinguish it from *atomic executions*

consistency : each transaction transforms a consistent state to another consistent state
- a consistent state is one in which certain *invariants* hold
- invariants depend on the application

isolation : transactions do not interfere with each other
- ⇝ not so evident with respect to non-transactional memory

durability : the effects are permanent ✓

Consistency During Transactions

Consistency during a transaction.

ACID states how committed transactions behave but not what may happen until a transaction commits.

- a transaction that is run on an inconsistent state may generate an inconsistent state ⇝ zombie transaction
- in the best case, the zombie transaction will be aborted eventually
- but transactions may cause havoc when run on inconsistent states

```
atomic {                          // preserved invariant: x==y
  int tmp1 = x;                   atomic {
  int tmp2 = y;                     x = 10;
  assert(tmp1-tmp2==0);             y = 10;
}                                 }
```

⚠ critical for null pointer derefs or divisions by zero, e.g.

Definition (opacity)

A TM system provides *opacity* if failing transactions are serializable w.r.t. committing transactions.

⇝ failing transactions still see a consistent view of memory

Weak- and Strong Isolation

If guarantees are only given about memory accessed inside `atomic`, a TM implementation provides *weak isolation*.
Can we mix transactions with code accessing memory non-transactionally?

- no conflict detection for non-transactional accesses
- standard *race* problems as in unlocked shared accesses

```
// Thread 1
atomic {                        // Thread 2
  x = 42;                       int tmp = x;
}
```

⇝ give programs with races the same semantics as if using a single global lock for all `atomic` blocks

- *strong isolation* retains order between accesses to TM and non-TM

Disadvantages of the SLA model

The SLA model is *simple* but often too strong:

1. SLA has a weaker *progress* guarantee than a transaction should have

```
// Thread 1                     // Thread 2
atomic {                        atomic {
  while (true) {};                int tmp = x; // x in TM
}                               }
```

2. SLA correctness is too strong in practice

```
                                // Thread 2
                                atomic {
// Thread 1                       int tmp = data;
data = 1;                         // Thread 1 not in atomic
atomic {                          if (ready) {
}                                   // use tmp
ready = 1;                        }
                                }
```

- ▸ under the SLA model, `atomic {}` acts as barrier
- ▸ intuitively, the two transactions should be independent rather than synchronize

⇝ need a weaker model for more flexible implementation of *strong isolation*

Transactional Sequential Consistency

How about a more permissive view of transaction semantics?

- TM should not have the blocking behaviour of locks

⇝ the programmer cannot rely on synchronization

Definition (TSC)

The *transactional sequential consistency* is a model in which the accesses within each transaction are sequentially consistent.

Software Transactional Memory