Script generated by TTT

Title: Simon: Programmiersprachen (25.11.2013)

Date: Mon Nov 25 14:17:23 CET 2013

Duration: 31:25 min

Pages: 14

∅ wait(A)
{A}
wait(B) $A \lhd B$
{A, B}
signal(B)
signal(A)
∅
wait(C)
{C}
wait(A) $C \lhd A$
{A, C}
--
∅

wait(C)
{C}
wait(D) $C \lhd D$
$p: \lambda(p) = \{C, D\}$
wait(A) $C \lhd A$ $D \lhd A$
{A, C, D}
signal(D)
{A, C}
signal(A)
{C}
signal(C)
∅

monitor_enter(F)
{F}
monitor_enter(F)
$F \lhd F$

$\prec = \lhd^+$
$\leftarrow = \lhd^+$

$l \lhd l'$
wait(l')
$\forall\, l \in \lambda(p)$

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be *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

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

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 $\prec = \lhd^+$.

Deadlock Prevention through Partial Order

Observation: A cycle cannot occur if locks can be *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

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

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 $\prec = \lhd^+$.

Atomic Executions, Locks and Monitors | Locked Atomic Executions | 32 / 41

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) at L_S and on monitors at 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 $\forall a \in L_M, b \in L \,.\, a \prec b \wedge b \prec a \Rightarrow a \neq b$ 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) at L_S and on monitors at 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 $\forall a \in L_M, b \in L \,.\, a \prec b \wedge b \prec a \Rightarrow a \neq b$ then the program is free of deadlocks.

Note: the set L contains *instances* of a lock.

- the set of lock instances can vary at runtime
- if we statically want to ensure that deadlocks cannot occur:
  - summarize every monitor that may have several instances into one
  - a summary lock $\bar{a} \in L_M$ represents several concrete ones
  - thus, if $\bar{a} \prec \bar{a}$ then this might not be a self-cycle
  - $\rightsquigarrow$ require that $\bar{a} \not\prec \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^n = L_M \setminus L_M^s$

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^n = L_M \setminus L_M^s$
- sort locks into ascending order according to lock sets $\prec \equiv \lhd^+$

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^n = L_M \setminus L_M^s$
- sort locks into ascending order according to lock sets

⚠ Ordering might be hard or impossible to find:

- determining which locks may be acquired at each program point is undecidable $\rightsquigarrow$ approximate lock set
- an array of locks: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists where $l' \prec l$ should be locked: release l, acquire l', then acquire l again $\rightsquigarrow$ inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

Avoiding Deadlocks in Practice

How can we modify a program so that locks can be ordered?

- identify mutex locks L_S and summarized monitor locks $L_M^s \subseteq L_M$
- identify non-summary monitor locks $L_M^n = L_M \setminus L_M^s$
- sort locks into ascending order according to lock sets

⚠ Ordering might be hard or impossible to find:

- determining which locks may be acquired at each program point is undecidable $\rightsquigarrow$ approximate lock set
- an array of locks: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists where $l' \prec l$ should be locked: release l, acquire l', then acquire l again $\rightsquigarrow$ inefficient
- if a lock set contains a summarized lock $\bar{a}$ and $\bar{a}$ is to be acquired, we're stuck

an example for the latter is the `Foo` class: two instances of the same class call each other

Refining the Queue: Concurrent Access

Add a second lock `s->t` to allow concurrent removal:

double-ended queue: removal

```
int PopRight(DQueue* q) {
  QNode* oldRightNode;
  wait(q->t); // wait to enter the critical section
  QNode* rightSentinel = q->right;
  oldRightNode = rightSentinel->left;
  if (oldRightNode==leftSentinel) { signal(q->t); return -1; }
  QNode* newRightNode = oldRightNode->left;
  int c = newRightNode==leftSentinel;
  if (c) wait(q->s);
  newRightNode->right = rightSentinel;
  rightSentinel->left = newRightNode;
  if (c) signal(q->s);
  signal(q->t); // signal that we're done
  int val = oldRightNode->val;
  free(oldRightNode);
  return val;
}
```

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

double-ended queue: removal

```
void PopRight() {
  ...
  wait(q->t);
{t}...
  if (*) { signal(q->t); ∅ return; }
{t}...
  if (c) wait(q->s);    t ◁ s
  {s,t}
  if (c) signal(q->s);
  {t}
  signal(q->t);
} ∅
```


Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

double-ended queue: removal

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

- in `PushLeft`, the lock set for s is empty
- here, the lock set of s is $\{t\}$
- $t \lhd s$ and transitive closure is $t \prec s$
- $\rightsquigarrow$ the program cannot deadlock

$s \downarrow t$

Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

double-ended queue: removal

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