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:

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

and two instances:

```java
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()
- → both block indefinitely

How can this situation be avoided?
### Treatment of Deadlocks

Deadlocks occur if the following four conditions hold [Coffman et al. (1971)]:

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 from processes
4. **circular wait**: waiting processes form a cycle

The occurrence of deadlocks can be:

1. **ignored**: for the lack of better approaches, can be reasonable if deadlocks are rare
2. **detection**: check within OS for a cycle, requires ability to preempt
3. **prevention**: design programs to be deadlock-free
4. **avoidance**: use additional information about a program that allows the OS to schedule threads so that they do not deadlock

---

### Deadlock Prevention through Partial Order

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

**Definition (lock sets)**

Let \( I \) denote the set of locks. We call \( \lambda(p) \subseteq I \) the lock set at point \( p \), 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 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 \( \sigma^+ \) of a relation \( \sigma \):

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \{ (x_1, x_3) \mid \exists x_2 \in X : (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma^j \}
\]

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

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \{ (x_1, x_3) \mid \exists x_2 \in X : (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma^i \}
\]

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

Definition (lock order)
Define \( \prec \subseteq L \times L \) such that \( l \prec l' \) if \( l \in \lambda(p) \) and the statement at \( p \) is of the form \( \text{wait}(l') \) or \( \text{monitor}_\text{enter}(l') \). Define the strict lock order \( \prec^+ \).

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.

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 \( \sigma^+ \) of a relation \( \sigma \):

\[
\sigma^0 = \sigma \\
\sigma^{i+1} = \{ (x_1, x_3) \mid \exists x_2 \in X : (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma^i \}
\]

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

Definition (lock order)
Define \( \prec \subseteq L \times L \) such that \( l \prec l' \) if \( l \in \lambda(p) \) and the statement at \( p \) is of the form \( \text{wait}(l') \) or \( \text{monitor}_\text{enter}(l') \). Define the strict lock order \( \prec^+ \).
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 \( a \prec \hat{a} \) then the program is free of deadlocks.

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 \( \sigma^+ \) of a relation \( \sigma \):

**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

\[
\begin{align*}
\sigma^0 & = \sigma \\
\sigma^{i+1} & = \{ \langle x_1, x_3 \rangle \mid \exists x_2 \in X. \langle x_1, x_2 \rangle \in \sigma^i \land \langle x_2, x_3 \rangle \in \sigma^i \}
\end{align*}
\]

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

**Definition (lock order)**

Define \( \prec \subseteq L \times L \) such that \( l \prec l' \) iff \( l \in \lambda(p) \) and the statement at \( p \) is of the form \textit{wait}(l') or \textit{monitor.enter}(l'). Define the strict lock order \( \prec^+ \).

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 \( \sigma^+ \) of a relation \( \sigma \):

**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

\[
\begin{align*}
\sigma^0 & = \sigma \\
\sigma^{i+1} & = \{ \langle x_1, x_3 \rangle \mid \exists x_2 \in X. \langle x_1, x_2 \rangle \in \sigma^i \land \langle x_2, x_3 \rangle \in \sigma^i \}
\end{align*}
\]

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

**Definition (lock order)**

Define \( \prec \subseteq L \times L \) such that \( l \prec l' \) iff \( l \in \lambda(p) \) and the statement at \( p \) is of the form \textit{wait}(l') or \textit{monitor.enter}(l'). Define the strict lock order \( \prec^+ \).
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 $\sigma^+$ of a relation $\sigma$:

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

$$
\sigma^0 = \sigma \\
\sigma^{i+1} = \{ (x_1, x_3) \mid \exists x_2 \in X, (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma \}
$$

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

Definition (lock order)

Define $\prec \subseteq L \times L$ such that $l \prec l'$ iff $l \in \lambda(p)$ and the statement at $p$ is of the form $\text{wait}(l')$ or $\text{monitor} \_\text{enter}(l')$. Define the strict lock order $\triangleleft := \prec^+$.

---

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 $\sigma^+$ of a relation $\sigma$:

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

$$
\sigma^0 = \sigma \\
\sigma^{i+1} = \{ (x_1, x_3) \mid \exists x_2 \in X, (x_1, x_2) \in \sigma^i \land (x_2, x_3) \in \sigma \}
$$

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

Definition (lock order)

Define $\prec \subseteq L \times L$ such that $l \prec l'$ iff $l \in \lambda(p)$ and the statement at $p$ is of the form $\text{wait}(l')$ or $\text{monitor} \_\text{enter}(l')$. Define the strict lock order $\triangleleft := \prec^+$.
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 \( \forall a \in L_M, a \neq a \) 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.

**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 lock/monitor that may have several instances into one
  - A summary lock/monitor \( \bar{a} \in L_M \) represents several concrete ones
  - Thus, if \( \bar{a} \prec \bar{a} \) then this might not be a self-cycle
  - Require that \( \bar{a} \neq \bar{a} \) for all summarized monitors \( \bar{a} \in L_M \)
Avoiding Deadlocks in Practice

How can we verify that a program contains no deadlocks?
- 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
- check that no cycles exist except for self-cycles of non-summary monitors

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 $\forall a \in L_S, a \nprec a$ and $\forall a \in L_M, b \in L, a \prec b \land b \prec a \Rightarrow a = b$ 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 lock/monitor that may have several instances into one
  - a summary lock/monitor $\bar{a} \in L_M$ represents several concrete ones
  - thus, if $\bar{a} \prec \bar{a}$ then this might not be a self-cycle
  - require that $\bar{a} \nprec \bar{a}$ for all summarized monitors $\bar{a} \in L_M$

Avoiding Deadlocks in Practice

How can we verify that a program contains no deadlocks?
- 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
- check that no cycles exist except for self-cycles of non-summary monitors

⚠️ What to do when lock order contains cycle?
- determining which locks may be acquired at each program point is undecidable → lock sets are an approximation
- an array of locks in $L_S$: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists $l' \prec l$ is to be acquired → change program: release $l$, acquire $l'$, then acquire $l$ again → 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 verify that a program contains no deadlocks?
- identify mutex locks $L_S$ and summarized monitor locks $L_M \subseteq L_M$
- identify non-summary monitor locks $L_M^\prime = L_M \setminus L_M$
- sort locks into ascending order according to lock sets
- check that no cycles exist except for self-cycles of non-summary monitors

⚠️ What to do when lock order contains cycle?
- determining which locks may be acquired at each program point is
  undecidable -- lock sets are an approximation
- an array of locks in $L_S$: lock in increasing array index sequence
- if $l \in \lambda(P)$ exists $l' < l$ is to be acquired -- change program: release $l$,
  acquire $l'$, then acquire $l$ again -- 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 
mean{Foo} class: two instances of the same class call
each other

Refining the Queue: Concurrent Access

Add a second lock $s \rightarrow t$ to allow concurrent removal/peeking:

```c

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 = oldRightNode->leftSentinel;
    if (c) wait(q->s);
    newRightNode->right = rightSentinel;
    rightSentinel->left = newRightNode;
    if (c) signal(q->s);
    free(oldRightNode);
    return val;
}
```

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

```c

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);
}
```

Example: Deadlock freedom

Is the example deadlock free? Consider its skeleton:

```c

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

- in \textbf{PushLeft} the lock set for $s$ is empty
- here, the lock set of $s$ is \{\textbf{t}\}
- \textbf{t} \textbf{<} \textbf{t}$ and transitive closure is $\textbf{t} \textbf{<} s$
- $\rightarrow$ the program cannot deadlock
Atomic Execution and Locks

Consider replacing the specific locks with `atomic` annotations:

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

- nested `atomic` blocks still describe one atomic execution
-locks convey additional information over `atomic`
-locks cannot easily be recovered from `atomic` declarations

### 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
  - see the `PushLeft, PopRight` example
- some statements might modify variables that are never read by other threads → no lock required
- statements in one `atomic` block might access variables in a different order to another `atomic` block → deadlock possible with locks implementation
- creating too many locks can decrease the performance, especially when required to release locks in λ(1) when acquiring!
ConcURRENCY across Languages

In most systems programming languages (C,C++) we have
- the ability to use atomic operations
- 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

<table>
<thead>
<tr>
<th>language</th>
<th>barriers</th>
<th>wait-/lock-free</th>
<th>semaphore</th>
<th>mutex</th>
<th>monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>C,C++</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(a)</td>
</tr>
<tr>
<td>Java,C#</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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 what they can do

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


Concurrency: Transactions

Dr. Michael Petter
Winter term 2015

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
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.

Consider an example:

- a linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`
- a set object may internally use the list object and expose a set of operations, including `PushLeft`

The `Insert` operation uses the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

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.

Consider an example:

- a linked list data structure exposes a fixed set of operations to modify the list structure, such as `PushLeft` and `ForAll`
- a set object may internally use the list object and expose a set of operations, including `PushLeft`

The `Insert` operation uses the `ForAll` operation to check if the element already exists and uses `PushLeft` if not.

Wrapping the linked list in a mutex does not help to make the `set` thread-safe.

```c
// wrap the two calls in Insert in a mutex
// but other list operations can still be called // use the same mutex
```

Transactional Memory [2]

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

```c
atomic {
    // code
    if (cond) retry;
    atomic {
        // more code
    }
}
// code
```
Transactional Memory [2]

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

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

Execute code as transaction:
- execute the code of an atomic block
- nested atomic blocks act like a single atomic block
- check that it runs without conflicts due to accesses from another thread
- if another thread interferes through conflicting updates:
  - undo the computation done so far
  - re-start the transaction
- provide a retry keyword similar to the retry of monitors

Managing Conflicts

Definition (Conflicts)

A conflict occurs when accessing the same piece of data, a conflict is detected when the TM system observes this, it is resolved when the TM system takes action (by delaying or aborting a transaction).

Design choices for transactional memory implementations:
- optimistic vs. pessimistic concurrency control:
  - pessimistic: detection/resolution when the conflict is about to occur
  - resolution here is usually delaying one transaction
  - can be implemented using locks: deadlock problem
  - optimistic: detection and resolution happen after a conflict occurs
  - resolution here must be aborting one transaction
  - need to repeat aborted transaction: livelock problem
- eager vs. lazy version management: how read and written data are managed during the transaction
  - eager: writes modify the memory and an undo-log is necessary if the transaction aborts
  - lazy: writes are stored in a redo-log and modifications are done on committing