Outline of this lecture:

1. Formal problem definitions
2. Solution for 2 threads
3. Solution for \( n \) threads
4. Inherent costs of mutual exclusion

1 **Formal problem definitions**

In this lecture, we study some simple, classical mutual exclusion algorithms for coordinating concurrent threads. Even though these algorithms are unlikely to be used in practice, they are worth studying because they help developing our understanding of the correctness issues of synchronization. This knowledge are useful when we reason about more advanced and practical mutual exclusion algorithms.

**Time notion**

In concurrent computation, there is really not a sense of time (i.e., a global clock) that all threads in the computation can read and use to relate to each other. Instead, we will think of the world in terms of the ordering among events, and will use time just as a tool for explaining this ordering. In other words, there is a notion of time but it is local and not global.

A thread is a a state machine, and its state transitions are called events. Examples of events are assignment of a shared variable, assignment of a local variable, invocation of a method, return from a method, and so on.

- Events are instantaneous
- Events are never simultaneous

A thread contains a sequence of events \( a_0, a_1, \ldots \). We denote the \( j \)-th occurrence of an event \( a_i \) by \( a_i^j \). We denote an event \( a \) precedes another event \( b \) by \( a \rightarrow b \), which means \( a \) occurs before \( b \). For 2 events \( a_0 \) and \( a_1 \), such that \( a_0 \rightarrow a_1 \), an interval \( (a_0, a_1) \) is the duration between \( a_0 \) and \( a_1 \). Intervals may overlap:
Interval $A_0 = (a_0, a_1)$ precedes interval $B_0 = (b_0, b_1)$, denoted by $A_0 \rightarrow B_0$ if $a_1 \rightarrow b_0$. There are some properties of events ordering:

- Irreflexive: never true that $A \rightarrow A$
- Anti-symmetric: if $A \rightarrow B$ then not true that $B \rightarrow A$
- Transitive: if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$
- But $A \rightarrow B$ and $B \rightarrow A$ might both be false!

**Mutual exclusion and critical section**

- Critical section: a block of code that access shared modifiable data or resource that should be operated on by only one thread at a time.
- Mutual exclusion: a property that ensures that a critical section is only executed by a thread at a time.

We consider the following implementation of Counter class, which is correct in a single-threaded execution, but incorrect in multi-threaded execution.
We can enforce mutual exclusion for a *Counter* object with a *Lock* object satisfies the following interface:

```java
public interface Lock {
    public void lock();
    public void unlock();
}
```

Then the updated code for *Counter* class using *Lock* object is:

```java
public class Counter {
    private long value;
    private Lock lock;
    public long getAndIncrement() {
        lock.lock();
        try {
            int temp = value;
            value = value + 1;
        } finally {
            lock.unlock();
        }
        return temp;
    }
}
```

Before accessing the critical section coded inside *try*{} block, thread must acquire lock first. The code for unlock part is put in *finally*{} to ensure the lock is released even if an exception is thrown in critical section. Let $CS_i^k$ denotes thread $i$’s $k$-th critical section execution and $CS_j^m$ denotes thread $j$’s $m$-th critical section execution, then mutual exclusion property guarantees that either $CS_i^k \rightarrow CS_j^m$ or $CS_j^m \rightarrow CS_i^k$.

Regarding to locking algorithms, a good algorithm should have the following properties:

- **Mutual exclusion**: as discussed. This is a safety guarantee.
• **Deadlock freedom**: this guarantees that system as a whole makes progress. (i.e. If some thread calls \( lock() \) and never acquires the lock, then other threads must complete \( lock() \) and \( unlock() \) calls infinitely often.)

• **Starvation freedom**: this guarantees that individual thread makes progress. (i.e. If some thread calls \( lock() \), it will eventually return). Thus, it implies deadlock freedom.

Having basic definitions and notions, we next look at algorithms for synchronizing 2 threads.

## 2 Solution for 2 threads

In this section, we consider 2 implementations of the \( Lock \) interface.

**LockOne algorithm**

The algorithm for \( LockOne \) class is as follows:

```java
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];
    // thread-local index, 0 or 1
    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {} // wait
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

In this algorithm, threads are indexed as 0 or 1 and its ID is obtained by calling \( ThreadID.get() \). We will prove that \( LockOne \) satisfies mutual exclusion property. Before going to the proof, we use \( write_A(x = \nu) \) to denote the event in which thread A assigns value \( \nu \) to variable \( x \), and \( read_A(\nu == x) \) to denote the event in which thread A reads \( \nu \) from variable \( x \).

**Lemma 1** The \( LockOne \) algorithm satisfies mutual exclusion property.

**Proof.** By contradiction. Suppose not, then there exists 2 critical sections, \( CS_A^j \) and \( CS_B^k \), of thread A and B respectively, such that \( CS_A^j \rightarrow CS_B^k \) and \( CS_B^k \rightarrow CS_A^j \). For each thread, we consider the last execution of the \( lock() \) method before it enters its \( j \)-th (\( k \)-th) critical section.
From the code, the order of events is:

\[
\begin{align*}
&\text{write}_A(flag[A] = \text{true}) \rightarrow \text{read}_A(flag[B] == \text{false}) \rightarrow CS_A \\
&\text{write}_B(flag[B] = \text{true}) \rightarrow \text{read}_B(flag[A] == \text{false}) \rightarrow CS_B
\end{align*}
\]

Since once flag[B] is set to true it remains true (until thread B call unlock()),
\[
\text{read}_A(flag[B] == \text{false}) \rightarrow \text{write}_B(flag[B] = \text{true}).
\]

Thus, it follows that:
\[
\begin{align*}
&\text{write}_A(flag[A] = \text{true}) \rightarrow \text{read}_A(flag[B] == \text{false}) \rightarrow \\
&\text{write}_B(flag[B] = \text{true}) \rightarrow \text{read}_B(flag[A] == \text{false}).
\end{align*}
\]

Hence, write\(_A(flag[A] = \text{true}) \rightarrow \text{read}_B(flag[A] == \text{false})\), which is impossible without a write to flag[A] in between those 2 events. Contradiction.

The LockOne algorithm can still cause deadlock if both events write\(_A(flag[A] = \text{true})\) and write\(_B(flag[B] = \text{true})\) happen before while statements. In this case, both thread wait forever.

**LockTwo algorithm**

The LockTwo algorithm is shown below:

```java
class LockTwo implements Lock {
    private int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i; // let the other go first
        while (victim == i) {}; // wait
    }

    public void unlock() {}
}
```

Similar to LockOne algorithm, LockTwo algorithm also satisfies mutual exclusion. We can prove this claim in a similar manner to the proof of LockOne.

**Lemma 2** The LockTwo algorithm satisfies mutual exclusion.

**Proof.** Again we prove by contradiction. Suppose not, then there exists 2 critical sections, \(CS_A^j\) and \(CS_B^k\), of thread A and B respectively, such that \(CS_A^j \rightarrow CS_B^k\) and \(CS_B^k \rightarrow CS_A^j\). For each thread, we consider the last execution of the lock() method before it enters its \(j\)-th (\(k\)-th) critical section.

From the code, the order of events is:

\[
\begin{align*}
&\text{write}_A(victim = A) \rightarrow \text{read}_A(victim == B) \rightarrow CS_A \\
&\text{write}_B(victim = B) \rightarrow \text{read}_B(victim == A) \rightarrow CS_B
\end{align*}
\]
Since thread A reads B from victim (in the read event of thread’s event sequence), it follows that thread B writes B to victim before thread A reads from victim. Thus, write\(_A(\text{victim} = A) \rightarrow \text{write}_B(\text{victim} == B) \rightarrow \text{read}_A(\text{victim} == B)\). After that victim == B, and event \text{read}_B(\text{victim} == A) of thread B cannot occur. Contradiction.

Similar to the LockOne algorithm, the LockTwo algorithm can also suffer from deadlock.

**Peterson’s algorithm**

Peterson’s algorithm combines ideas from LockOne and LockTwo algorithm to satisfy both mutual exclusion and starvation-free requirements (thus, also satisfy deadlock-free requirement). The algorithm is as follows:

```java
class Peterson implements Lock {
    private boolean[] flag = new boolean[2];
    private int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true; // I’m interested
        victim = i; // you go first
        while (flag[j] && victim == i) {};
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false; // I’m not interested
    }
}
```

**Lemma 3** The Peterson algorithm satisfies mutual exclusion.

**Proof.** By contradiction. Suppose not, we consider the executions of lock() by 2 threads, A and B. From the code, the order of events is:

\[
\begin{align*}
\text{write}_A(\text{flag}[A] = \text{true}) & \rightarrow \text{write}_A(\text{victim} = A) \rightarrow \text{read}_A(\text{flag}[B]) \rightarrow \text{read}_A(\text{victim}) \rightarrow CS_A \\
\text{write}_B(\text{flag}[B] = \text{true}) & \rightarrow \text{write}_B(\text{victim} = B) \rightarrow \text{read}_B(\text{flag}[A]) \rightarrow \text{read}_B(\text{victim}) \rightarrow CS_B
\end{align*}
\]

Without loss of generality, assume that A was the last thread to write to victim.

\[
\text{write}_B(\text{victim} = B) \rightarrow \text{write}_A(\text{victim} == A)
\]

Thus, thread A read victim == A in its sequence of events. Since thread A still entered its critical section, it must be true that thread A reads \text{flag}[B] == false. Thus it follows that,

\[
\begin{align*}
\text{write}_B(\text{flag}[B] = \text{true}) & \rightarrow \text{write}_B(\text{victim} = B) \rightarrow \text{write}_A(\text{victim} == A) \rightarrow \text{read}_A(\text{flag}[B] == \text{false})
\end{align*}
\]

This is impossible because there is no other write false to flag[B] between write\(_B(\text{flag}[B] = \text{true}) and read\(_A(\text{flag}[B] == \text{false}). Contradiction. \square
Lemma 4  The Peterson algorithm is starvation-free.

Proof.  By contradiction. Without loss of generality, suppose thread $A$ waits forever in the lock() method. Particularly, it runs while() forever waiting for either flag[$B$] becomes false or victim becomes $B$. If thread $B$ also got stuck in its while() loop, then it must read $B$ from victim. But since victim cannot be both $A$ and $B$, the hypothesis that thread $B$ also got stuck in its lock() method is impossible. Thus, thread $B$ must be able to enter its critical section. Then after thread $B$ finishes its critical section and calls unlock() method, flag[$B$] becomes false, and this triggers thread $A$ to enter its own critical section. Hence, thread $A$ must not wait forever at its while() loop.

Because starvation-free implies deadlock-free, it follows that Peterson’s algorithm is deadlock-free.

Corollary 5  The Peterson’s algorithm is deadlock-free.

3  Solution for $n$ threads

In this section, we discuss a solution for synchronizing $n$ threads, named Lamport’s Bakery algorithm. The algorithm is shown in the figure below:

```java
class Bakery implements Lock {
    boolean[] flag;
    Label[] label;
    public Bakery(int n) {
        flag = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
            flag[i] = false; label[i] = 0;
        }
    }
    public void lock() {
        int i = ThreadID.get();
        flag[i] = true;
        label[i] = max(label[0], ..., label[n-1]) + 1;
        while((3k != i) (flag[k] && (label[i],i) > (label[k],k));
    }
    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

In this algorithm, a flag[$A$] indicates whether thread $A$ wants to enter its critical section, and label[$A$] is a number indicating the order in which thread $A$ is going to enter its critical section, the smaller the number is, the sooner it enters its critical section. When a thread calls lock(), it updates its label to a value of 1 larger than the current maximum among all labels. There is a
chance that 2 threads read the same maximum label at the same time, and thus update their label to the same value. To break tie in this case, the while() loop in the lock() method orders threads using pairs \((label[i], i)\); in which \((label[i], i) > (label[k], k)\) if and only if \(label[i] > label[k]\) or \(label[i] == label[k]\) and \(i > k\).

In thread \(i\)'s while() loop, it waits until there is no other thread with raised flag and smaller value of its pair \((label[k], k)\).

**Lemma 6** The Bakery algorithm satisfies mutual exclusion.

*Proof.* Suppose not. Let \(A\) and \(B\) be two threads which have overlapping critical sections. Let \(labeling_A\) and \(labeling_B\) be the labels of \(A\) and \(B\) right before they enter the critical sections. Without loss of generality, suppose \((label[A], A) < (label[B], B)\). Since labels of \(A\) and \(B\) do not change until the next call to the lock() method, when thread \(B\) successfully passed its test in the while() loop, it must have read \(flag[A] == false\). This means thread \(B\) must have read \(flag[A] == false\) before thread \(A\) updated \(flag[A]\) to true. Thus, the sequence of events is:
\[
labeling_B \rightarrow \text{read}_B(flag[A] == false) \rightarrow \text{write}_A(flag[A] = true) \rightarrow labeling_A.
\]

It means that \(label[B] < label[A]\), which contradicts to the hypothesis. Contradiction.

**Lemma 7** The Bakery algorithm is deadlock-free.

*Proof.* There always exist some thread \(A\) with smallest pair \((label[A], A)\), and it never has to wait for any other thread.

**Lemma 8** The Bakery algorithm is first-come-first-served.

*Proof.* Suppose thread \(A\)'s doorway precedes thread \(B\)'s doorway, \(D_A \rightarrow D_B\). Then \(label[A]\) must be smaller than \(label[B]\). Thus, thread \(B\) must have been locked while thread \(A\)'s \(flag[A] == true\).

Since Lamport's Bakery algorithm is both deadlock-free and first-come-first-served, it is also starvation-free. One concern with this algorithm is that the label can overflow. However, if we use 64-bit integer for labeling, it will be unlikely to happen.

## 4 Inherent costs of mutual exclusion

Although the Lamport's Bakery algorithm is succinct, elegant, and fair, it is not used in practice because of its inherent cost: it has to read \(n\) labels in the lock() method.

Shared read/write memory locations are called Registers, and can be classified as:

- Multi-Reader Single-Writer (MRSW) (as \(flag[]\) in the Bakery algorithm)
- Multi-Reader Multi-Writer (MRMW) (as \(victim\) in the Peterson’s algorithm)
- The others: SRMW and SRSW are not of interested to us
There is a theorem states that: "At least $n$ MRSW registers are needed to solve deadlock-free mutual exclusion". The Bakery algorithm uses $2n$ such MRSW registers, thus its bound is pretty tight. There is also another theorem states that: "At least $n$ MRMW registers are needed to solve deadlock-free mutual exclusion". This means multiple-writer registers don’t help to reduce the bound. Hence, because of the expensive costs inherently associate with the software solutions to the mutual exclusion problem, we need stronger hardware primitives for this problem.