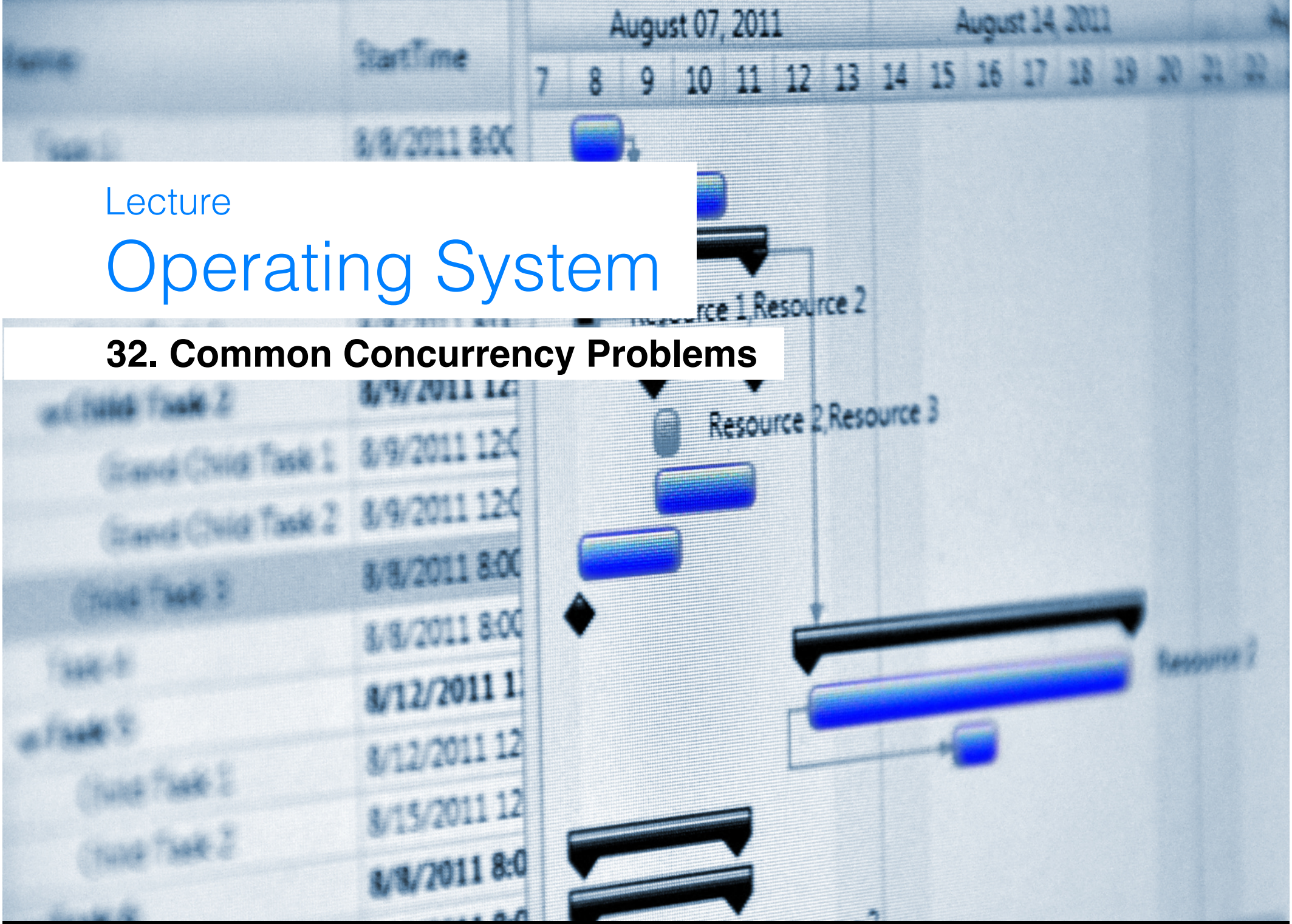
Lecture

Operating System

32. Common Concurrency Problems



Professor Dr. Michael Mächtel

32. Common Concurrency Problems

1. **Common Concurrency Bugs**
2. **Why Deadlocks occur**
3. **Deadlock Prevention**
4. **Deadlock Avoidance**

32. Common Concurrency Problems

1. **Common Concurrency Bugs**
2. Why Deadlocks occur
3. Deadlock Prevention
4. Deadlock Avoidance

Concurrency in Medicine:

Therac-25 (1980's)

"The accidents occurred when the high-power electron beam was activated instead of the intended low power beam, and without the beam spreader plate rotated into place. Previous models had hardware interlocks in place to prevent this, but Therac-25 had removed them, depending instead on software interlocks for safety. The software interlock could fail due to a **race condition**."

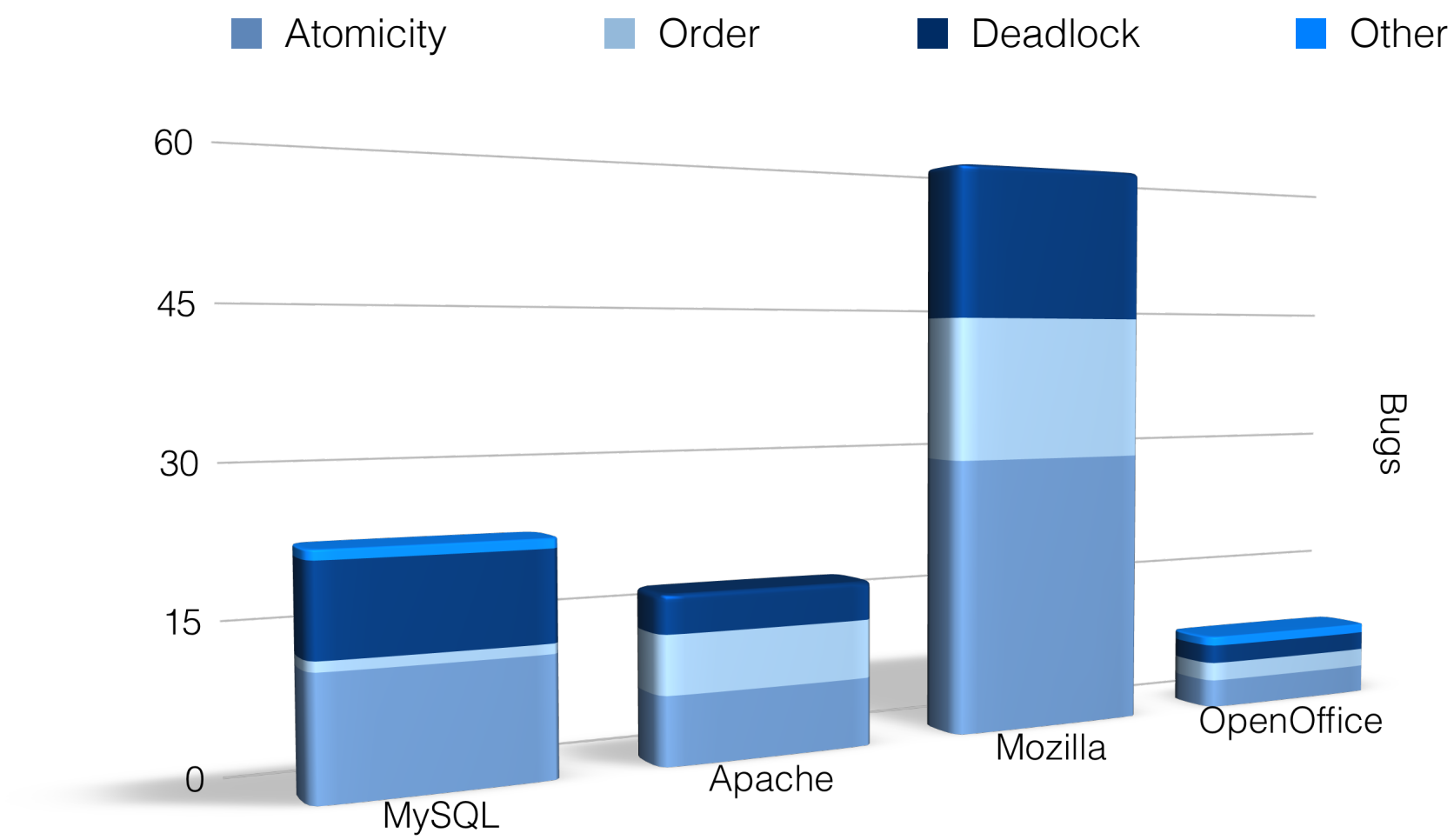
"…in three cases, the injured patients **later died**."

http://en.wikipedia.org/wiki/Therac-25

Common Concurrency Problems

- Concurrency bugs are indeterministic
- Difficult to debug
- Called "*Heisenbug*":
  - “A bug that seems to disappear when one attempts to study it”
- More recent work focuses on studying other types of common concurrency bugs.
  - Take a brief look at some example concurrency problems found in real code bases.
- What Types Of Bugs Exist?
  - Focus on four major open-source applications
    - MySQL, Apache, Mozilla, OpenOffice.

Concurrency Study from 2008



Non-Deadlock Bugs

- Make up a majority of concurrency bugs.
- Two major types of non deadlock bugs:
  - Atomicity violation
  - Order violation

Atomicity-Violation Bugs

- The desired serializability among multiple memory accesses is violated.
  - Simple Example found in MySQL:
    - Two different threads access the field `proc_info` in the struct `thd`.

```
Thread1 ::
if(thd→proc_info){
  …
  fputs(thd→proc_info , …);
  …
}

Thread2 ::
thd→proc_info = NULL;
```

Atomicity-Violation Bugs

- **Solution**: Simply add locks around the shared-variable references.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

 Thread1 ::
 pthread_mutex_lock(&lock);
 if(thd→proc_info){
    …
    fputs(thd→proc_info , …);
    …
 }
 pthread_mutex_unlock(&lock);

 Thread2 ::
 pthread_mutex_lock(&lock);
 thd→proc_info = NULL;
 pthread_mutex_unlock(&lock);
```

Order-Violation Bugs

- The desired order between two memory accesses is flipped.
  - i.e., A should always be executed before B, but the order is not enforced during execution.
  - Example:
    - The code in Thread2 seems to assume that the variable `mThread` has already been *initialized* (and is not `NULL`).

```
Thread1::
void init(){
    mThread = PR_CreateThread(mMain, …);
}

Thread2::
void mMain(…){
    mState = mThread→State
}
```

Order-Violation Bugs

- **Solution:** Enforce ordering using condition variables

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0;
```

Thread1 ::

```
void init(){
 …
 mThread = PR_CreateThread(mMain,…);
 // signal that the thread has
 // been created.
 pthread_mutex_lock(&mtLock);
 mtInit = 1;
 pthread_cond_signal(&mtCond);
 pthread_mutex_unlock(&mtLock);
 …
}
```

Thread2 ::

```
void mMain(…){
 …
 // wait for the thread to be
 //initialized …
 pthread_mutex_lock(&mtLock);
 while(mtInit == 0)
   pthread_cond_wait(&mtCond, &mtLock);
 pthread_mutex_unlock(&mtLock);

 mState = mThread→State;
 …
}
```

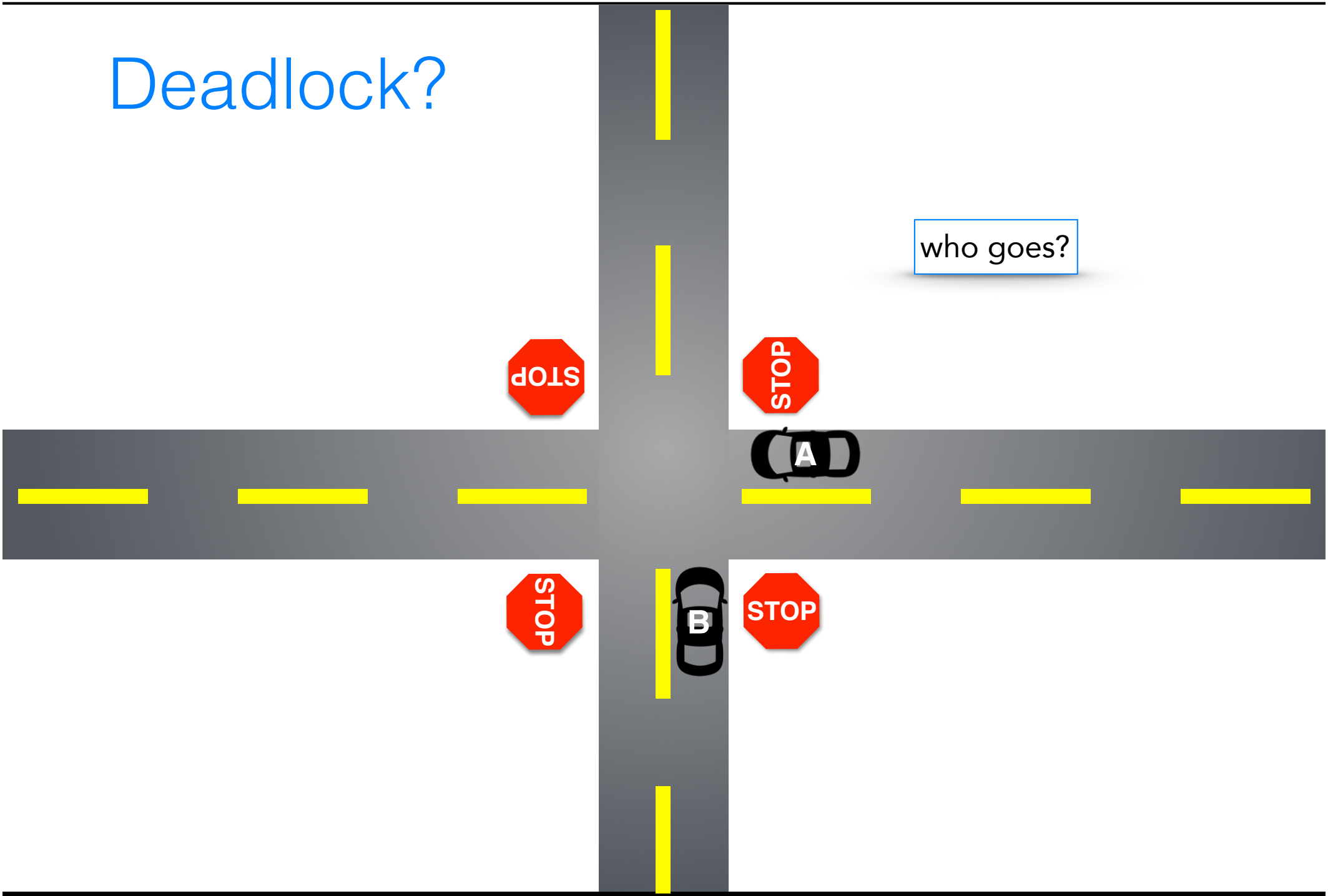
32. Common Concurrency Problems

1. Common Concurrency Bugs
2. **Why Deadlocks occur**
3. Deadlock Prevention
4. Deadlock Avoidance

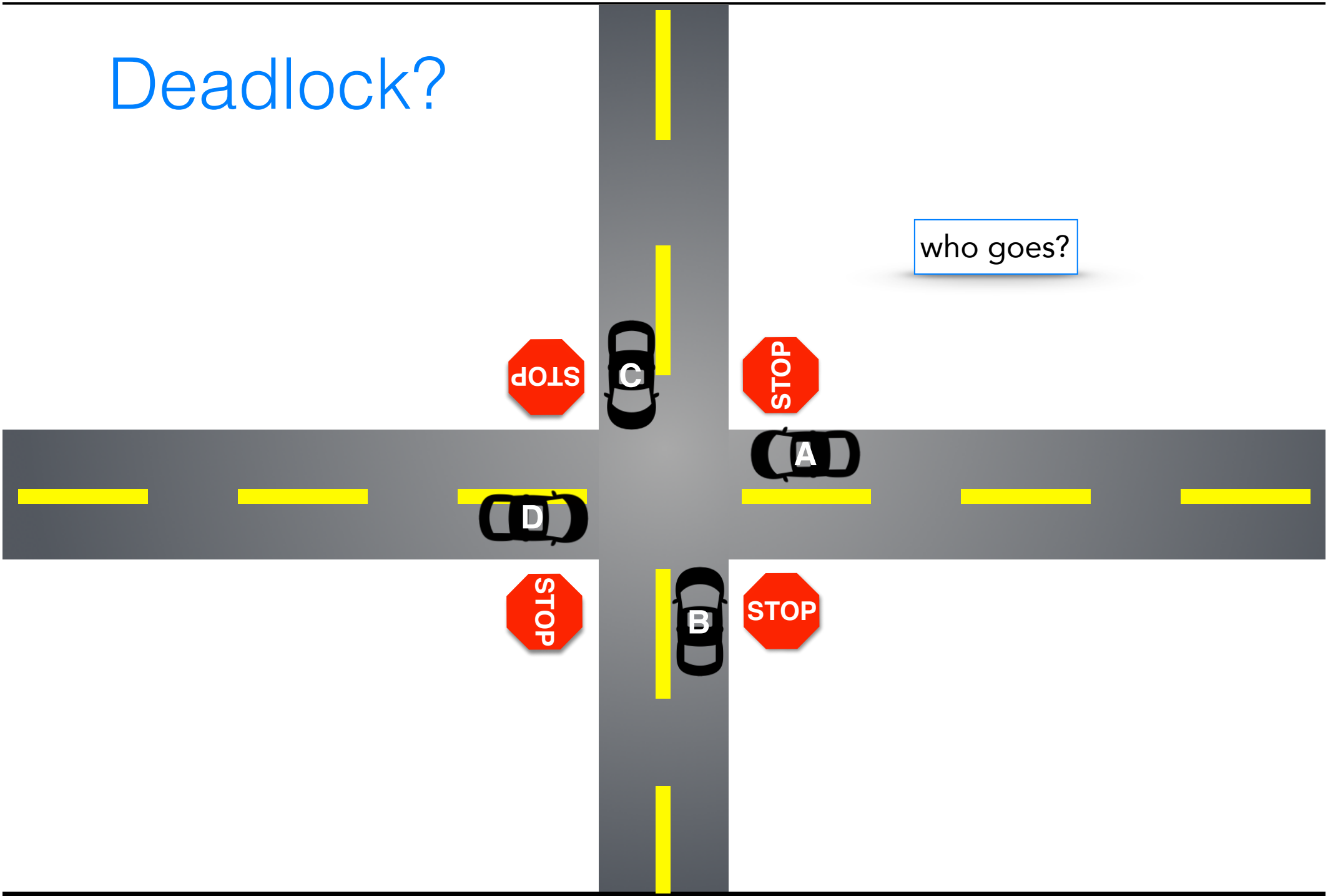
Deadlock

- Deadlock: No progress can be made because two or more threads are waiting for the other to take some action and thus neither ever does
- “Cooler" name: the deadly embrace (Dijkstra)
  - https://de.wikipedia.org/wiki/Edsger_W._Dijkstra

Deadlock?



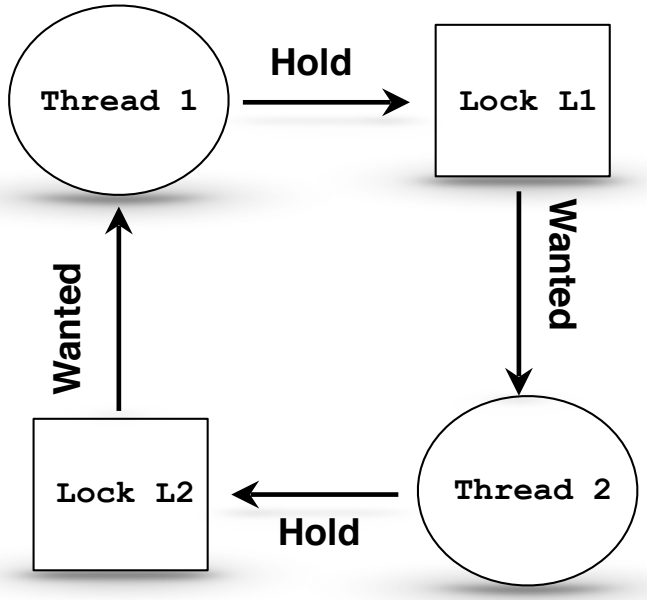
Deadlock?



Deadlock Bugs

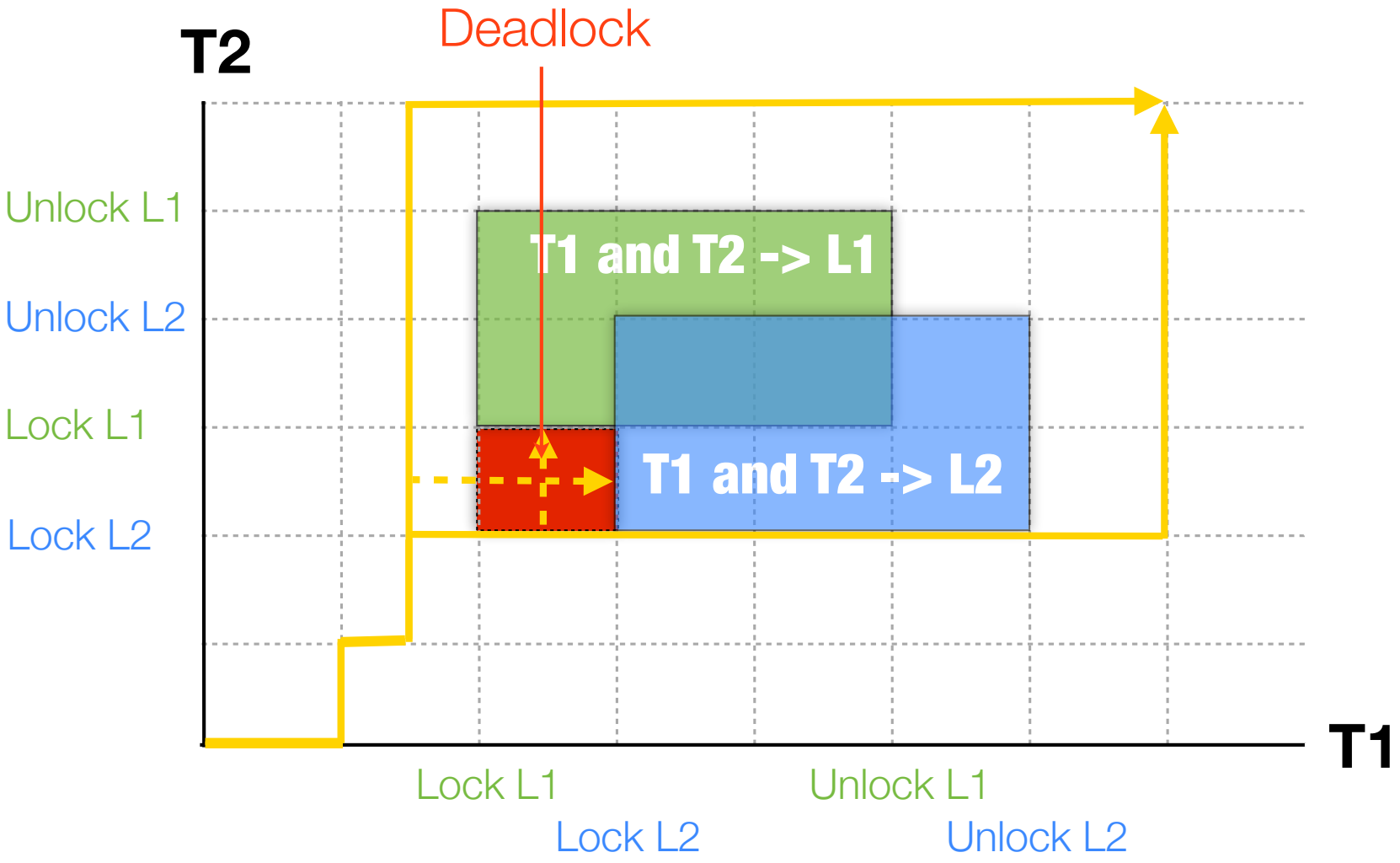
- The presence of a cycle
  - *Thread1* is holding a lock L1 and waiting for another one, L2.
  - *Thread2* that holds lock L2 is waiting for L1 to be release.

```
Thread 1:    Thread 2:
lock(L1);    lock(L2);
lock(L2);    lock(L1);
```



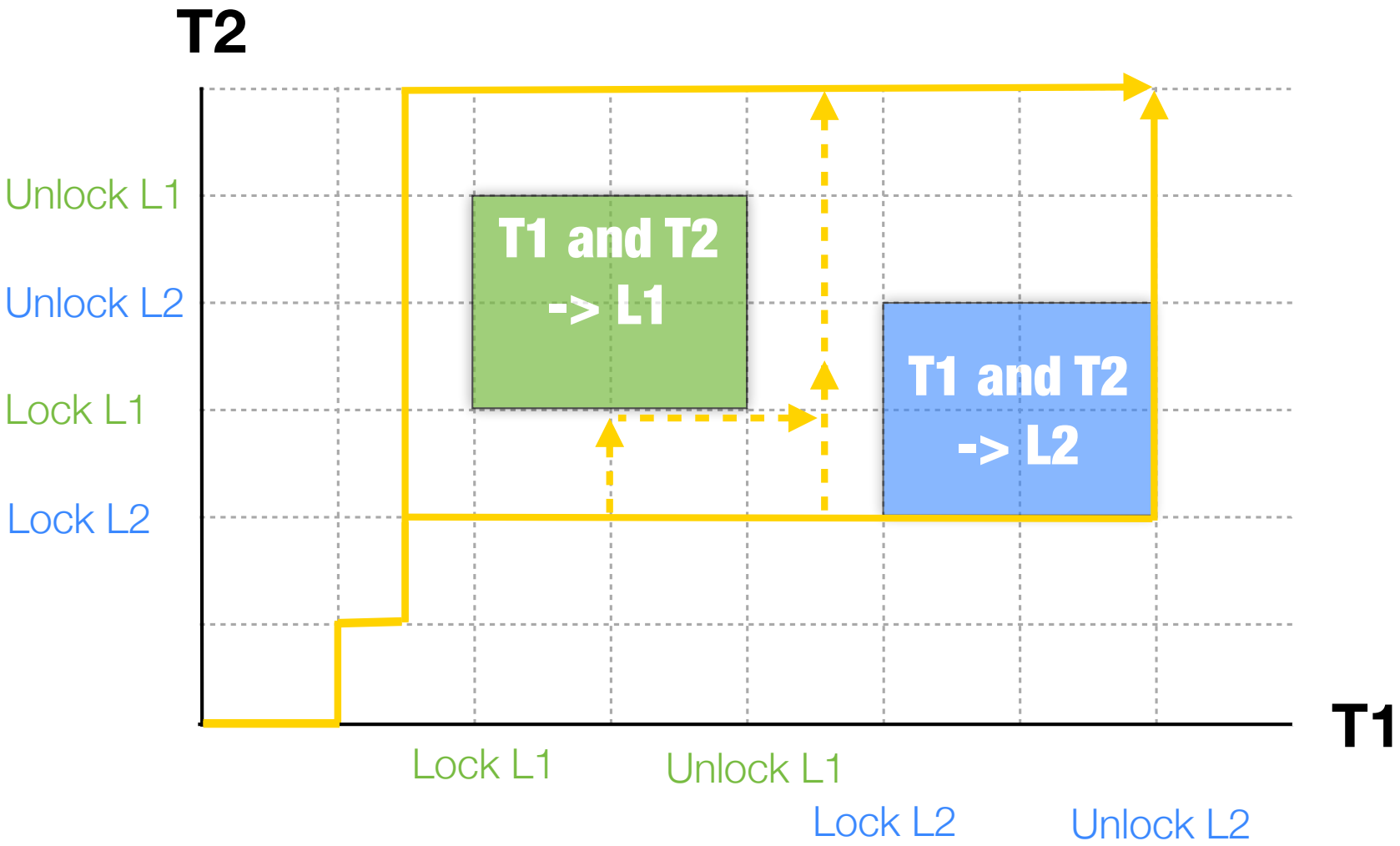
Example

2 tasks and 2 resources



Example

2 tasks und 2 resources (task T1 changed)



Why Do Deadlocks Occur?

- **Reason 1:**
  - In large code bases, complex dependencies arise between components.
- **Reason 2:**
  - Due to the nature of encapsulation
  - Hide details of implementations and make software easier to build in a modular way.
  - Such modularity does not mesh well with locking.

Why Do Deadlocks Occur? (Cont.)

- Example: Java Vector class and the method AddAll()

```
Vector v1,v2;
v1.AddAll(v2);
```

- Locks for both the vector being added to (v1) and the parameter (v2) need to be acquired.
  - The routine acquires said locks in some arbitrary order (v1 then v2).
  - If some other thread calls `v2.AddAll(v1)` at nearly the same time:
    - We have the potential for **deadlock**.

32. Common Concurrency Problems

1. Common Concurrency Bugs
2. Why Deadlocks occur
3. **Deadlock Prevention**
4. Deadlock Avoidance

Conditional for Deadlock

- Four conditions need to hold for a deadlock to occur.
  - If any of these four conditions are not met, **deadlock cannot occur.**

Condition	Description
Mutual Exclusion	Threads claim exclusive control of resources that they require.
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources
No preemption	Resources cannot be forcibly removed from threads that are holding them.
Circular wait	There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain

Prevention – Circular Wait

- Provide **a total ordering** on lock acquisition
  - This approach requires careful design of global locking strategies.
- **Example**:
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2.

Prevention – Hold-and-wait

- Acquire all locks **at once, atomically**.

```
lock(prevention);
lock(L1);
lock(L2);
…
unlock(prevention);
```

- This code guarantees that **no untimely thread switch can occur** *in the midst of* lock acquisition.
- **Problem**:
  - Require us to **know** when calling a routine **exactly** which locks must be held and to acquire them **ahead of time**.
  - Decrease *concurrency*

Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock **we are holding another**.
- `trylock()`
  - Used to build a *deadlock-free, ordering-robust* lock acquisition protocol.
  - Grab the lock (if it is available).
  - Or, return -1: you should try again later.

```
top:
   lock(L1);
   if( tryLock(L2) == -1 ){
       unlock(L1);
       goto top;
   }
```

Prevention – No Preemption (Cont.)

- **New problem: livelock**
  - Both systems are running through the code sequence over and over again.
  - Progress is not being made.
  - Solution:
    - Add **a random delay** before looping back and trying the entire thing over again.

Prevention – Mutual Exclusion

- **wait-free**
  - Using powerful hardware instruction.
  - You can build data structures in a manner that *does not require* explicit locking.

```
int CompareAndSwap(int *address, int expected, int new){
    if(*address == expected){
        *address = new;
        return 1; // success
    }
    return 0;
}
```

Prevention – Mutual Exclusion (Cont.)

- We now wanted to atomically increment a value by a certain amount:

```
void AtomicIncrement(int *value, int amount){
    do{
        int old = *value;
    }while( CompareAndSwap(value, old, old+amount)==0);
 }
```

- Repeatedly tries to update the value *to the new amount* and uses the `compare-and-swap` to do so.
  - **No lock** is acquired
  - **No deadlock** can arise
  - **livelock** is still a possibility.

Example: List Insertion

- More complex example: list insertion
  - If called by multiple threads at the "*same time*", this code has a race condition.

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n→value    = value ;
    n→next     = head;
    head       = n;
}
```

Solution for List Example

- Surrounding this code with a **lock acquire** and **release**.

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value = value ;
    lock(listlock); // begin critical section
    n->next = head;
    head    = n;
    unlock(listlock) ;  //end critical section
 }
```

- **wait-free manner** using the `compare-and-swap` instruction

```
 void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n));
 }
```

32. Common Concurrency Problems

1. Common Concurrency Bugs
2. Why Deadlocks occur
3. Deadlock Prevention
4. **Deadlock Avoidance**

Deadlock Avoidance via Scheduling

- In some scenarios **deadlock avoidance** is preferable.
- **Global knowledge** is required:
  - **Which locks** various threads might grab during their execution.
  - Subsequently **schedules** said threads **in a way** as to guarantee **no deadlock can occur**.

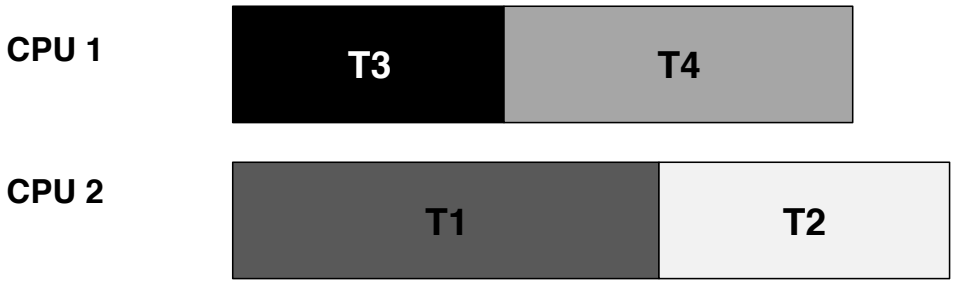
Example: Deadlock Avoidance

via Scheduling

- We have two processors and four threads.
- Lock acquisition demands of the threads:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise.



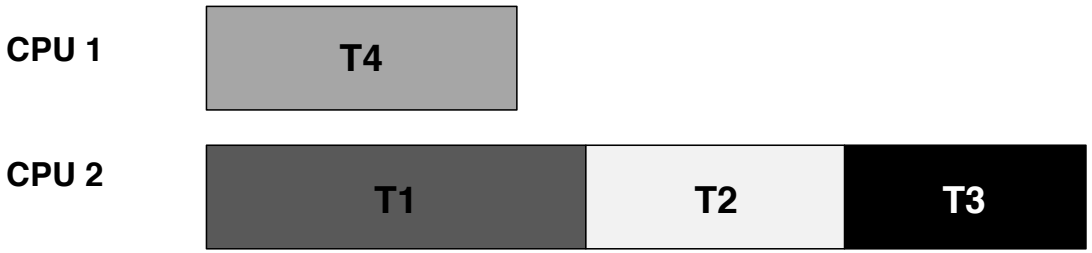
Example: Deadlock Avoidance

via Scheduling

- More contention for the same resources

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

- A possible schedule that guarantees that no deadlock could ever occur.



  - The total time to complete the jobs is lengthened considerably.

Resource Allocation Denial

- A strategy of resource allocation denial is known as banker's algorithm.
- If a resource is going to be claimed:
  - the system is checked, if by this claim a deadlock is possible
- If no deadlock is possible,
  - the system is **safe**, and the resource is allocated
- I a deadlock might be possible
  - the system is **unsafe**, and the allocation is paused

Resource Allocation Denial (Cont.)

Necessary quantities for deadlock avoidance policy

- $(R_1,R_2,...R_m)$ **Total amount of each resource in the system**
- $(V_1,V_2,...V_m)$ **Total amount of each resource not allocated to any process**
- $C_{11},C_{12},...C_{1m}$
  $C_{21},C_{22},...C_{2m}$
  $C_{n1},C_{n2},...C_{nm}$
  C_{ij}: Requirement of process i for resource j
- $A_{11},A_{12},...A_{1m}$
  $A_{21},A_{22},...A_{2m}$
  $A_{n1},A_{n2},...A_{nm}$
  A_{ij}: Current Allocation to process i of res. j

$$R = V + \sum A$$

$$C_i \leq R_i$$

$$A_i \leq C_i$$

Example Bankier's Algorithm

Is System safe or unsafe?

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

Available Vector

P2 -> Exit:

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
6	2	3

Available Vector

Example Bankier's Algorithm

Is System safe or unsafe?

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
6	2	3

Available Vector

P1 -> Exit:

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
7	2	3

Available Vector

Example Bankier's Algorithm

Is System safe or unsafe?

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
7	2	3

Available Vector

P3 -> Exit:

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	4

Available Vector

Example Bankier's Algorithm

Is System safe or unsafe?

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	4

Available Vector

P4 -> Exit:

All processes have been run to completion ➔ Safe State!

Restriction to Bankier's Algorithm

- The maximum resource requirement for each process must be started in advance
- The processes under consideration must be independent: that is, the order in which they execute must be unconstrained by any synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Just Detect and Recover Deadlock?

- **Allow deadlock** to occasionally occur and then take some action.
  - **Example**: if an OS froze, you would **reboot it**.
- Many database systems employ *deadlock detection* and *recovery technique*.
  - A deadlock detector **runs periodically**.
  - Building a **resource graph** and checking it for cycles.
  - In deadlock, the system **need to be restarted**.

Lock Ordering in Linux

In linux-3.2.51/include/linux/fs.h

```
/* inode→i_mutex nesting subclasses for the lock
 * validator:
 * 0: the object of the current VFS operation
 * 1: parent
 * 2: child/target
 * 3: quota file
 * The locking order between these classes is
 * parent → child → normal → xattr → quota
 */
```

Summary

- When in doubt about correctness, better to limit concurrency (i.e., add unneccessary lock)
- Concurrency is hard, encapsulation makes it harder!
- Have a strategy to avoid deadlock and stick to it
- Choosing a lock order is probably most practical

Thanks

Questions?