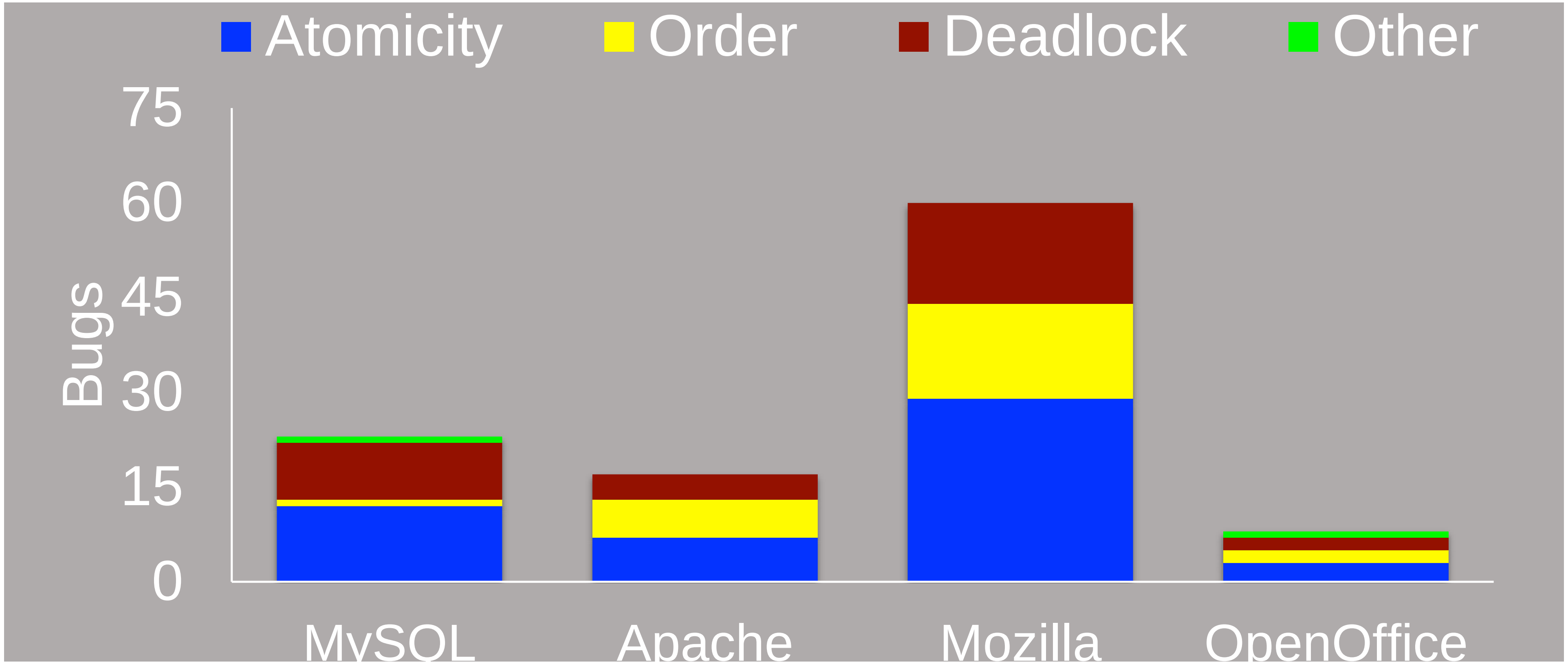
Concurrency: Common Bugs

Sridhar Alagar

Concurrency is hard



- Lu et al. study: Analyzed a sample of around 100 bugs from among > 500k bugs

Atomicity Violation

Thread 1:

```
if (thd->proc_info) {
    …
    fputs(thd->proc_info, …);
    …
}
```

Thread 2:

```
thd->proc_info = NULL;
```

- What is wrong?
  - Test (if()) and Set (fputs()) should be atomic

Atomicity Violation: Fix with locks

Thread 1:

```
pthread_mutex_lock(&lock);
if (thd->proc_info) {
   …
   fputs(thd->proc_info, …);
pthread_mutex_unlock(&lock);
  …
}
```

Thread 2:

```
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Order Violation

Thread 1:

```
mThread = CreateThread(…);
…
```

Thread 2:

```
state = mThread->state;
```

- What is wrong?
  - createThread() should be executed before thread access

Order Violation: Fix

Thread 1:

```
mThread = CreateThread(…);

sem_post(s);
   …
}
```

Thread 2:

```
sem_wait(s);

thd->proc_info = NULL;
```

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: DEADLY EMBRACE (Dijkstra)

Deadlock Bug

Thread 1:

```
lock(&L1);
lock(&L2);
```

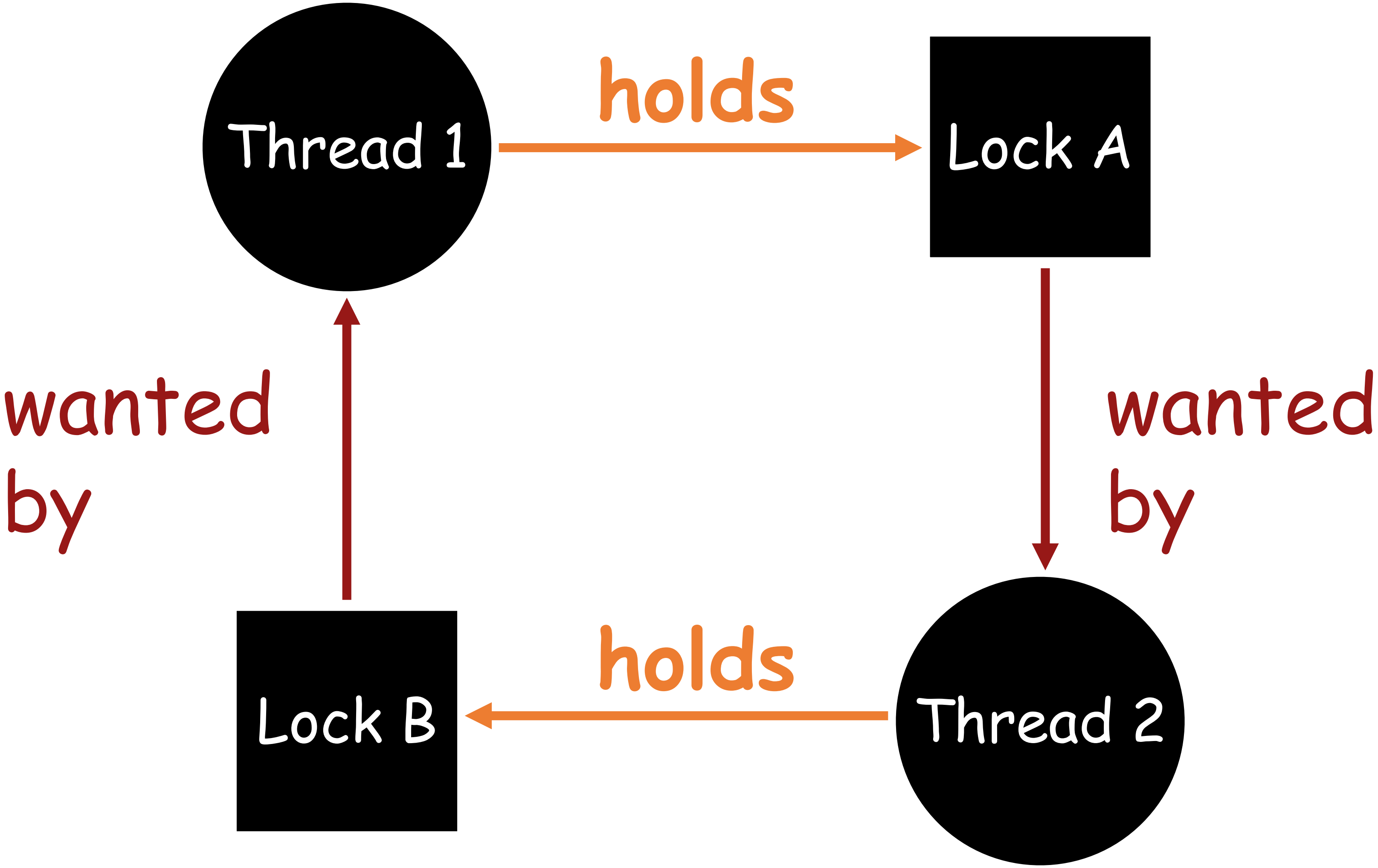
Thread 2:

```
lock(&L2);
lock(&L1);
```

Circular wait

Wait For Graph(WFG)

Cycle <=> Deadlock



Necessary Conditions for Deadlock

- Mutual Exclusion
  - Only one thread can use a resource at a time
- Hold and Wait
  - Threads holding resources waiting for resources held by other threads
- No Preemption
  - Resources cannot be forcibly removed from threads holding it
- 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

Deadlock Prevention

- Prevent deadlock by ensuring that one of the conditions does not hold

- Constrains how resources are acquired. Incurs extra cost:
  - overhead
  - reduction in concurrency

Prevention – Eliminate Circular Wait

- Provide a total ordering of lock/resource acquisition
  - Cannot acquire a resource that is earlier in the order than any of the resources currently held

Thread 1:

```
lock(&L1);
lock(&L2);
```

Thread 2:

```
lock(&L1);
lock(&L2);
```

Prevention – Eliminate Hold and Wait

- Acquire all locks at once, atomically

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

- Problems:
  - Need to know the locks that will be acquired in the future
  - Reduces concurrency

Prevention - Preempt

- If cannot get the lock needed, release the locks held

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

- Problems:
    - Livelock
    - use exponential back-off

Prevention – Wait Free Mutual Exclusion

- Using hardware atomic instruction build data structure that does not require explicit locking for update

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

Prevention – Wait Free Mutual Exclusion

- Want to atomically increment a the value of a variable by a certain amount

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

Prevention – Wait Free Mutual Exclusion

- Want to atomically increment a the value of a variable by a certain amount

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

- No lock acquired; so no deadlock

Prevention – Wait Free Mutual Exclusion

- A more complex example: list insertion

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

- If called by multiple threads concurrently, race condition can arise

Prevention - Wait Free Mutual Exclusion

- Solution using locks

```
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
}
```

Prevention – Wait Free Mutual Exclusion

- Solution in wait-free manner

```
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));
}
```

Deadlock Avoidance: via Scheduling

- Requires global knowledge about thread locks/resource usage

- Schedule threads in such a way deadlocks can be avoided

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

Deadlock Avoidance: via Scheduling

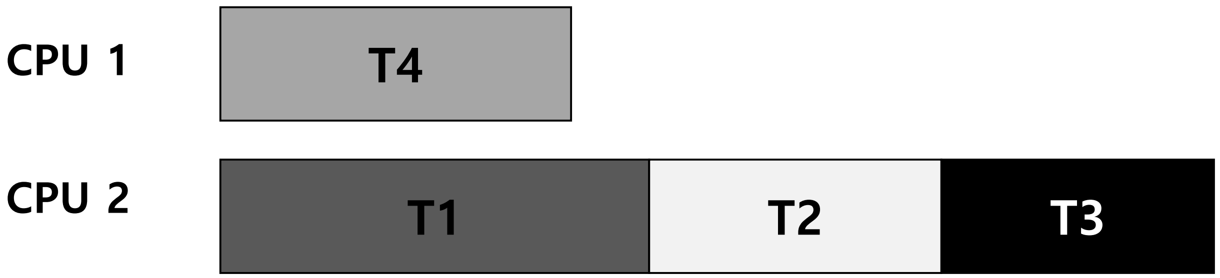
	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- A deadlock avoiding schedule

Deadlock Avoidance: via Scheduling

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

- A deadlock avoiding schedule



Detect and Recover

- 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
  - Build a resource graph and checking it for cycles
  - If in deadlock, restart the system

Disclaimer

- Some of the materials in this lecture slides are from the lecture slides by Prof. Andrea, Prof. Youjip, and other educators. Thanks to all of them.