

CS3.301 Operating Systems and Networks

Classical Concurrency Problems and Concurrency Bugs

Karthik Vaidhyanathan

<https://karthikvaidhyanathan.com>

1



Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

Sources:

- Operating Systems in three easy pieces by Remzi et al.



An Analogy

One Person writing



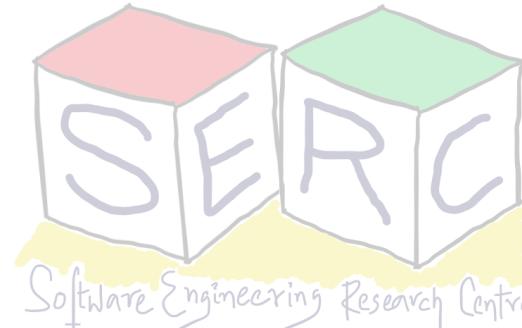
Many people reading at the same time



Online word processors

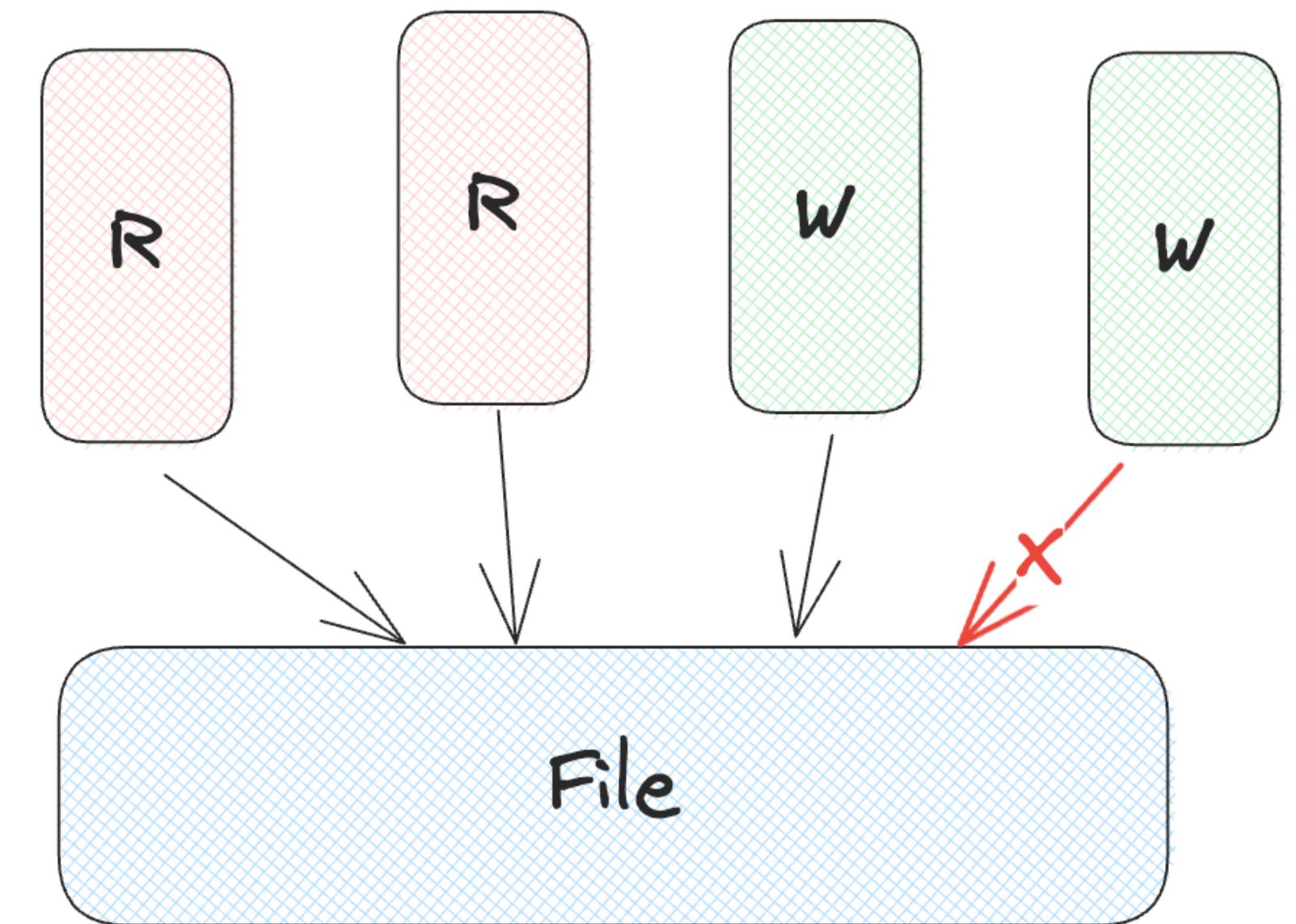


Databases



Readers/Writers Problem

- **Reader:** Process or thread that reads from memory
- **Writer:** Process or thread that writes on the memory
- Two readers can work on the same file at the same time
- Multiple writers cannot work on the same file at the same time



Readers/Writers Problem

Intuition

- **Only one writer can write at any point of time!**
- Reader thread can come in:
 - More readers come in, they can be allowed access
 - The moment writers come, it can be blocked
 - Once readers are done with reading, writers can start writing
- Can you think about writing a solution to this?
 - Do you foresee any challenges here?



Readers/Writers Problem



Readers/Writers Problem Solution

```
typedef struct _rwlock_t
{
    int readers;
    sem_t lock;
    sem_t writelock;
}rwlock_t;

void rwlock_init(rwlock_t *rw)
{
    rw -> readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}
```



Readers/writers Problem Solution



Readers/Writers Problem Solution

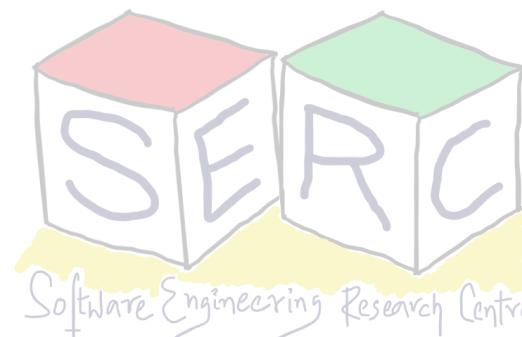
```
void acquire_readlock(rwlock_t *rw)
{
    sem_wait(&rw->lock);
    rw->readers++;
    if(rw->readers == 1)
        //disable writers to enter
        sem_wait(&rw->writelock);
    sem_post(&rw->lock);
}

void release_readlock(rwlock_t *rw)
{
    sem_wait(&rw->lock);
    rw->readers--;
    if(rw->readers == 0)
        //free the write lock
        sem_post(&rw->writelock);
    sem_post(&rw->lock);
}

void acquire_writelock(rwlock_t *rw)
{
    sem_wait(&rw->writelock)
}

void release_writelock(rwlock_t *rw)
{
    sem_post(&rw->writelock)
}
```

Writers Starve!!!



Readers/Writers Problem Solution

Add a lock that can act as priority common to both



```
sem_t serviceQueue;
void acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->serviceQueue);
    sem_post(&rw->serviceQueue);

    sem_wait(&rw->mutex);
    if (++rw->readers == 1)          // first reader blocks writers
        sem_wait(&rw->writelock);
    sem_post(&rw->mutex);
}
```

**serviceQueue is a semaphore
With an initial value of 1 and it acts like a
turnstile**

Reader



```
/* Writers (hold turnstile to block new arrivals) */
void acquire_writelock(rwlock_t *rw) {
    sem_wait(&rw->serviceQueue);    // get serviceQueue
    sem_wait(&rw->writelock);      // wait until no readers
}
```

The Dinning Philosophers

An Analogy



- Five philosophers sit around a dinning table
- They think for sometime and eat spaghetti for sometime!
- There is one fork on the left and one on the right of each
- If they get two forks, then they can start eating, once done, they can keep it down
- How to solve it?



Image source: Wikipedia

Classic Problem: Dining Philosophers

```
while (1)
{
    think();
    get_forks(p);
    eat();
    put_forks(p);
}
```

- Each philosopher is a unique thread with an id ($p = 0$ to 4);
- Get forks and put forks needs to be written by ensuring there is no deadlock
 - is there a **possibility of deadlock?** Why?
 - Also **no philosopher should starve!**
- Can you think of implementing `get_forks(p)` and `put_forks(p)`?



Possible Solution

All semaphores initiated to 1

● ● ● Dinning Philosophers Problem

```
int left(int p)
{
    return p;
}

int right (int p)
{
    return (p+1)%5; //consider the person on right
}
```

● ● ● Dinning Philosophers Problem

```
sem_t forks[5]; //array of semaphores, one for each fork

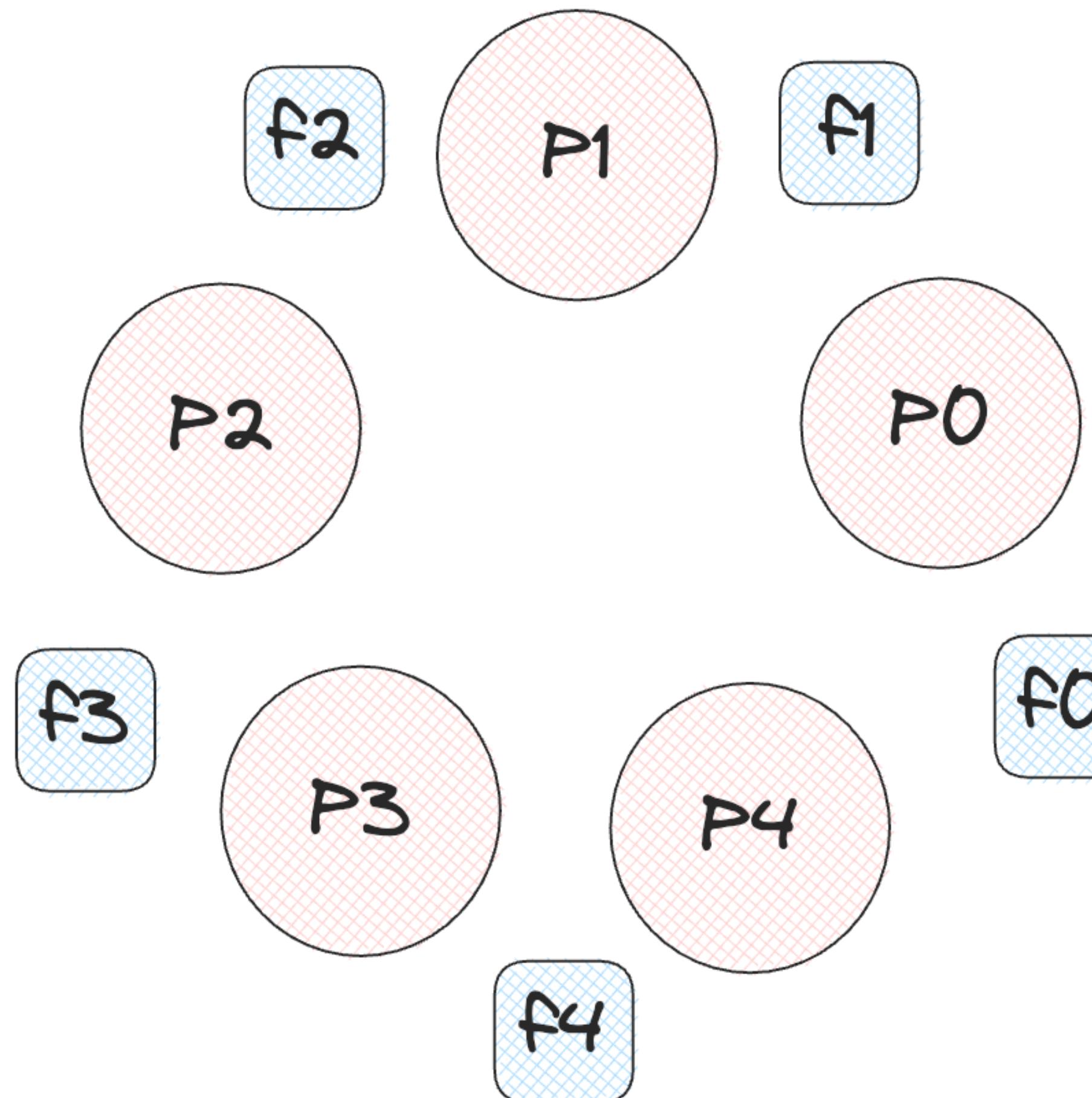
void get_forks(int p)
{
    sem_wait(&forks[left(p)])
    sem_wait(&forks[right(p)])
}

void put_forks(int p)
{
    sem_post(&forks[left(p)])
    sem_post(&forks[right(p)])
}
```

Any issues here? Deadlock!!, How?



How deadlock happened?



- Each philosopher is one thread and they start running
- The first philosopher (0) has wait on 1, gets it (since initial semaphore is 1)
- Immediately second philosopher (1) runs, wait on 0, but gets on right
- Third will run, waits on 2nd fork but gets the 4th one
- Fourth will run, waits on third but waits on 0
- **All philosophers wait for their left fork and we have a deadlock**

Possible Solution

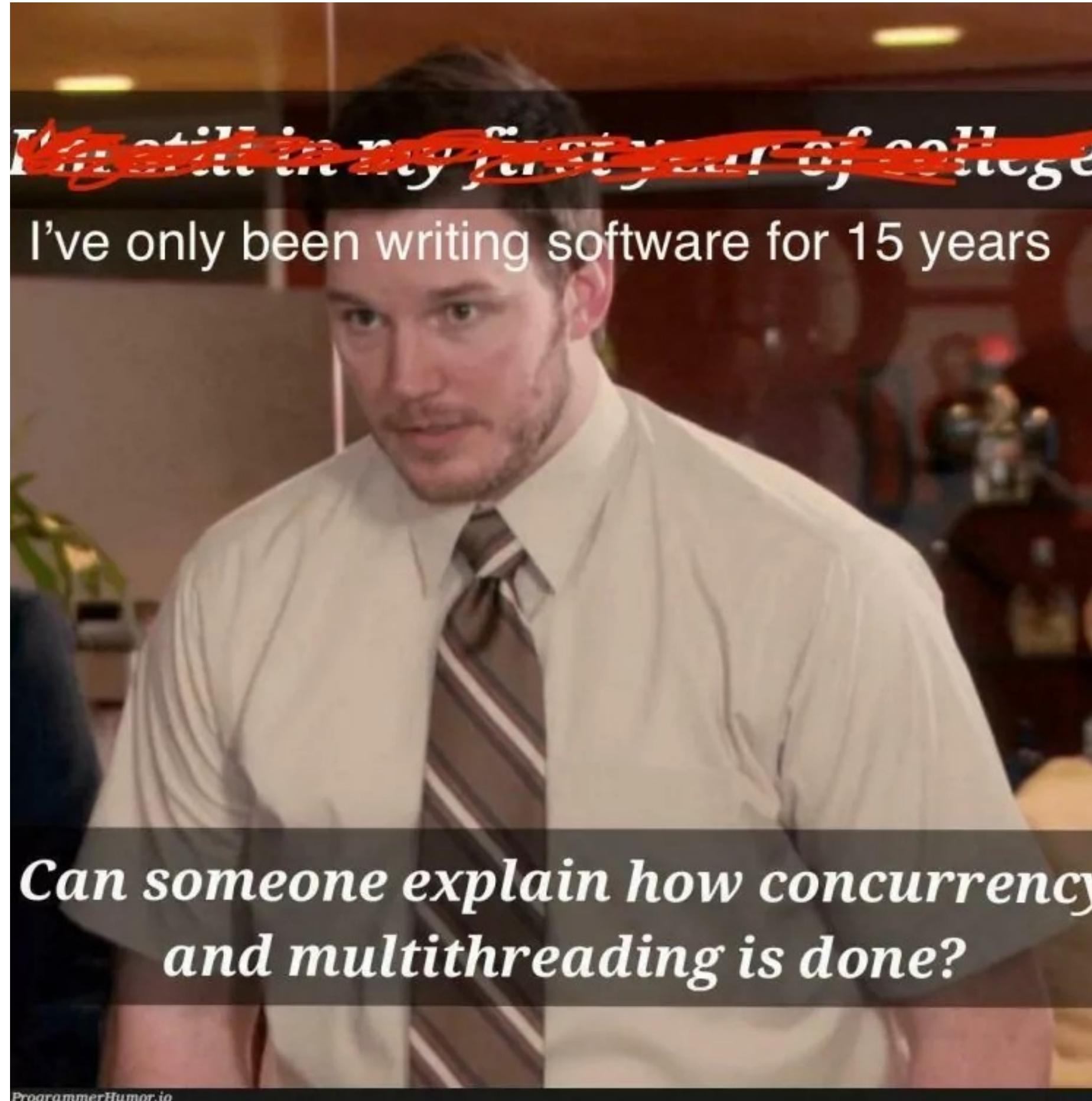
● ● ● Dinning Philosophers Problem

```
sem_t forks[5]; //array of  
semaphores, one for each fork  
  
void get_forks(int p)  
{  
    if (p==4)  
    {  
        sem_wait(&forks[right(p)])  
        sem_wait(&forks[left(p)])  
    }  
    sem_wait(&forks[left(p)])  
    sem_wait(&forks[right(p)])  
}  
  
void put_forks(int p)  
{  
    sem_post(&forks[left(p)])  
    sem_post(&forks[right(p)])  
}
```

- Change the order in which they eat
- Philosopher 4 acquires the fork in a different order
- There won't be a situation in which one philosopher grabs one and has to wait for other
- **The cycle of waiting is broken**
- More solutions exist!



Concurrency Can be tricky!



There are some common **Concurrency Bugs** that can help identify some common bugs



Types of Bugs

- Bugs are very non-deterministic - Occurrence order cannot be fixed
- Two types of bugs
 - **Non-deadlock bugs:** Incorrect results when threads execute
 - **Deadlock bugs:** Threads keep waiting for each other

Application	Description	# of Bug Samples	
		Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Browser Suite	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. **Learning from mistakes: a comprehensive study on real world concurrency bug characteristics**, ASPLOS, 2008



Non-deadlock Bugs

Findings on Bug Patterns (Section 3)	Implications
(1) Almost all (97%) of the examined non-deadlock bugs belong to one of the <i>two simple bug patterns</i> : atomicity-violation or order-violation*.	Concurrency bug detection can focus on these two bug patterns to detect most concurrency bugs.
(2) About one third (32%) of the examined non-deadlock bugs are <i>order-violation bugs</i> , which are <i>not</i> well addressed in previous work.	<i>New</i> concurrency bug detection tools are needed to detect order-violation bugs, which are not addressed by existing atomicity violation or race detectors.

Non-deadlock bugs make the majority of the bugs among concurrency bugs

- Two types of non-deadlock bugs
 - Atomicity violation bugs
 - Order violation bugs



Atomicity Bugs

- Atomicity assumptions made during development are violated during execution of threads
- **Example:** From MySQL where one thread reads and modifies a shared variable while other tries to modify it

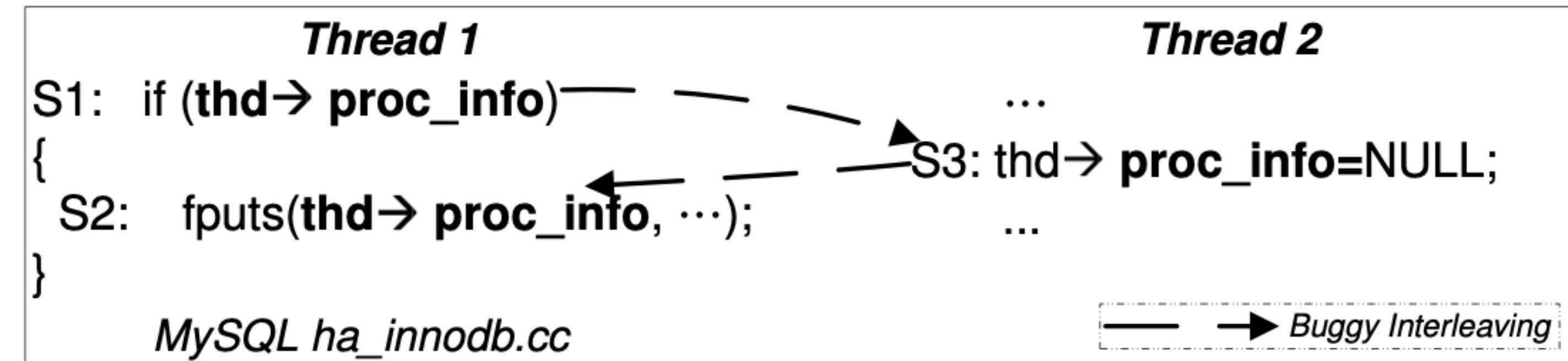
Atomicity Bug

Thread 1::

```
if (thd->proc_info)
    fputs(thd->proc_info, ...);
....
```

Thread 2::

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



How to go about solving it?



Atomicity Bugs

Use locks when accessing shared data



Atomicity Bug

```
pthread_mutex_t thd_proc_info = PTHREAD_MUTEX_INITIALIZER;
Thread 1::

pthread_mutex_lock(&thd_proc_info);
if (thd->proc_info)
{
    ...
    fputs(thd->proc_info,...);
    ...
}
pthread_mutex_unlock(&thd_proc_info);
.....


Thread 2::
pthread_mutex_lock(&thd_proc_info);
thd->proc_info = NULL;
pthread_mutex_unlock(&thd_proc_info);
```

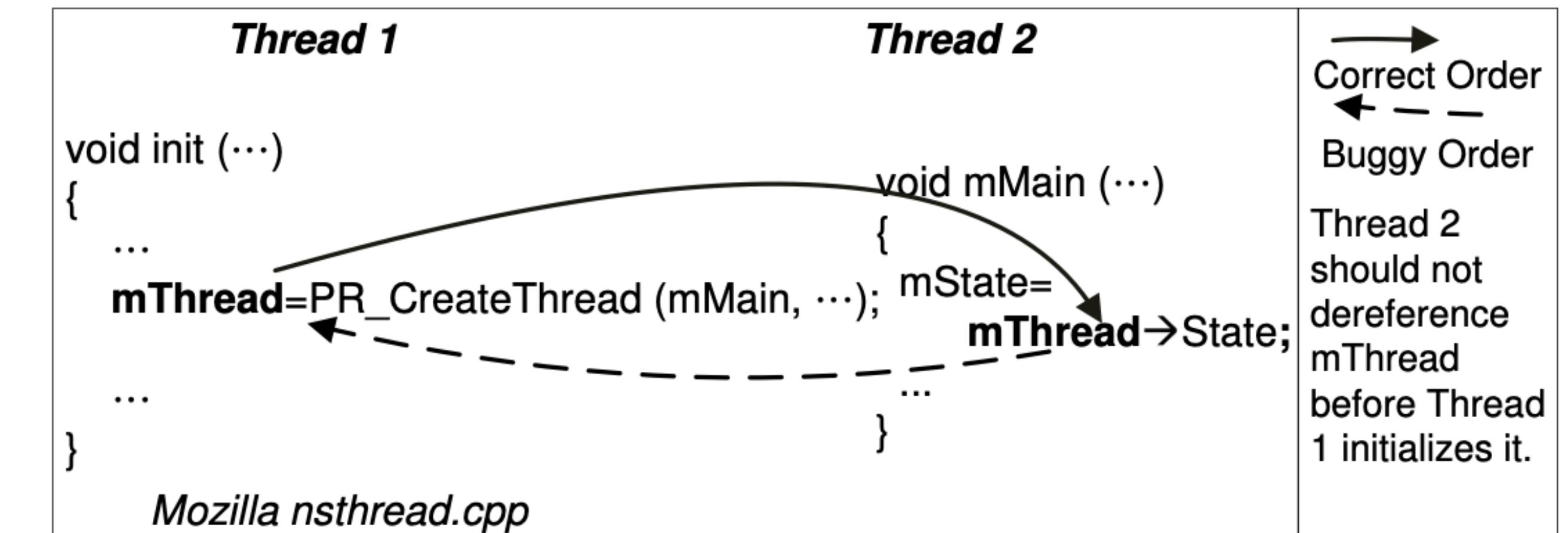


Order Violation Bugs

- Desired/assumed order of execution of memory access is violated during concurrent execution of threads
- Example:** Assume thread 1 and thread 2. Thread 2 may assume that thread 1 has already run

Order Violation Bug

```
Thread 1::  
  
void init(..)  
{  
    ...  
    mThread = PR_CreateThread(mMain, ...);  
    ...  
}  
  
Thread 2::  
  
void mMain(..)  
{  
    mState = mThread->state;  
}
```



How to go about solving it?

Order Violation Bugs

Use condition variables or semaphores

Order Violation Bug - Solution

```
pthread_mutex_t mLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  mCond  = PTHREAD_COND_INITIALIZER;
int mInit = 0;
```

Thread 1::

```
void init(..)
{
    ...
    mThread = PR_CreateThread(mMain,...);
    pthread_mutex_lock(&mLock);
    mInit = 1
    pthread_cond_signal(&mCond);
    pthread_mutex_unlock(&mLock);
    ...
}
```

Thread 2::

```
void mMain(..)
{
    pthread_mutex_lock(&mLock);
    while(mInit == 0)
        pthread_cond_wait(&mCond,&mLock);
    pthread_mutex_unlock(&mLock);
    mState = mTrhead->state;
    ...
}
```

- Use condition variables
 - Dependant thread can wait for dependency operation to be completed
 - Use combination of wait and signal
 - Semaphores can also be used here!
- **Remember:** Locks are still needed to handle the shared variable operation

Deadlock Bugs

(7) Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most *two resources*.

Pairwise testing on the acquisition/release sequences to two resources can expose most deadlock concurrency bugs, and reduce testing complexity.

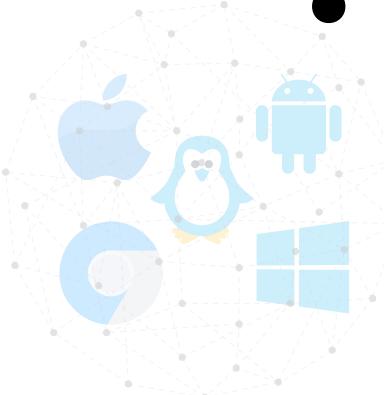
Thread 1

```
pthread_mutex_lock(L1);  
pthread_mutex_lock(L2);
```

Thread 2

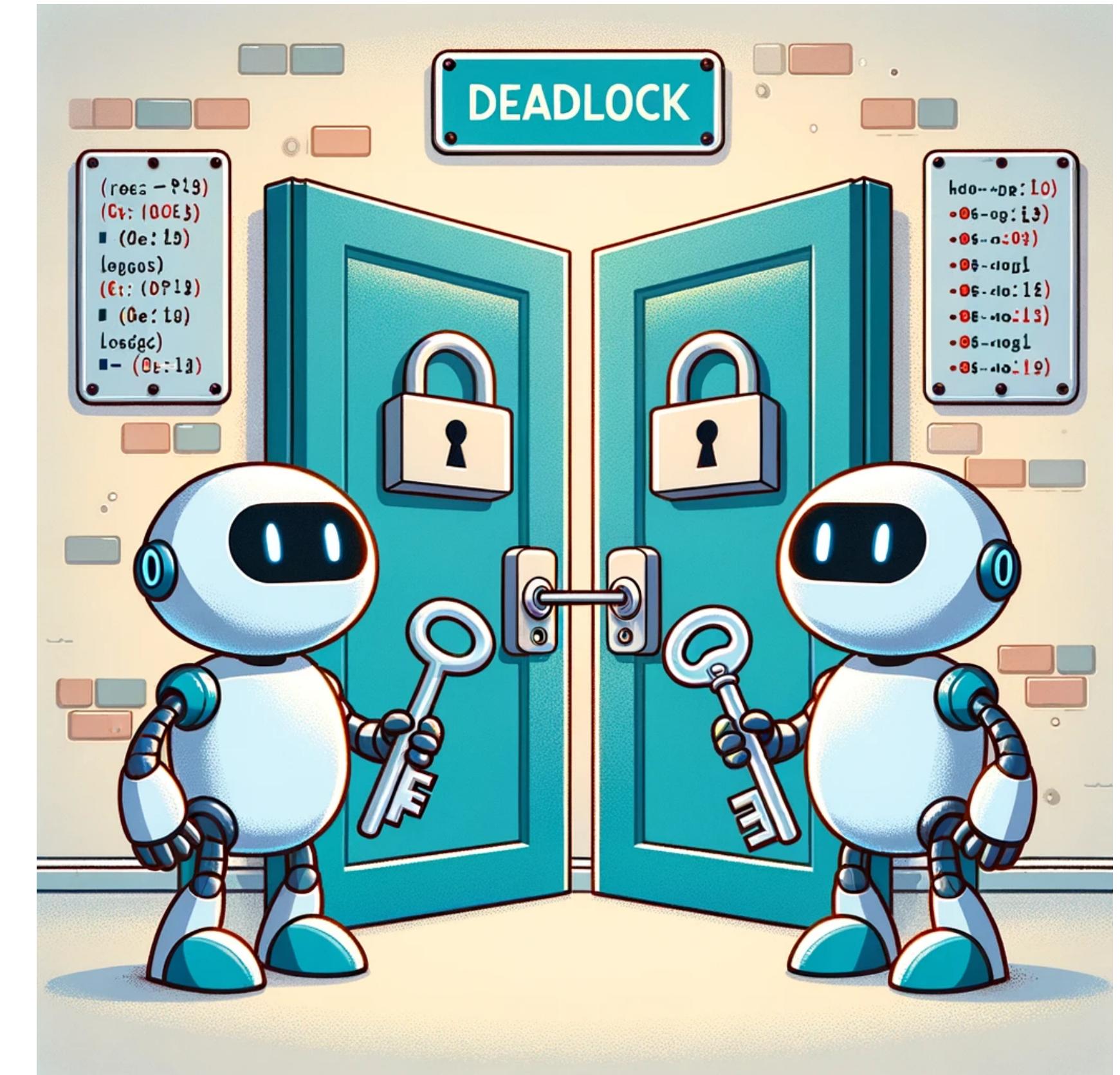
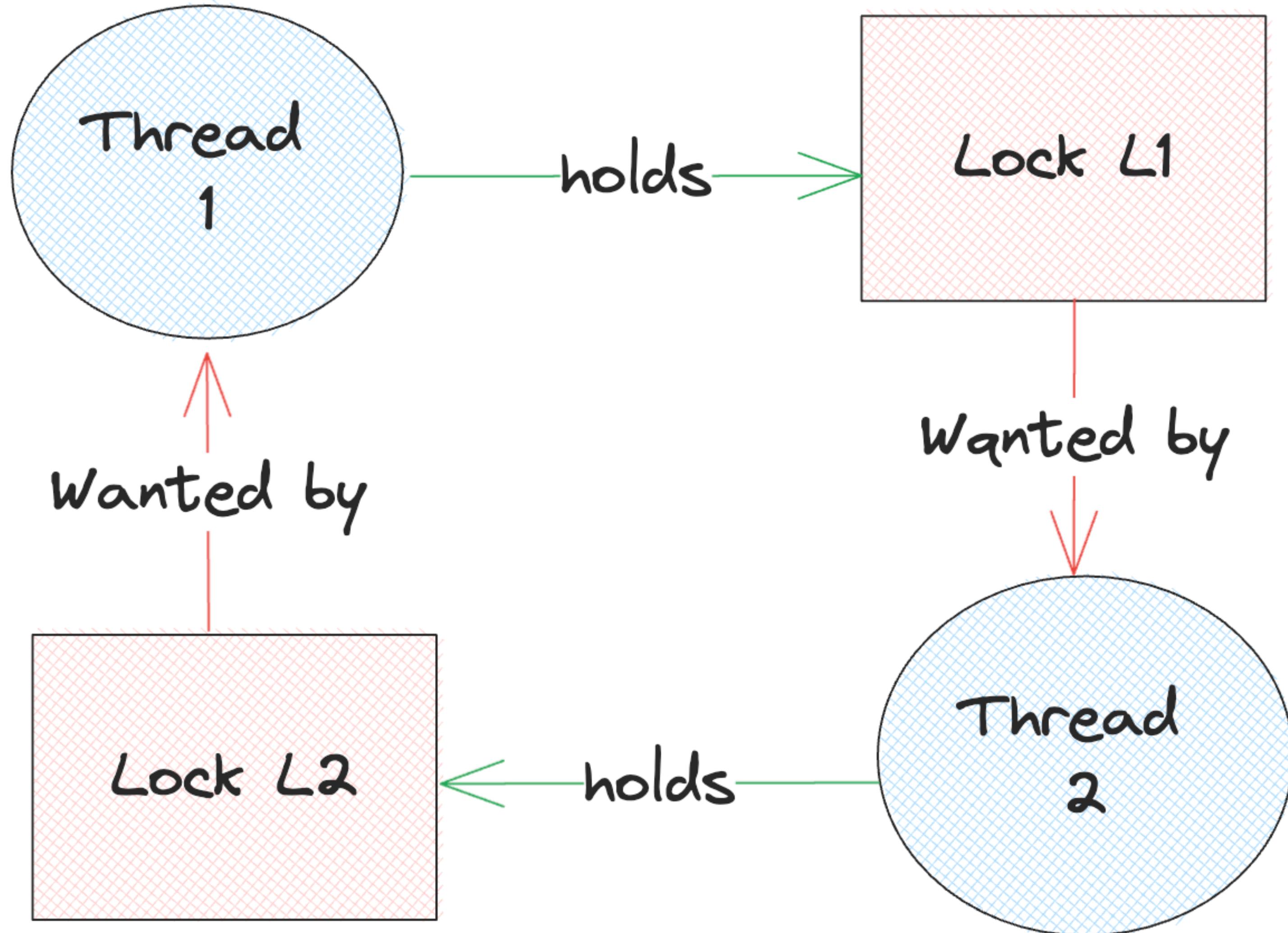
```
pthread_mutex_lock(L2);  
pthread_mutex_lock(L1);
```

- Its not always the case that deadlock occurs
- If executions overlap and context switches from thread after acquiring one lock



Deadlock: A Visual Representation

Cycle in a dependency graph



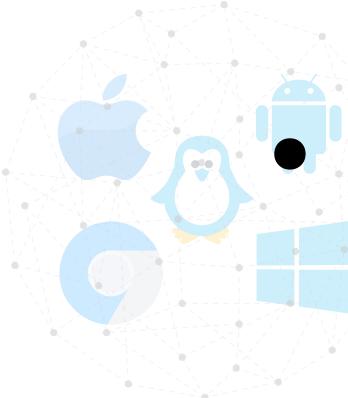
Generated by DALLE-3



Conditions for deadlock

Four conditions should together hold for deadlock

- **Mutual Exclusion:** Thread claims exclusive control of a resource (eg: lock)
- **Hold-and-wait:** Thread holds a resource and is waiting for another
- **No Preemption:** Thread cannot be made to give up its resource (eg: cannot take back a lock)
- **Circular Wait:** There exists a cycle in the resource dependency graph



Prevention of Circular Wait

- Acquire locks in a particular order
 - Eg: Thread 1 and thread 2 acquires lock in the same order
- Provide a **total ordering** for lock acquisition
 - If there are only two locks, L1 and L2 => always acquire L1 before L2
 - In more complex systems, more than two locks exist => **partial ordering**
 - Some locks can be given higher ordering than other locks

```
if (m1 > m2)
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1); }
```



- Lock ordering can also be done using the address of the lock

Preventing Hold-and-Wait

- Hold all the locks at once, atomically by acquiring a master lock first

```
pthread_mutex_lock(master);  
pthread_mutex_lock(L1);  
pthread_mutex_lock(L2);  
....  
...  
pthread_mutex_unlock(master);
```

- This may have an impact on concurrent execution and performance gains



“Trying” to get some Preemption Done

```
top:  
    pthread_mutex_lock(L1);  
    .  
    if (pthread_mutex_trylock(L2)≠ 0) {  
        pthread_mutex_unlock(L1);  
        goto top;  
    }
```

- Thread can try for a lock before getting it - **pthread_mutex_trylock**
- Function returns 0 on successfully acquiring the lock
- If other thread also does in same order => **possibility of livelock**
- Periodic delay can be added to avoid live locking



What about avoiding need for mutual exclusion?

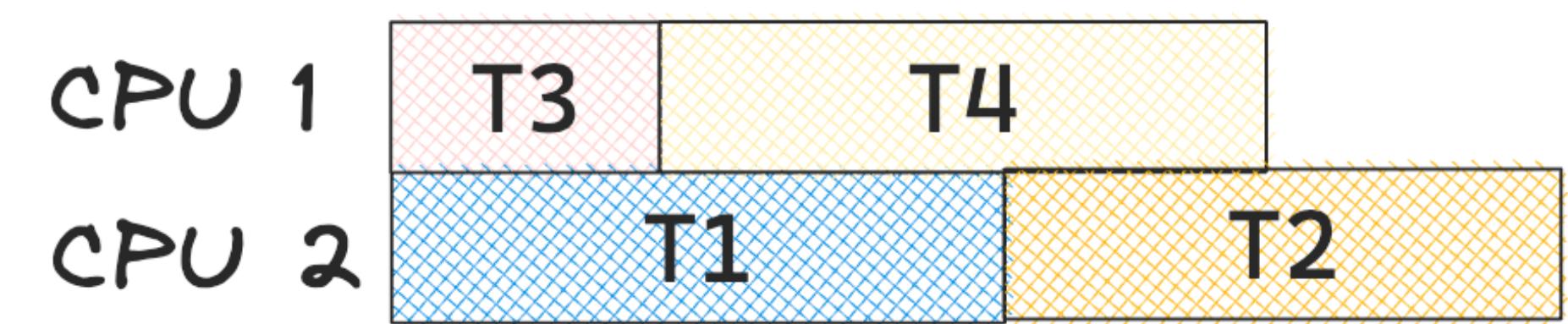
- Not using any locks like pthread_locks or condition variables
- Using powerful hardware instructions
 - No need to do explicit locking
 - Hardware primitives like Compare-and-swap can be used
 - For instance, atomic incremental of shared value can be done using 1 line of compare and swap
 - No lock, no deadlock but livelock is still a possibility



Deadlock Avoidance

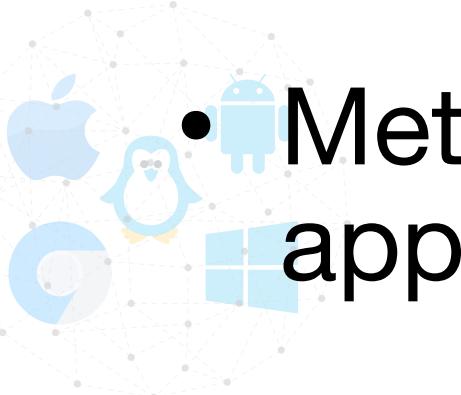
- In some scenarios avoidance is preferable instead of prevention
- Deadlock avoidance via **Scheduling**
 - If OS knows which threads requires locks at which point of times, it can schedule them accordingly

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



- T1 and T2 are not run at the same time
- T1 and T3 do not share a lock

• Methods like **Bankers algorithm** by Dijkstra have been suggested but practically not applicable



Deadlock Avoidance

Detect and Recover

- Allow deadlocks to occur occasionally and take some action
 - If OS freezes, reboot the system
 - Some systems like databases employ deadlock detection and recovery technique
 - Deadlock detector **runs periodically**
 - **Resource graph** is created to detect cycles
 - In the event of cycles, **restart the system**





Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: [@karthi_ishere](https://twitter.com/karthyishere)

