死锁

Section 4: Part 2

2024

并发程序中的 Bugs

真实世界中并发程序 Bugs 特征的实证研究 (empirical study)

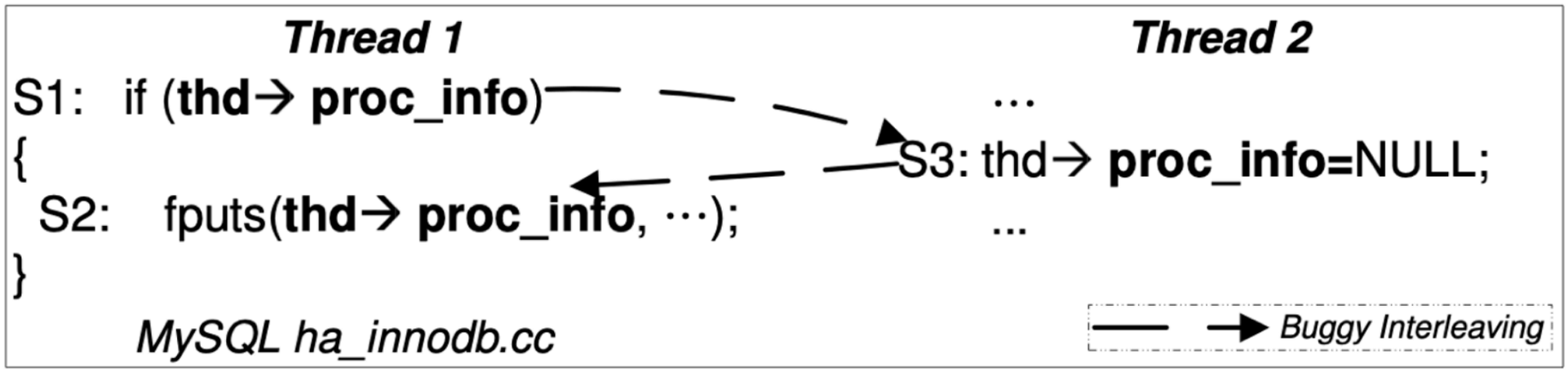
- 绝大部分 non-deadlock bugs (97%) 都是由于违反原子性或顺序要求而造成的

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. 2008: 329-339

并发程序中的 Bugs

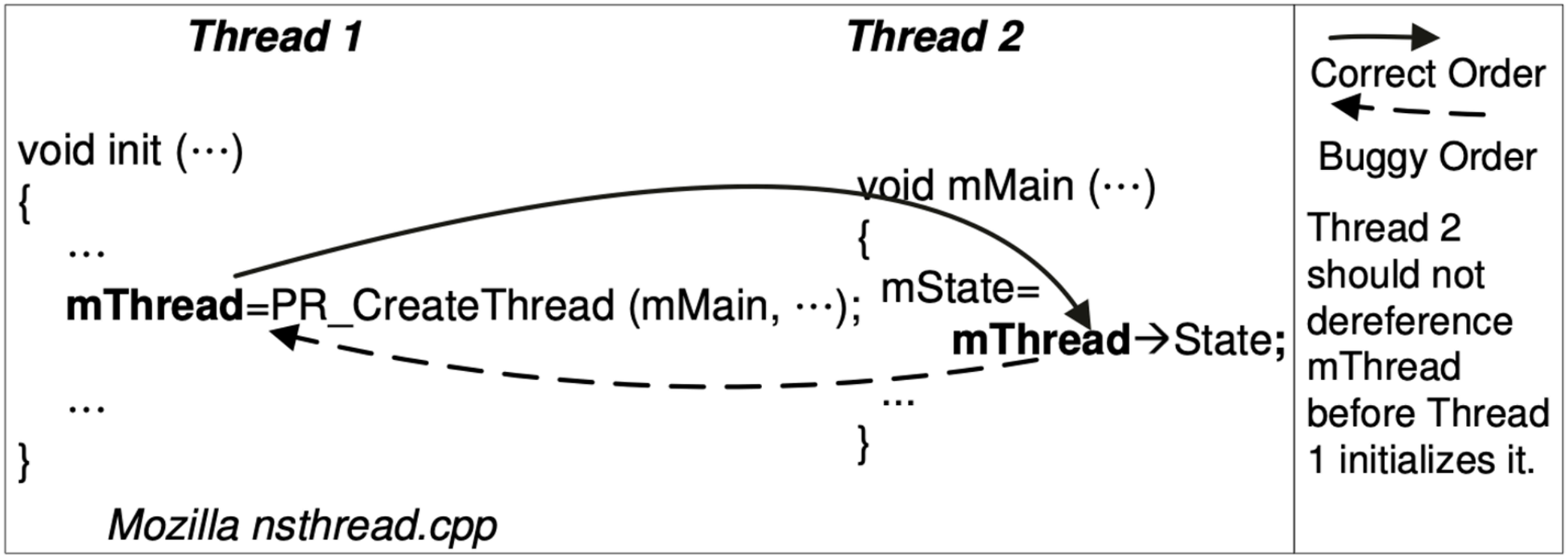
Atomicity Violation



The desired serializability among multiple memory accesses is violated (i.e., a code region is intended to be atomic, but the atomicity is not enforced during execution)

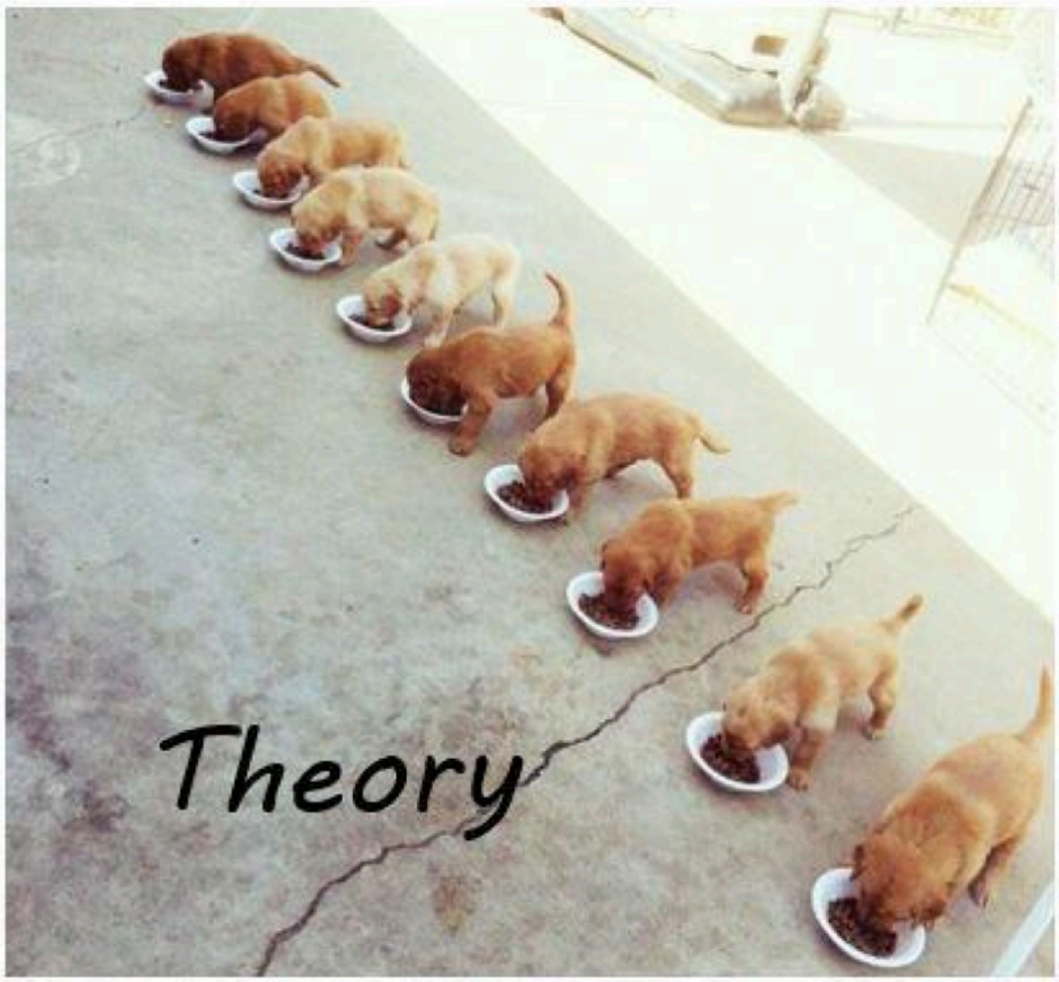
并发程序中的 Bugs

Order Violation



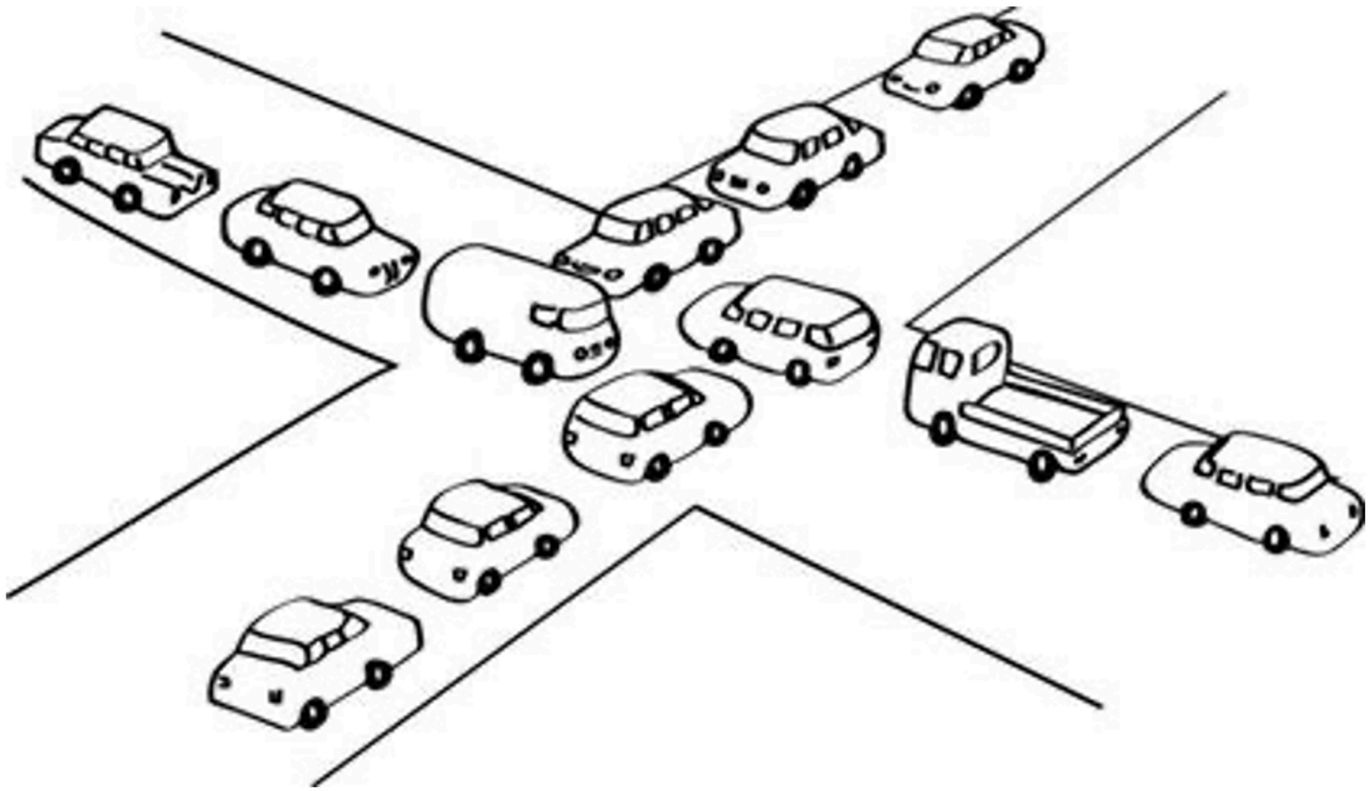
The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced)

多线程编程: 从入门到放弃



死锁 DeadLock

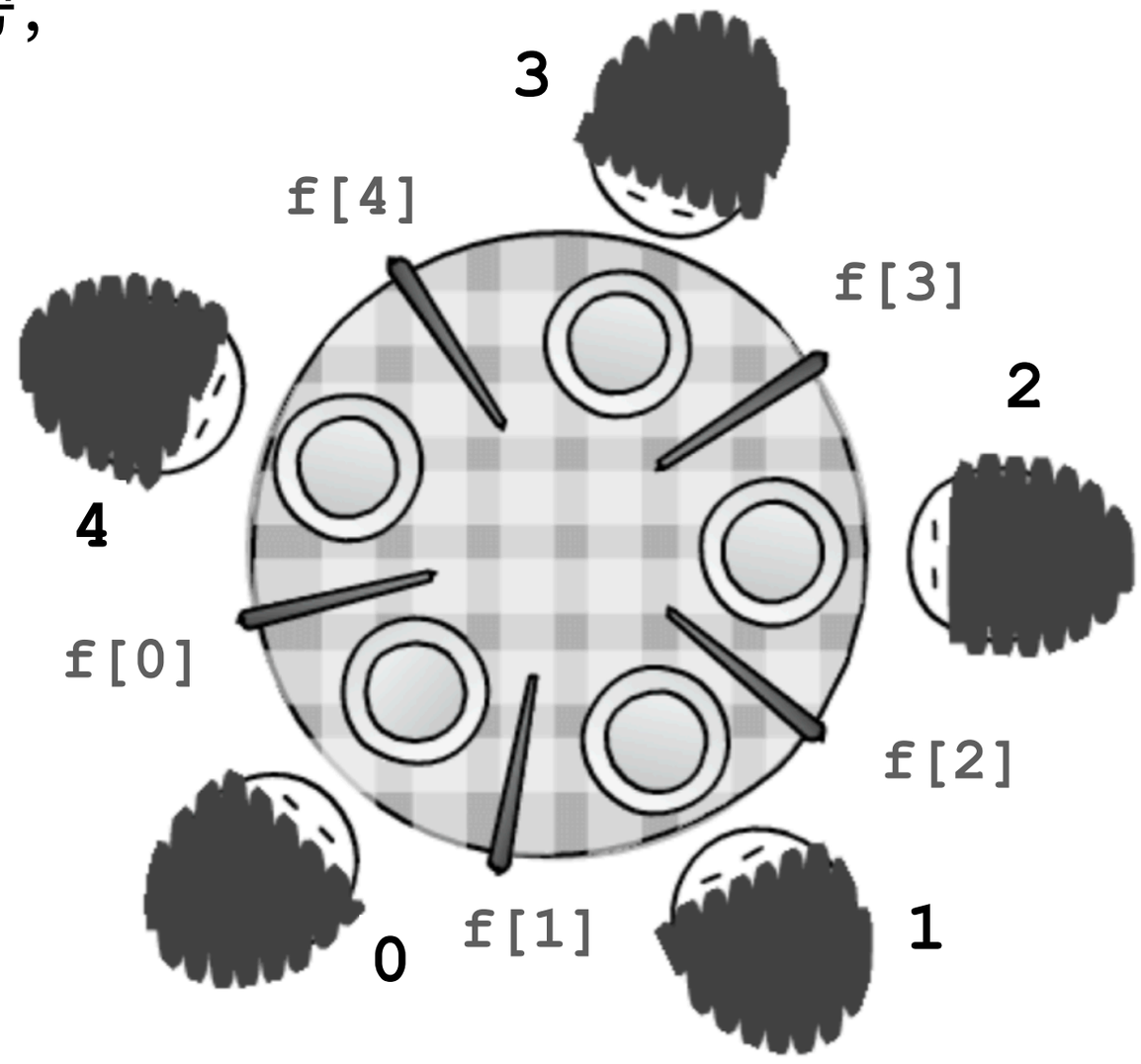
每个线程都在同时持有一些资源而又在等待获取其它线程正持有的另一些资源，从而导致所有线程都无法推进执行的一种情况



Dining Philosophers Problem

Dijkstra 提出的一个并发问题，并经由 Hoare 修改得到了如今的版本

- 五位哲学家围坐在一个圆形餐桌旁，
  每两位哲学家之间有一个叉子
- 每位哲学家就做两个事情:
  think 或 *eat*
  - 在 *think* 时不需要叉子
  - 需要持有左右手两侧的
    两个叉子才能 *eat*
- 对有限资源 (五个叉子) 的争用



Dining Philosophers Problem

```
void philosophers(i) {   // i: philosopher number
  while(1) {
    think();
    get_fork();
    eat();
    put_fork();
  }
}
```

Dining Philosophers Problem

```
sem_t forks[5];  // one for each fork, initialise to 1

void philosophers(i) {
  while(1) {
    think();
    P(fork[left]);   // left = i
    P(fork[right]);  // right = (i + 1) % 5
    eat();
    V(fork[left]);
    V(fork[right]);
  }
}
```

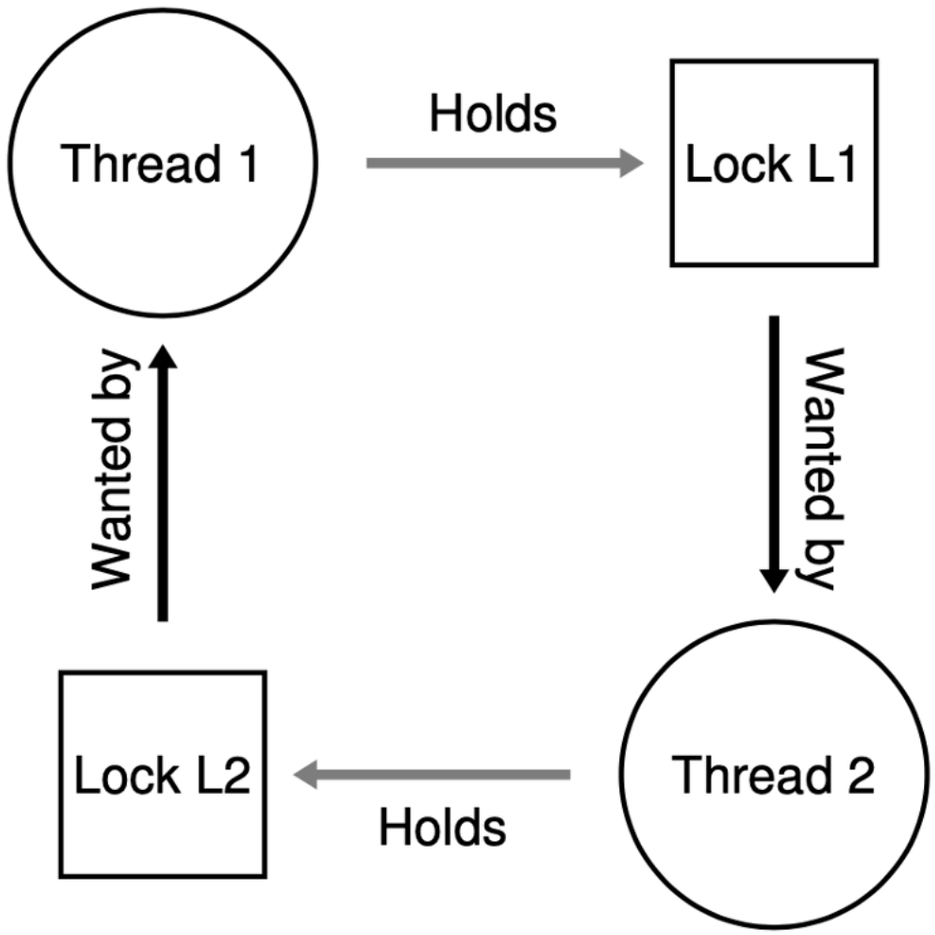
Deadlock

死锁 DeadLock

每个线程都在同时持有一些资源而又在等待获取其它线程正持有的另一些资源，从而导致所有线程都无法推进执行的一种情况

```
Thread 1        Thread 2

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



A cycle of waiting for resources

死锁 DeadLock

每个线程都在同时持有一些资源而又在等待获取其它线程正持有的另一些资源，从而导致所有线程都无法推进执行的一种情况

Thread 1

```
lock(L1);
lock(L2);
while(need to wait)
  wait(cv, L2);
unlock(L2);
unlock(L1);
```

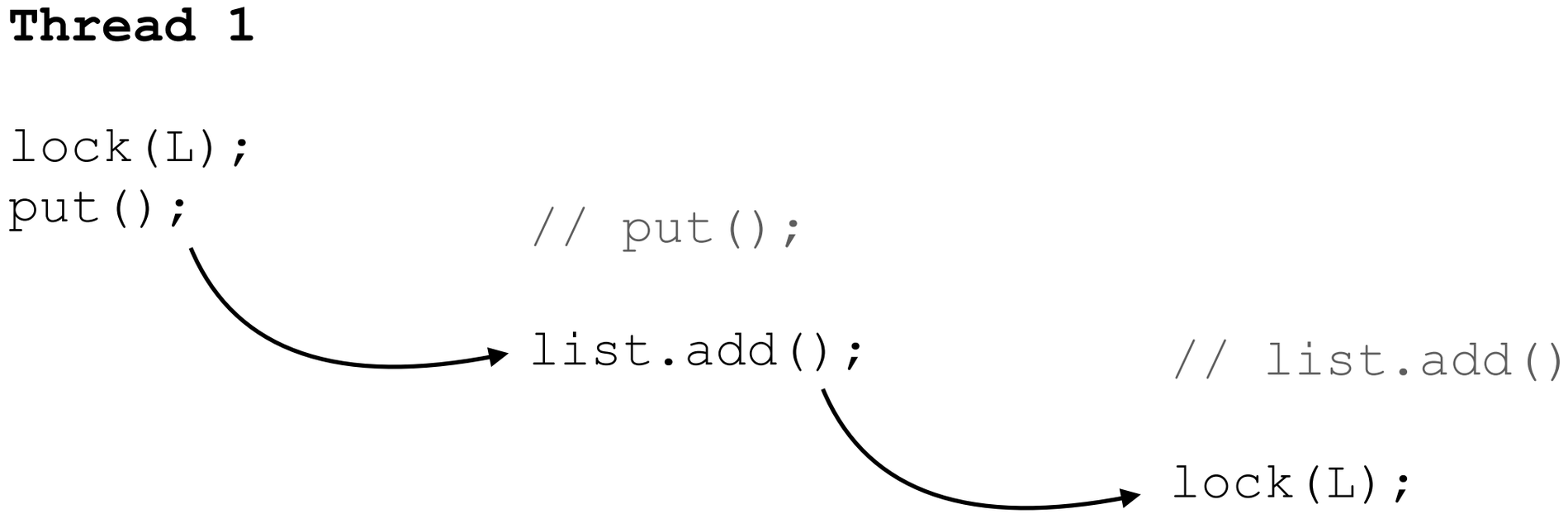
Thread 2

```
lock(L1);
lock(L2);
signal(cv);
unlock(L2);
unlock(L1);
```

Deadlock occurs if the thread that can signal the condition variable needs the first lock to make progress

死锁 DeadLock

每个线程都在同时持有一些资源而又在等待获取其它线程正持有的另一些资源，从而导致所有线程都无法推进执行的一种情况



A single thread can also cause deadlock

死锁 DeadLock

每个线程都在同时持有一些资源而又在等待获取其它线程正持有的另一些资源，从而导致所有线程都无法推进执行的一种情况

```
Vector v1, v2;   // to be thread safe, locks for
                 // both v1 and v2 need to be acquired
```

Thread 1

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

Thread 2

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

Deadlocks might also occur due to encapsulation
(detailed implementation is hidden from the calling thread)

死锁 DeadLock

死锁的产生需要满足四个必要条件：

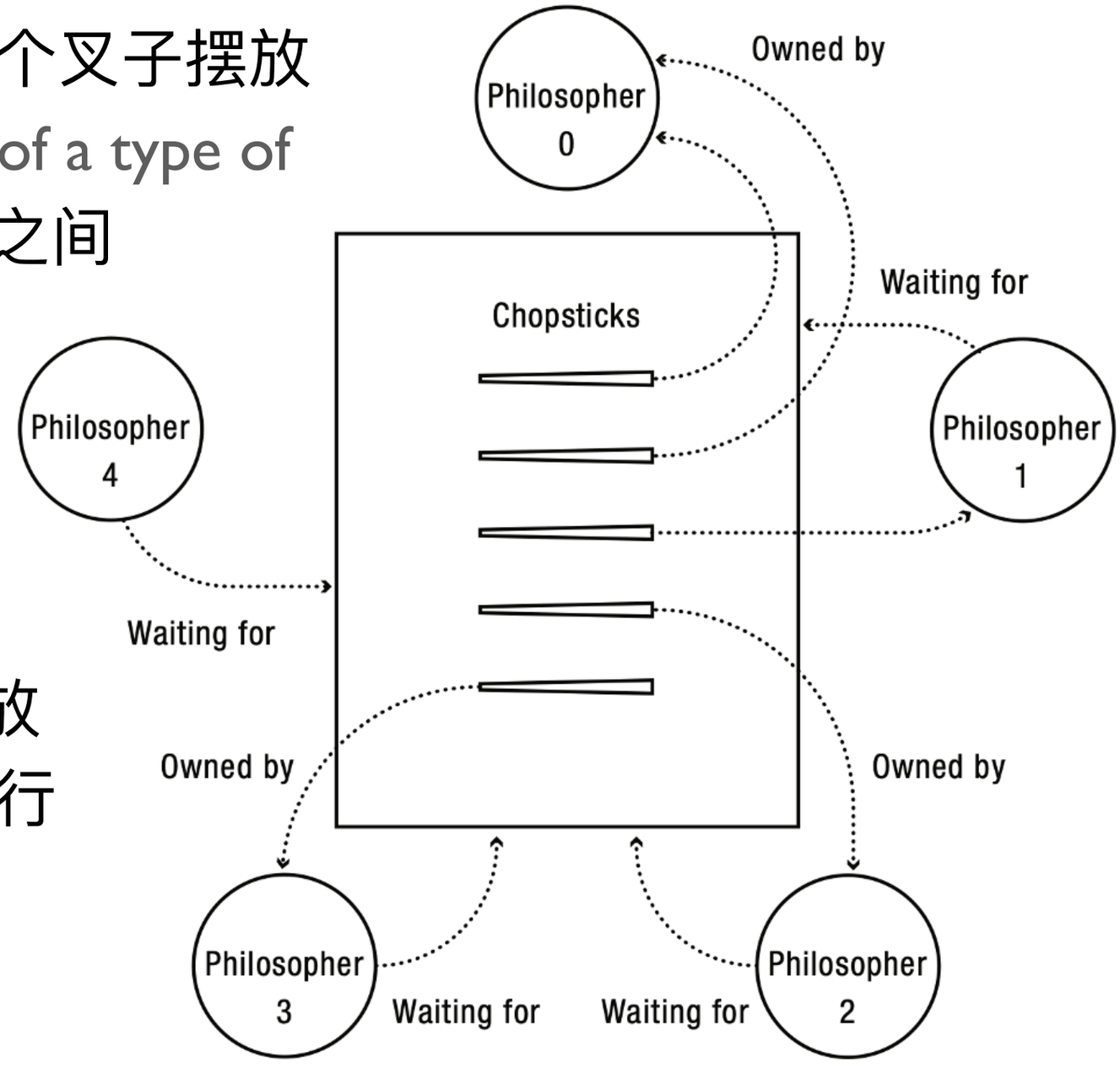
- Mutual Exclusion: 线程对资源的使用是互斥的 (e.g., locks)
- Hold and Wait: 线程当前正持有一些资源 (e.g., locks that have already acquired) 并且在等待获取另外一些资源 (e.g., locks that wish to acquire)
- No Preemption: 线程当前持有的资源不可以被抢占
- Circular Wait: 线程对资源的持有和请求形成一种循环等待的情形 (每个线程都持有其它线程想获取的资源)

Coffman E G, Elphick M, Shoshani A. System deadlocks. ACM Computing Surveys (CSUR), 1971, 3(2): 67-78

死锁 DeadLock

注意上述四个条件是死锁的必要条件，而非充分条件

- 假设对于哲学家就餐问题，五个叉子摆放在桌子正中 (multiple instances of a type of resource)，而不是两两哲学家之间
  - 当前 `P0` 持有两个叉子
  - `P1`, `P2`, `P3` 各持有一个叉子，并构成一种循环等待状态
  - 此时环外的 `P0` 可以通过释放叉子来使其它哲学家推进执行



应对死锁的方法

- 忽略死锁 (Ignoring Deadlocks)
  - 系统有问题重启就好了
  - 在问题发生频率很低且代价较小的情况下是一种非常实际的选择
- 预防死锁 (Preventing Deadlocks)
  - 通过破坏死锁的必要条件来防止死锁的产生
- 避免死锁 (Avoiding Deadlocks)
  - 通过精心的资源分配来避免系统进入死锁状态
- 死锁检测和恢复 (Detecting and Recovering from Deadlocks)
  - 定期检测当前是否出现了死锁，并在产生死锁后尝试进行恢复

预防死锁

破坏 Mutual Exclusion 条件

- 使用 lock-free 方法来实现互斥 (例如借助硬件原子指令的帮助)
  - 依赖具体的硬件、且非常容易出错

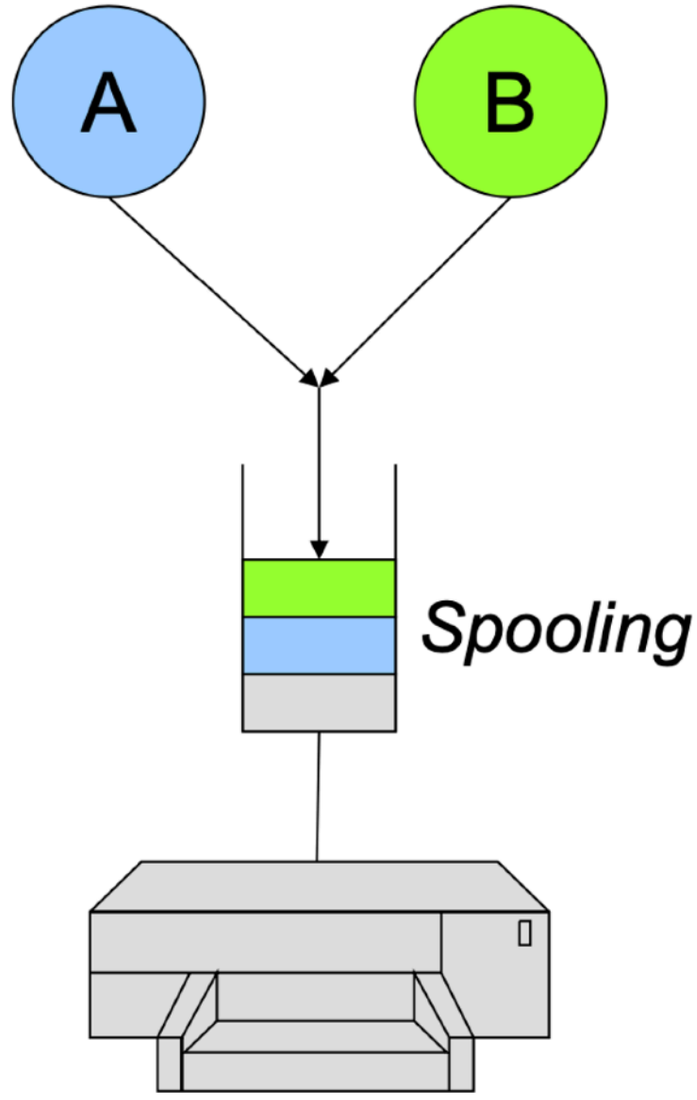
```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  n->value = value;
  lock(&lock);     // critical section
  n->next = head;
  head = n;
  unlock(&lock);   // end critical section
}
```

```
do {
  n->next = head;
} while(cmpxchg(&head, n->next, n) != n->next);
```

预防死锁

破坏 Mutual Exclusion 条件

- 提供充足的资源
- 例如，Spooling 技术 (for resource like printer)
  - 利用存储资源来虚拟化打印机 (virtualise a printer into multiple resources)
  - 由一个 printer daemon 进程来和物理打印机进行交互
  - 其它进程不需要等待打印机完成 (一个物理打印机同时被多个进程所使用)



预防死锁

破坏 Mutual Exclusion 条件

- 提供充足的资源 → 解决哲学家就餐问题

```
sem_t forks[5];
sem_t can_eat = K;

void philosophers(i) {
  while(1) {
    think();
    P(can_eat);
    P(fork[left]);
    P(fork[right]);
    eat();
    V(fork[left]);
    V(fork[right]);
    V(can_eat);
  }
}
```

At most `K` philosopher can eat simultaneously (bounded resources)

预防死锁

破坏 Hold and Wait 条件

- 要求所有线程一次性获得所有需要的资源 (e.g., 要么能一次性得到所有的锁，要么什么锁也不获取)
  - 但是，当前线程执行需要多少资源往往是动态决定的
  - 要求提前获取在未来才需要的资源也会伤害并发

```
lock(mutex);
lock(L1);
lock(L2);
unlock(mutex);
```

预防死锁

破坏 No Preemption 条件

- 允许线程在请求一个不能立即分配的资源时抢占该资源
  - 不是每种资源都可以被强制抢占的 (一个线程可以去抢别的线程当前正持有的内存资源，但如果资源是打印机或锁？)
  - 即使可以让线程主动放弃持有的资源，但也会存在活锁 (live lock) 的问题 (不断重复资源的请求和释放操作)

```
retry:
  lock(L1);
  if (try_lock(L2) != 0) {
    // should carefully release resources
    unlock(L1);
    goto retry;
  }
```

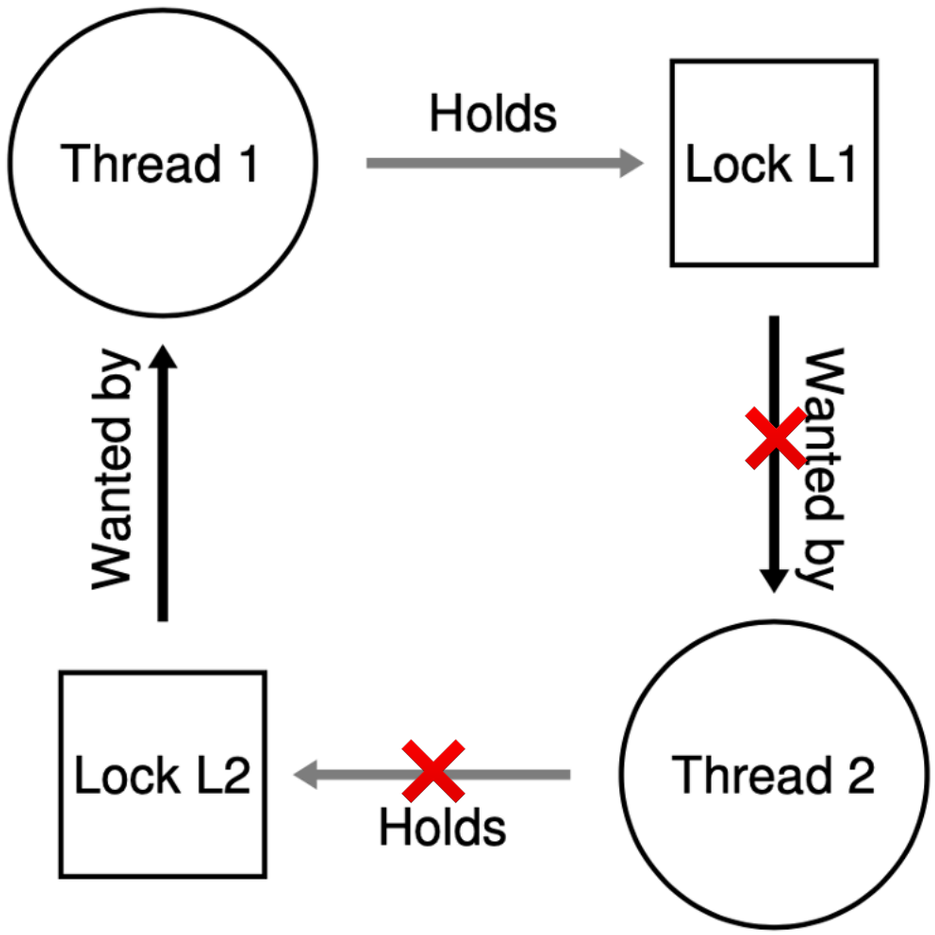
预防死锁

破坏 Circular Wait 条件

- 强制按照特定的顺序来申请资源

```
Thread 1        Thread 2

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



预防死锁

破坏 Circular Wait 条件

- 强制按照特定的顺序来申请资源 → 解决哲学家就餐问题

```
sem_t forks[5];

void philosophers(i) {
  while(1) {
    think();
    if (i == 4)
      P(fork[right]);
      P(fork[left]);
    else
      P(fork[left]);
      P(fork[right]);
    eat();
    V(fork[left]);
    V(fork[right]);
  }
}
```

Acquire the locks in a specific order

预防死锁

破坏 Circular Wait 条件

- 强制按照特定的顺序来申请资源 → 在 Kernel 中得到广泛应用

```
/*
 * Lock ordering in mm:
 *
 * inode->i_rwsem (while writing or truncating, not reading or faulting)
 *   mm->mmap_lock
 *     mapping->invalidate_lock (in filemap_fault)
 *       page->flags PG_locked (lock_page)
 *         hugetlbfs_i_mmap_rwsem_key (in huge_pmd_share, see hugetlbfs below)
 *           vma_start_write
 *             mapping->i_mmap_rwsem
 *               anon_vma->rwsem
 *                 mm->page_table_lock or pte_lock
 *                   swap_lock (in swap_duplicate, swap_info_get)
 *                     mmlist_lock (in mmput, drain_mmlist and others)
 *                     mapping->private_lock (in block_dirty_folio)
 *                       folio_lock_memcg move_lock (in block_dirty_folio)
 *                         i_pages lock (widely used)
 *                           lruvec->lru_lock (in folio_lruvec_lock_irq)
 *                     inode->i_lock (in set_page_dirty's __mark_inode_dirty)
 *                     bdi.wb->list_lock (in set_page_dirty's __mark_inode_dirty)
 *                       sb_lock (within inode_lock in fs/fs-writeback.c)
 *                       i_pages lock (widely used, in set_page_dirty,
 *                                 in arch-dependent flush_dcache_mmap_lock,
 *                                 within bdi.wb->list_lock in __sync_single_inode)
```

Linux Kernel: mm/rmap.c

预防死锁

破坏 Circular Wait 条件

- 强制按照特定的顺序来申请资源 → 在 Kernel 中得到广泛应用
  - 虽然仔细设计了锁的获取顺序和策略，但程序员在写代码时很容易忽视该约定

Preventing Deadlock

Textbooks will tell you that if you always lock in the same order, you will never get this kind of deadlock. Practice will tell you that this approach doesn't scale: when I create a new lock, I don't understand enough of the kernel to figure out where in the 5000 lock hierarchy it will fit.

The best locks are encapsulated: they never get exposed in headers, and are never held around calls to non-trivial functions outside the same file. You can read through this code and see that it will never deadlock, because it never tries to grab another lock while it has that one. People using your code don't even need to know you are using a lock.

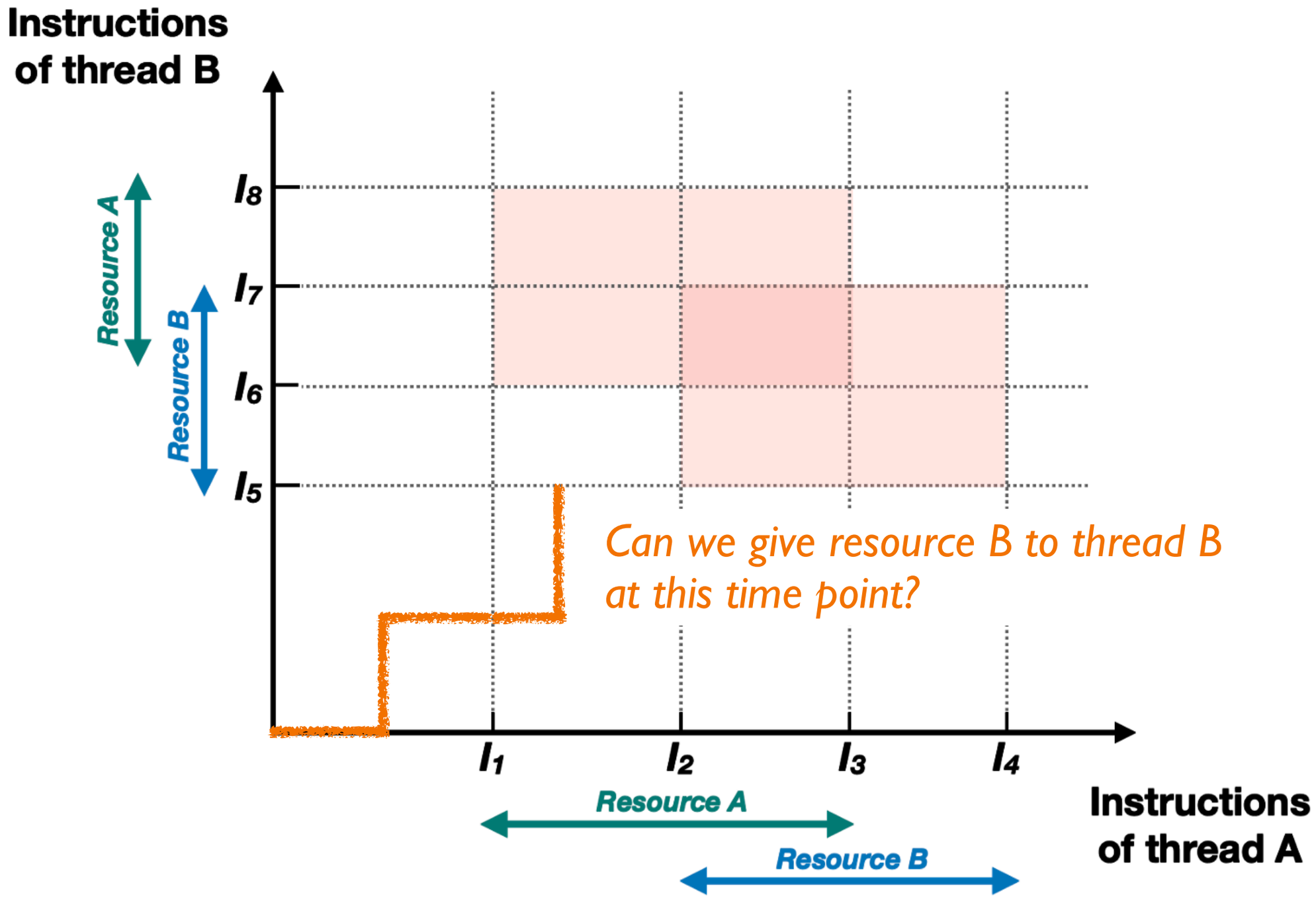
https://www.kernel.org/doc/html/v4.16/kernel-hacking/locking.html

避免死锁

通过精心的资源分配来避免系统进入死锁状态

- 假设系统中有一定数量的各类资源，同时假设每个线程对每种资源都有一个需求上限
- 每个线程在执行过程中会动态请求和释放资源
  - 在每次资源请求时，谨慎判断进行该资源分配是否会导致系统从安全 (safe) 状态变为非安全 (unsafe) 状态
  - 如果会进入不安全状态，则推迟相关资源的分配
- 如果每个线程最终都获得了所需的各类资源上限，则该线程最终将执行结束并释放其持有的所有资源

避免死锁



Resource trajectories of two threads and two resources

Banker's Algorithm

通过精心的资源分配来确保系统总是处于安全 (safe) 状态

- 安全状态: 存在至少一个资源分配序列，以使得最终系统中所有线程都能执行结束 (即所有线程都能获得其所需的各类资源上限)
- 为此，在系统运行过程中需要追踪如下信息:
  - ***Existing Resource*** E: 系统中每类资源的总数
  - ***Maximum Claim*** M: 系统中每个线程对每类资源的上限需求
  - ***Available Resource*** A: 系统中当前每类资源的可用数目
  - ***Current Allocation*** C: 系统中当前每个线程持有每类资源的数目
  - ***Request*** R: 系统中当前每个线程对每个资源的请求情况

Banker's Algorithm

Existing Resource (E)

10

Available Resource (A)

3

Current Allocation (C)

T_1	3
T_2	2
T_3	2

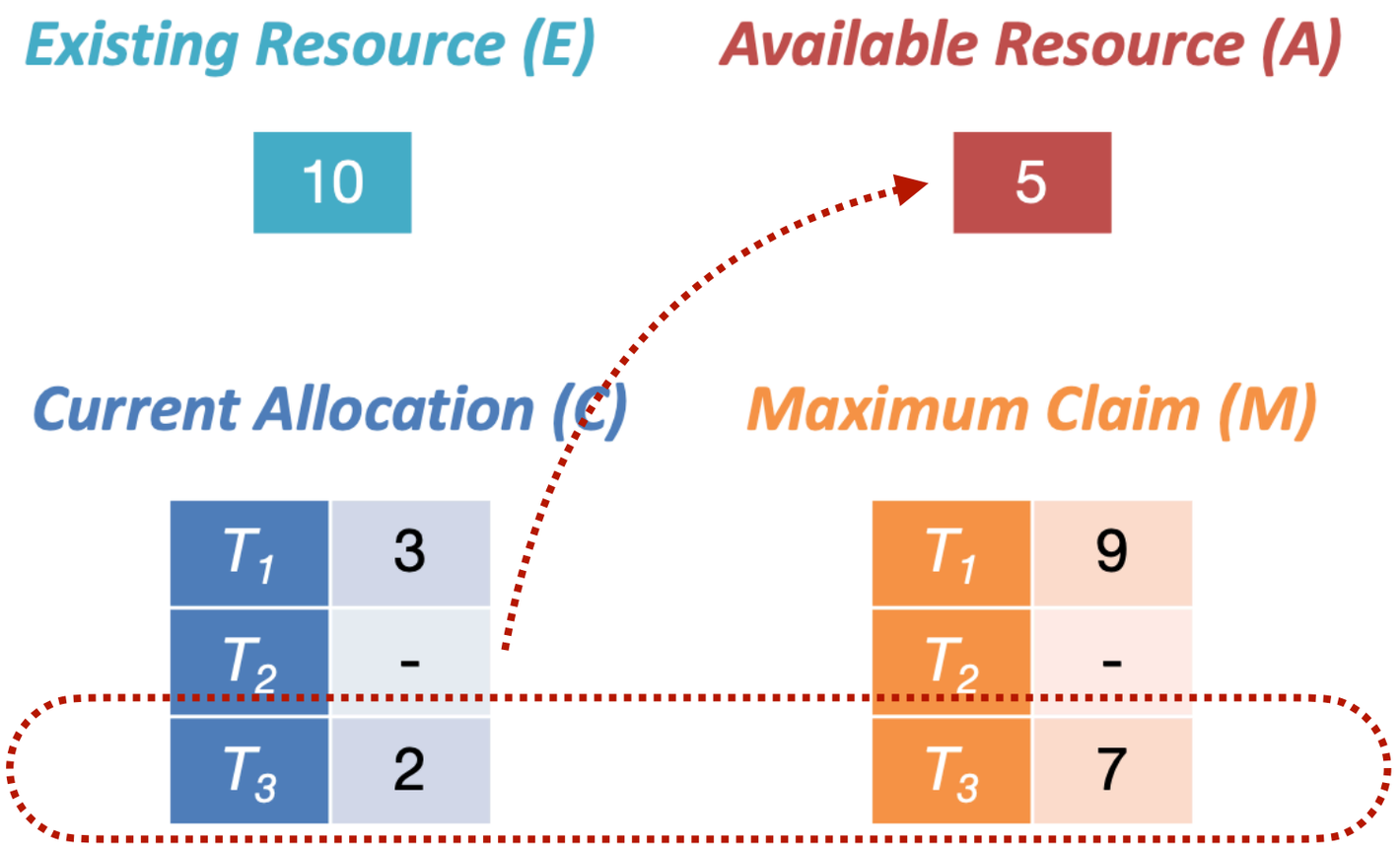
Maximum Claim (M)

T_1	9
T_2	4
T_3	7

The maximum claim of T2 can be satisfied (need 2 more resources)

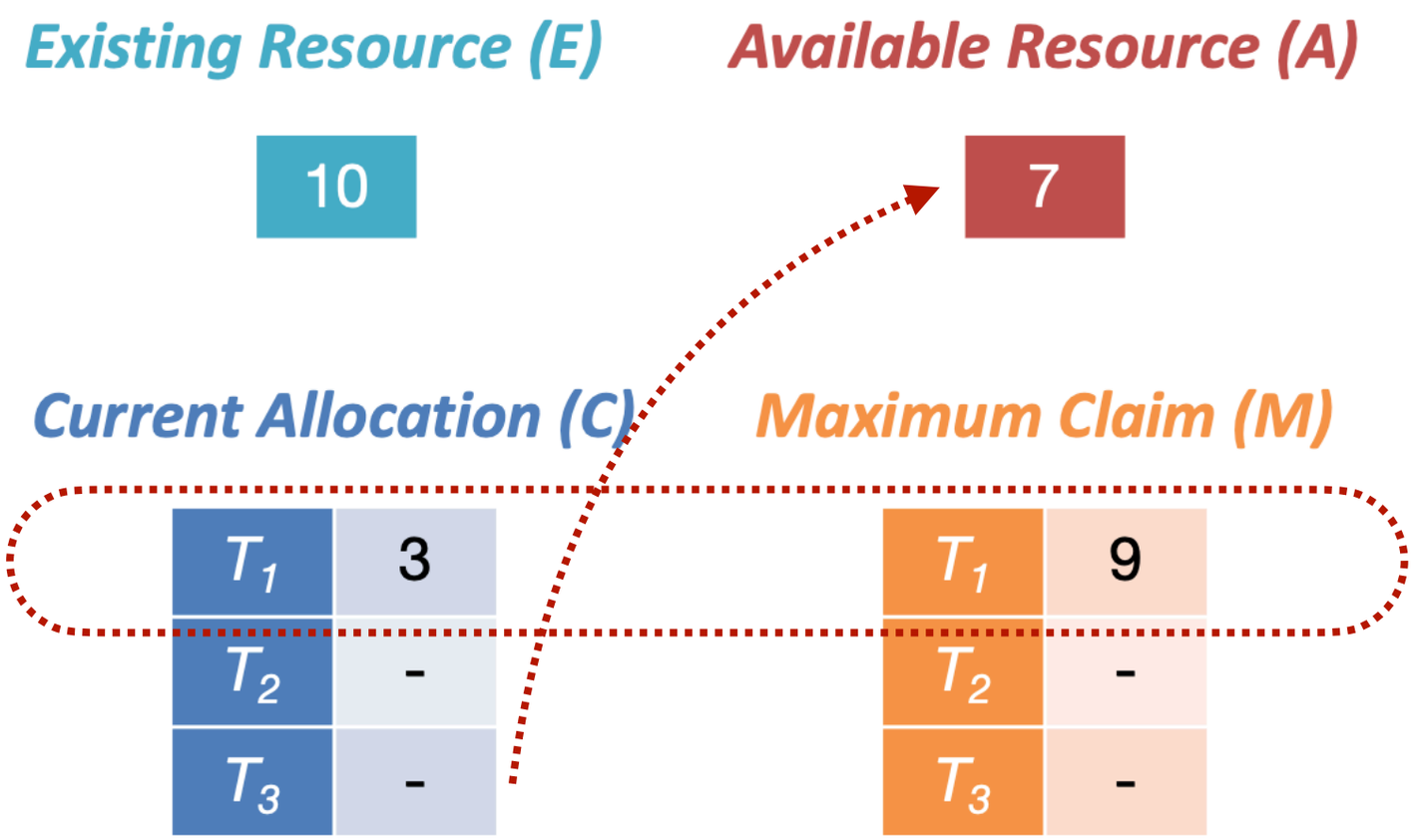
Is this current state safe?

Banker's Algorithm



The maximum claim of T3 can be satisfied (need 5 more resources)

Banker's Algorithm



The maximum claim of T1 can be satisfied (need 6 more resources)

Banker's Algorithm

Existing Resource (E)

10

Available Resource (A)

10

Current Allocation (C)

T_1	-
T_2	-
T_3	-

Maximum Claim (M)

T_1	-
T_2	-
T_3	-

Safe state

Banker's Algorithm

Existing Resource (E)

10

Available Resource (A)

3

Current Allocation (C)

T_1	3
T_2	2
T_3	2

Request (R)

T_1	1
T_2	0
T_3	0

Maximum Claim (M)

T_1	9
T_2	4
T_3	7

Can we grant this request?

Banker's Algorithm

Suppose we grant it

Are we still in a safe state?

Banker's Algorithm

Existing Resource (E)

10

Available Resource (A)

4

Current Allocation (C)

T_1	4
T_2	-
T_3	2

Request (R)

T_1	1
T_2	0
T_3	0

Maximum Claim (M)

T_1	9
T_2	-
T_3	7

Banker's Algorithm

Existing Resource (E)	*Available Resource (A)*
10	4

	Current Allocation (C)	*Request (R)*	*Maximum Claim (M)*
T_1	4	1	9
T_2	-	0	-
T_3	2	0	7

Both Unsatisfied

Unsafe state

But not deadlock at this time point

Banker's Algorithm

虽然能通过总是让系统处于安全 (safe) 状态来避免死锁，但该算法通常无法在实际系统中应用 (useless in practice)

- 系统中的资源个数、以及线程个数需要提前知道并且保持不变
- 每个线程需要指明其对各类资源的请求上限
- 线程之间需要相互独立 (without synchronization)

Quiz

假设某系统中有 A 和 B 两种资源，当前的已分配资源数目、每个线程所需的最大资源数、以及可用资源数如下表所示

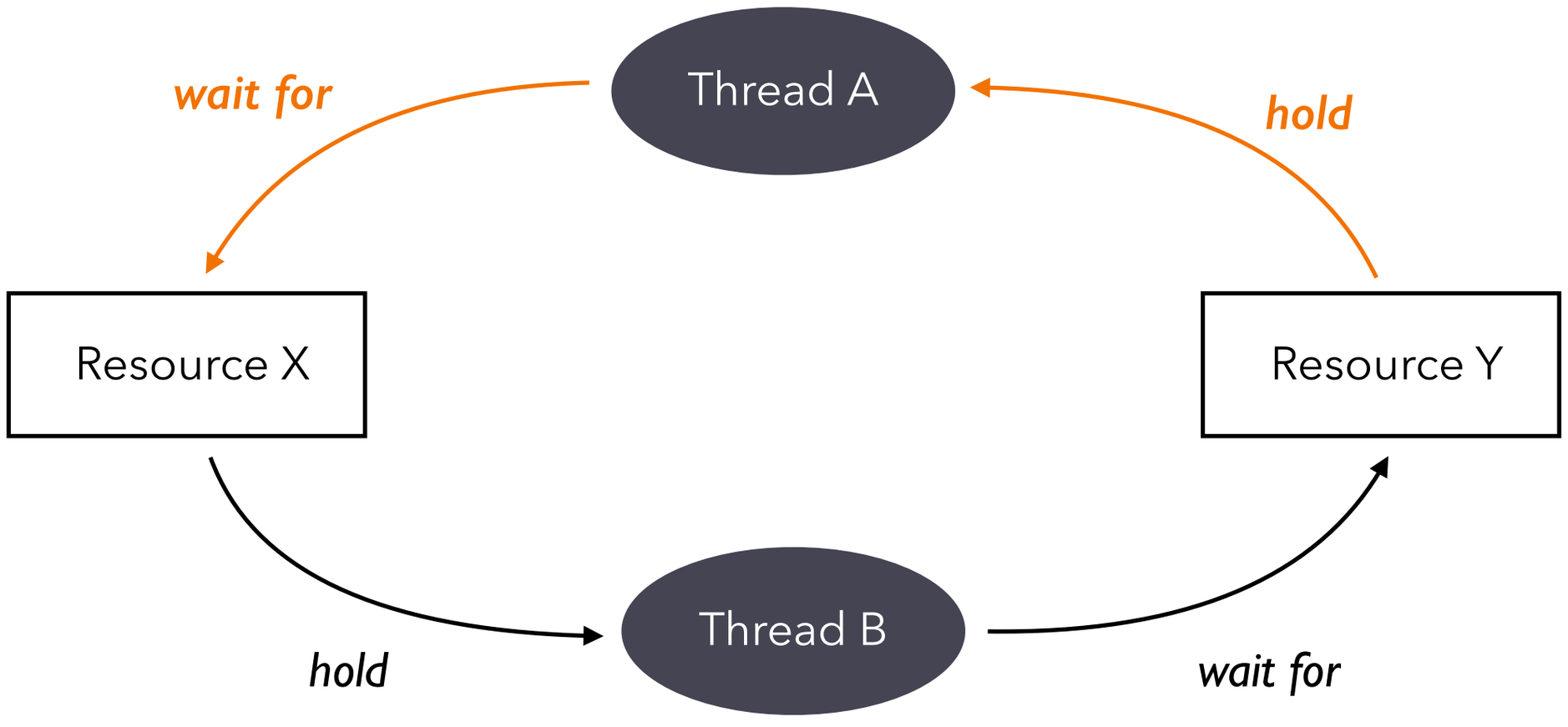
- 当前系统是否处于安全状态?
- 假设此时 $T3$ 向系统申请 2 个 B 资源，此时是否能满足其需求?

	Current Allocation		Maximum Claim		Available Resource	
	A	B	A	B	A	B
T_1	0	3	4	6	2	5
T_2	2	0	4	1		
T_3	1	0	4	8		
T_4	7	1	12	5		

死锁检测和恢复

对于无法完全避免死锁 (infeasible or too expensive) 以及死锁不频繁出现的场景，另一种思路是允许死锁出现，并在其出现时进行恢复

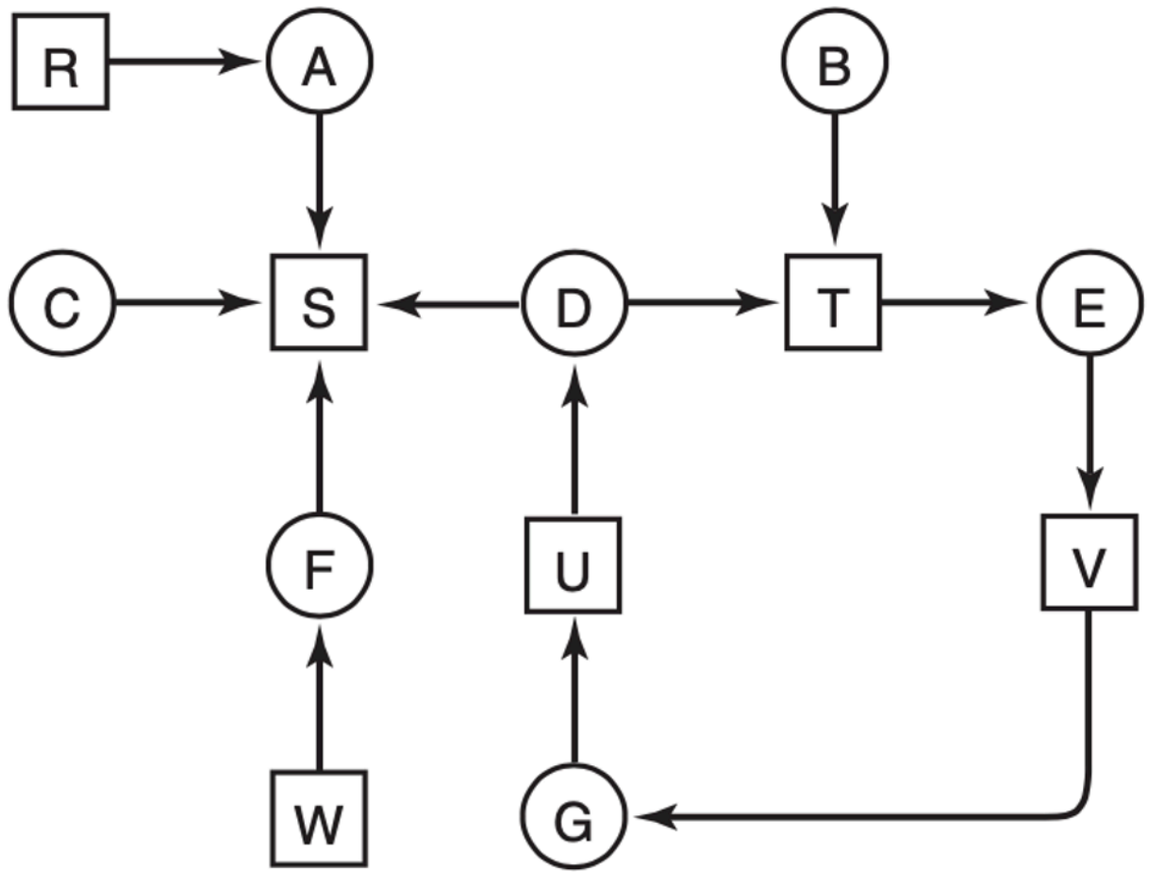
- 利用资源分配图 (resource allocation graph) 来检测死锁



Deadlock: a circle in the graph

死锁检测和恢复

- 死锁检测问题就转换为 "判断资源分配图中是否存在环" 的问题



For example, depth-first search

Start from node R

- $L = \{R, A, S\}$ *[dead end]*
- $L = \{R, A\}$ *[go back]*
- $L = \{R\}$ *[terminate]*

Start from node A

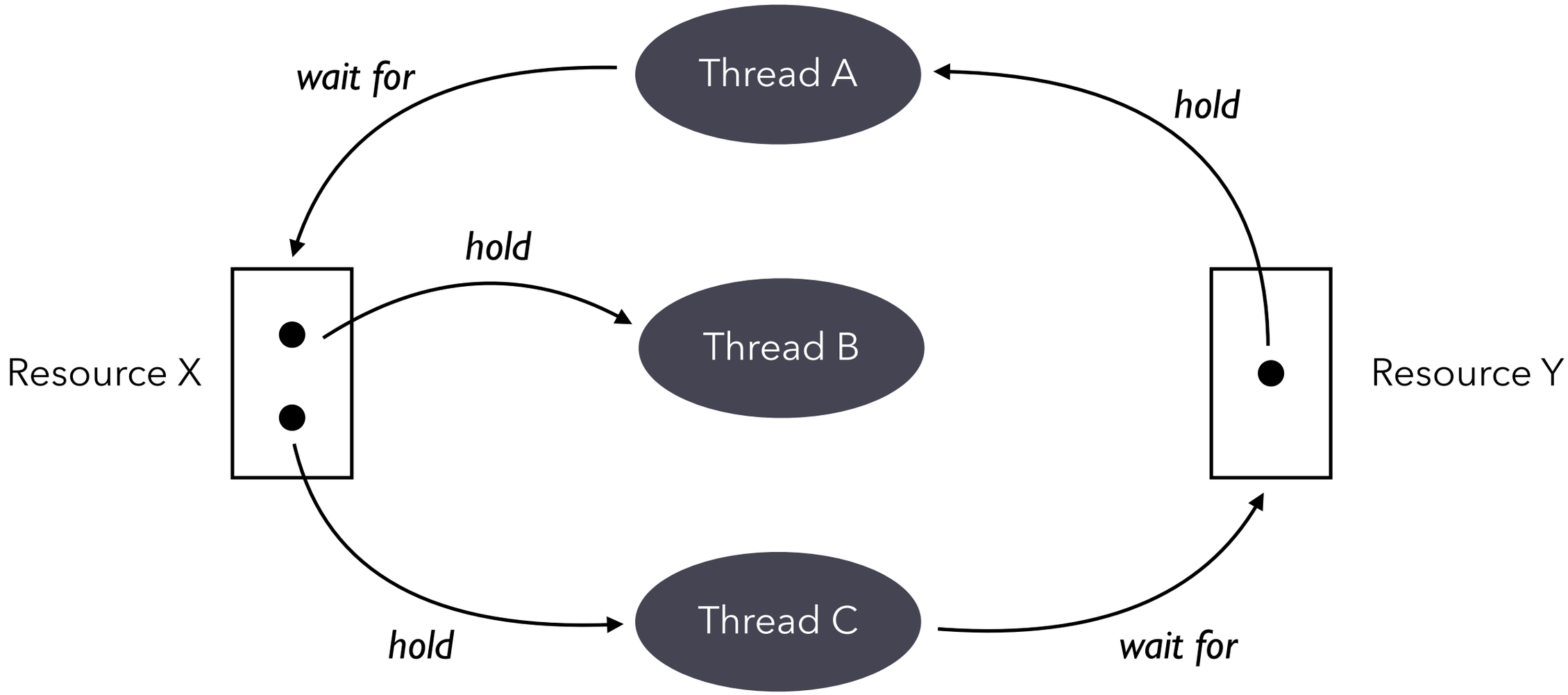
- $L = \{A, S\}$

Start from node B

- $L = \{B, T, E, V, G, U, D\}$
- $L = \{B, T, E, V, G, U, D, S\}$
- $L = \{B, T, E, V, G, U, D\}$ *[go back]*
- $L = \{B, T, E, V, G, U, D, T\}$ *[circle]*

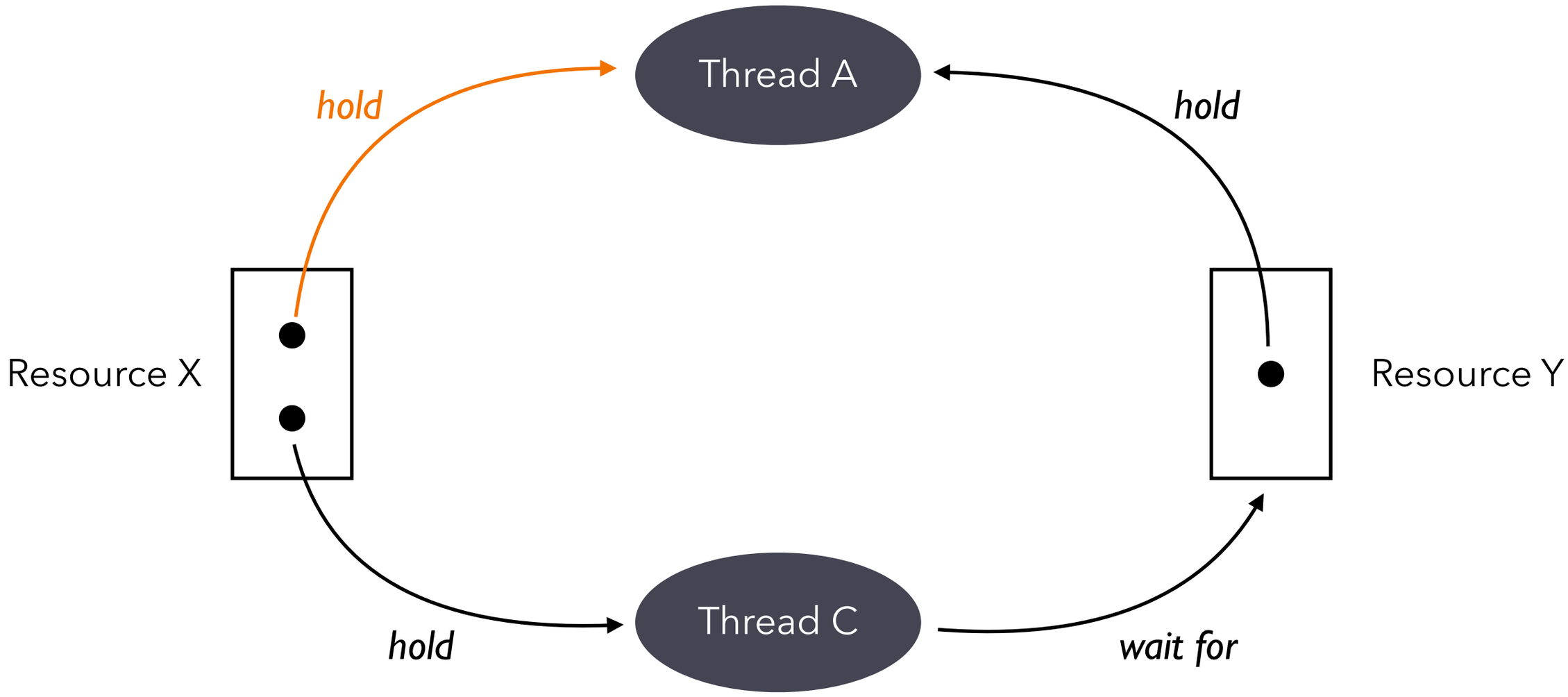
死锁检测和恢复

- 如果系统中的同一类资源存在多个实例，则资源分配图中存在环就仅是死锁的必要条件 (此时可以通过 “化简” 资源分配图来判断是否死锁)



死锁检测和恢复

- 如果系统中的同一类资源存在多个实例，则资源分配图中存在环就仅是死锁的必要条件 (此时可以通过 “化简” 资源分配图来判断是否死锁)



死锁检测和恢复

Existing Resources

X	Y
2	1

Available Resources

X	Y
0	0

Hold Matrix (Current Allocation)

	X	Y
Thread A	0	1
Thread B	1	0
Thread C	1	0

Wait For Matrix (Current Request)

	X	Y
Thread A	1	0
Thread B	0	0
Thread C	0	1

这里的死锁检测看的是系统的 Current Request 能否被满足，银行家算法看的是 Maximum Claim - Current Allocation (request of the worst case) 能否被满足

死锁检测和恢复

虽然可以检测系统是否产生死锁，但想从死锁状态中恢复并不容易

- 通过资源抢占 (Resource Preemption) 的方式?
  - 不是每种资源都可以被强制抢占的
- 通过回滚和重试 (Rollback and Retry) 的方式?
  - 定期创建检查点 (Checkpoint) 来记录系统当前的状态，在死锁产生时让系统回退到该状态
  - 增加系统设计的复杂度、丢失检查点后的信息、并且仍然存在活锁 (live lock) 的问题

死锁检测和恢复

虽然可以检测系统是否产生死锁，但想从死锁状态中恢复并不容易

- 通过终止部分线程 (Kill Threads) 的方式?
  - 被终止的线程将释放其持有的资源
  - 终止哪个线程?
    - 线程的优先级
    - 线程已经执行了多长时间、以及还需多长时间能执行结束
    - 线程当前持有多少资源、以及还需要请求多少资源
    - 终止该线程是否会产生其他副作用
    - …

总结

- 哲学家就餐问题 (Dining Philosophers Problem)
- 死锁的必要条件
- 应对死锁的方法
  - 避免死锁 (Preventing Deadlocks)
    - 预防死锁 (Avoiding Deadlocks)
      - 银行家算法 (Banker's Algorithm)
    - 死锁检测和恢复 (Detecting and Recovering from Deadlocks)