并发问题
Concurrency Problems

钮鑫涛
南京大学
2024春

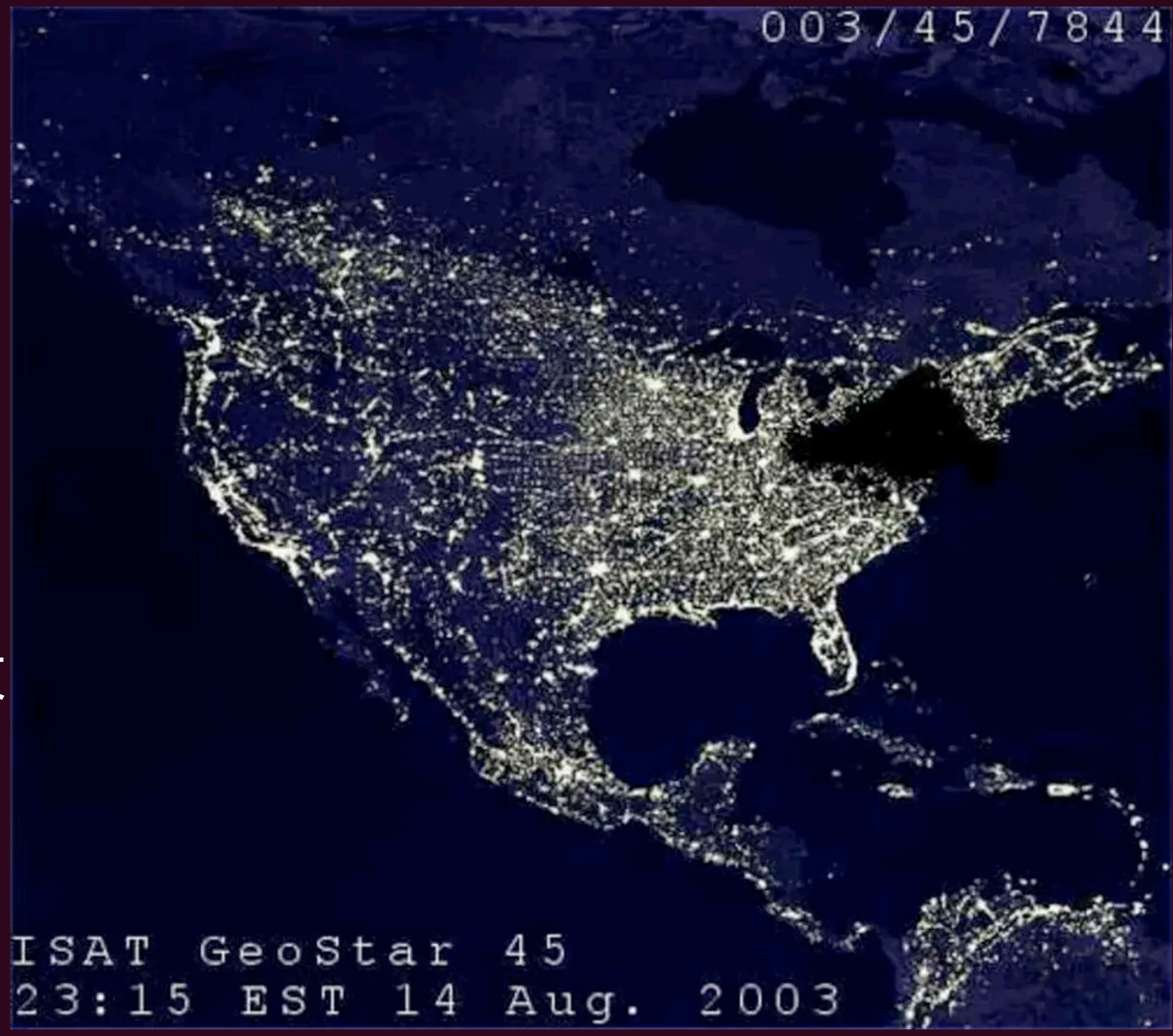
本门课由蒋炎岩和钮鑫涛共同构建

现实中的并发问题

- Therac-25 Incident (1985-1987)
- 存在一个并发bug使病患接受到比正常剂量高一百倍的辐射，因此造成患者死亡或重伤（至少6位患者死于这个bug）
  - Therac-25之前的机型有一个硬件锁避免此情况，但Therac-25取消了这个机制，取而代之软件锁，但该锁会由于一个数据竞争(竞态条件)而失效

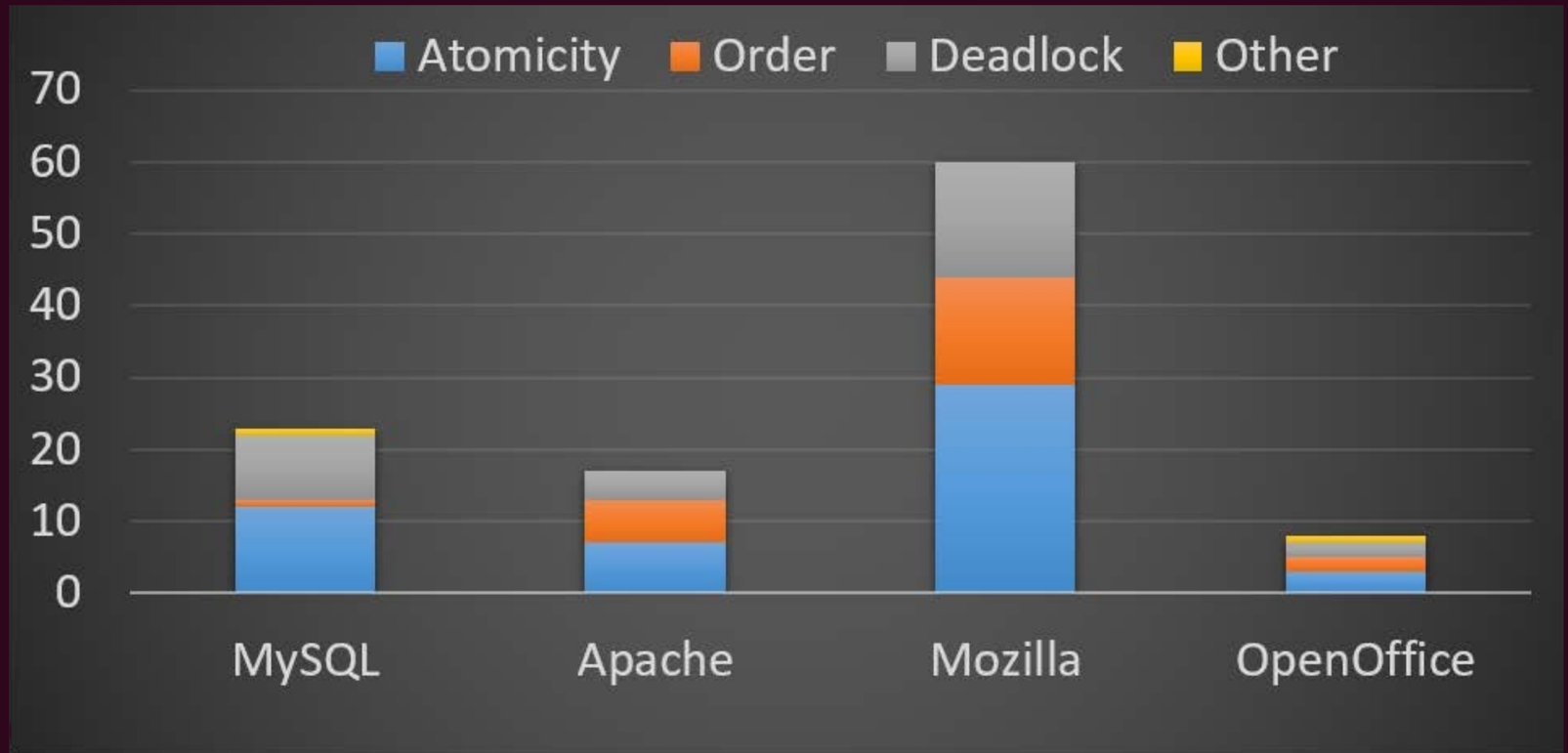
现实中的并发问题

- 美加大停电，估计经济损失250 亿—300 亿美元。
- 原因由于能源管理系统中的“竞争条件”触发导致
  - 代码从1990年就运行，但2003年才出发了这个微妙的bug



并发bug的经验性调研

- Shan Lu等人的文章(ASPLOS'08, Most Influential Paper Award)调研了现实中软件（4个代表性的大型软件）的并发bug的组成
  - 经验性研究往往更能体现“人类”的倾向



并发控制的机制的前提是正确使用

- 我们学了这么多并发控制的工具
  - 互斥锁、条件变量、信号量...
  - 但还是写不好并发程序!
- 这些同步工具你得正确使用才行，然而我们人类还是会犯很多“低级”错误，比如下面两个代表性的错误：

上错了锁

```
void T_1() { spin_lock(&A); sum++; spin_unlock(&A); }
void T_2() { spin_lock(&B); sum++; spin_unlock(&B); }
```

忘记上锁

```
void T_1() { spin_lock(&A); sum++; spin_unlock(&A); }
void T_2() { sum++; }
```

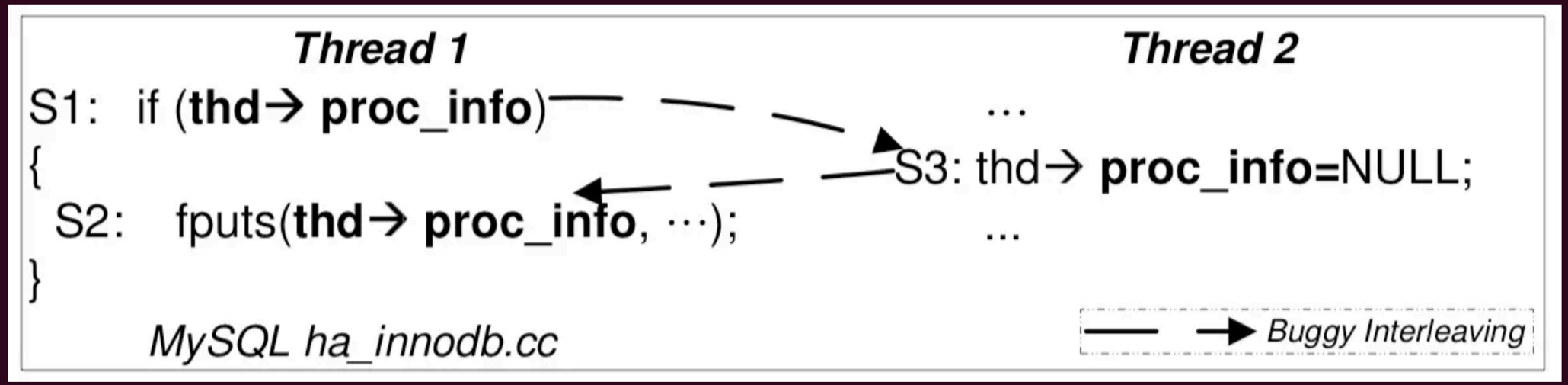
原子性和顺序性bug

并发控制的机制的前提是正确使用

- 根据之前的调研，除了死锁之外，我们犯的大部分（97%）并发错误都可归类为如下两种：
  - 原子性违背(Atomicity Violation, AV)
    - 比如忘记上锁
  - 顺序性违背(Order Violation, OV)
    - 忘记同步，或者同步条件写错

原子性bug

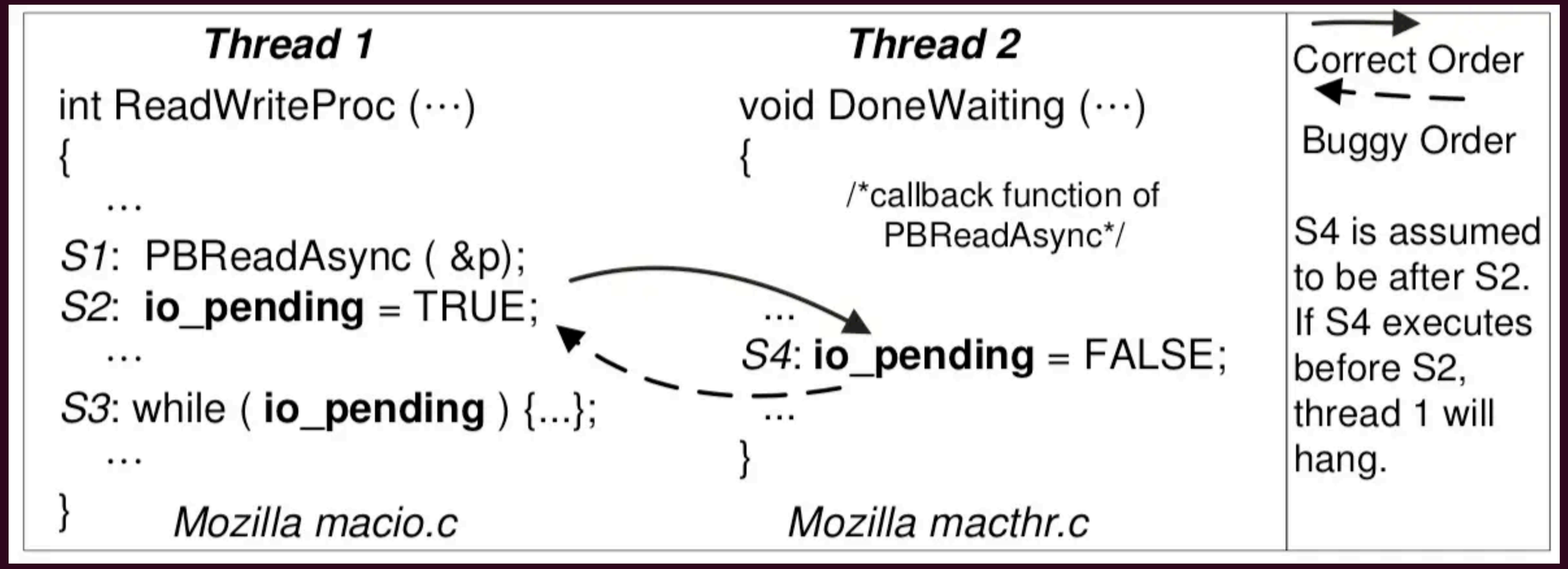
- “ABA”： 代码被别人 “强势插入”
  - 有时即使你意识到了需要上锁，并且也加锁了，也会犯错



从check->operation整体都应该是原子的，部分分别原子并不能保证安全

顺序性违反

- “BA”: 事件未按预定的顺序发生



checking和setting的顺序会在一个不幸运的调度下违背了期待的顺序

死锁

死锁

- 在并发计算中，死锁是指某个小组中的成员因为每个成员都在等待另一个成员（包括自己）采取行动，比如发送消息或释放锁，因此无法继续执行的状态。

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.
—Kansas state law, early 1900s

Self-deadlock（A-A型死锁）

- 一个线程在已经持有一把锁的情况下再次尝试获得这把锁
  - 看起来很傻?
  - 然而真实系统的复杂性很容易让人懵圈
    - 多层函数调用
    - 隐藏的控制流
- 解决方案：xv6防御性编程

```
void foo(){
    lock();
    foo();
    unlock();
}
foo();
```

```
// Acquire the lock.
void acquire(struct spinlock *lk){
  ...
  if(holding(lk))
    panic("acquire");
  ...
```

更一般的死锁

- 一个涉及多个线程以及多个锁的情形，每个线程都在等待其他线程所持有的锁，从而没有一个可以行进
  - 一个典型案例就是ABBA-型死锁(mutually recursive locking)：线程1拿到了锁A，需要锁B，线程2拿到了锁B，需要锁A，这时两个线程都无法行进
  - 哲学家进餐问题的第一方案就是这种类型的死锁

```
void T1(){
    lock(&A);
    lock(&B);
    dosomething();
    unlock(&B);
    unlock(&A);
}
```

```
void T2(){
    lock(&B);
    lock(&A);
    dosomething();
    unlock(&A);
    unlock(&B);
}
```

死锁的样例

- 条件变量和锁的使用也会产生死锁—> 等待某个条件变量看成是等待某个资源，而释放的那个线程看成是持有某个资源！

```
void T1(){
        lock(&A);
        lock(&B);
        while (need-to-wait)
          wait(&cv, &B);
        unlock(&B);
        unlock(&A);
}
```

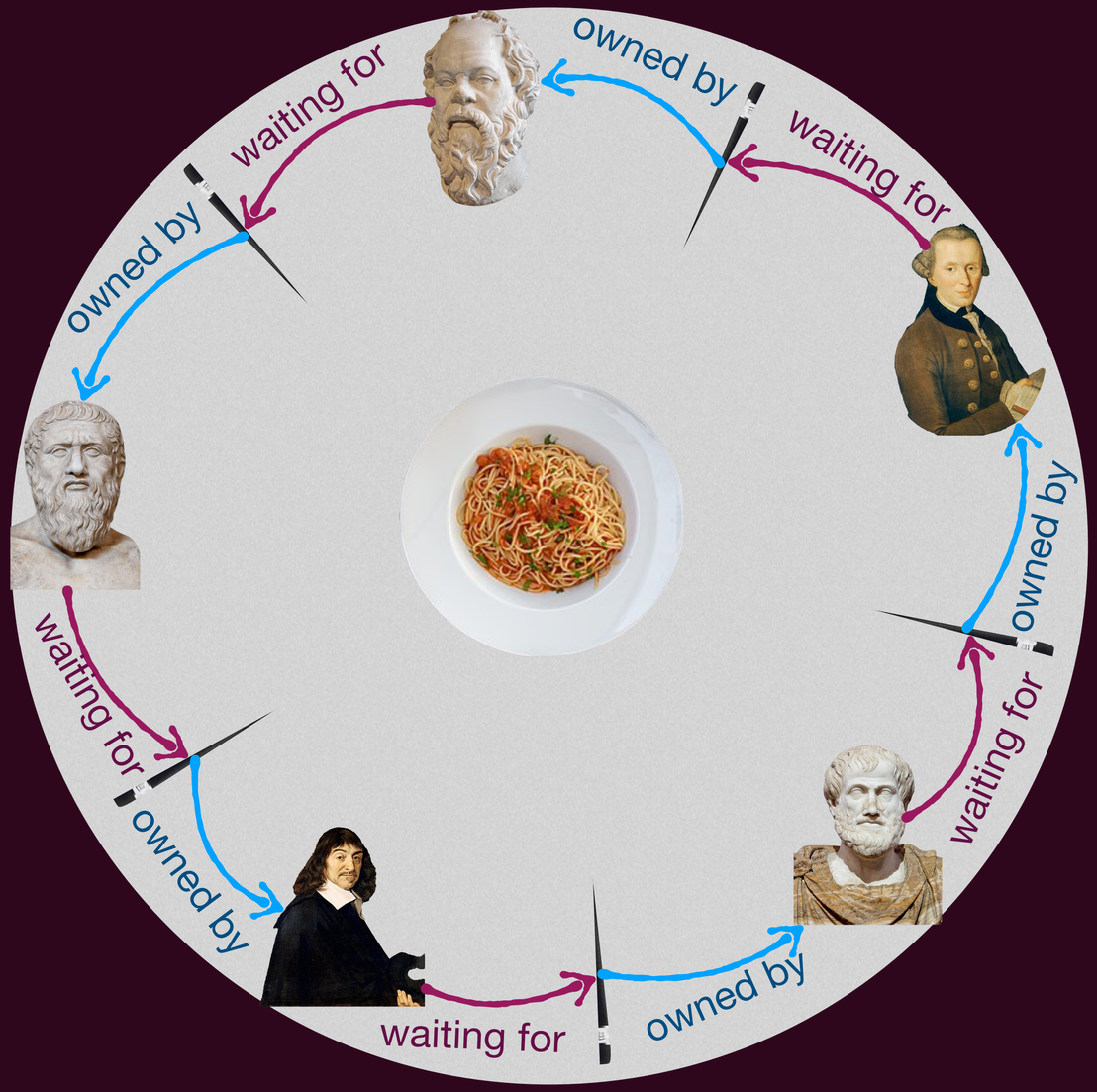
while(need-to-wait) {unlock(&A); wait(&cv, &B); }

```
void T1(){
        lock(&A);
        lock(&B);
        signal(&cv);
        unlock(&B);
        unlock(&A);
}
```

死锁产生的必要条件

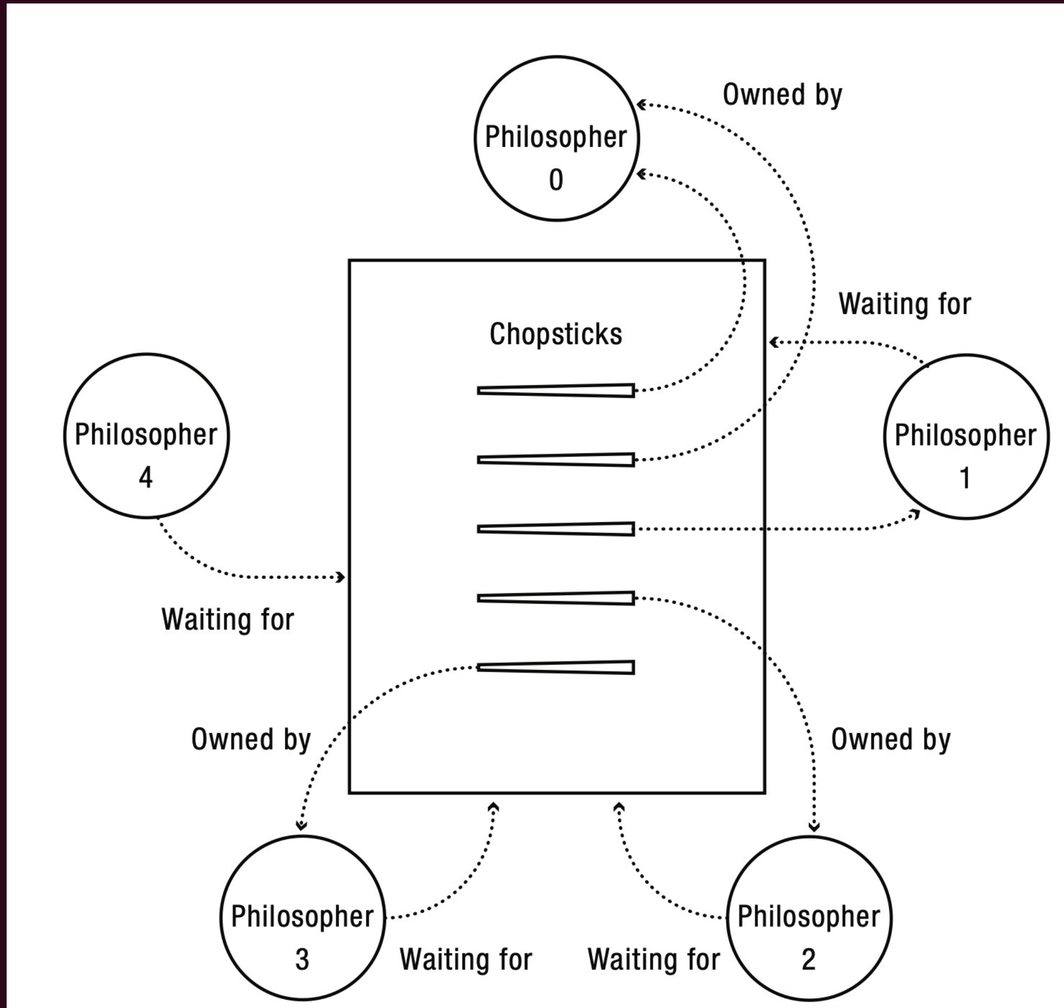
- 死锁的四个必要条件(by Coffman et al. 1971)：
  - Mutual-exclusion — 所需要的资源是互斥的
    - 互斥也可改为资源有限（Bounded resources），即能够共享的线程数有限
    - 当然，Bounded resources也可以看成是多个同种“互斥”的资源
  - Hold and wait — 持有某个资源并等待更多资源
  - No-preemption — 不可直接“抢”别人持有的资源，只有等持有的人主动释放
  - Circular wait condition — 形成循环等待的环

哲学家问题解决方法1满足条件



注意这是“必要条件”

- 达成条件也不一定就一定死锁
  - 比如对于同一种类型的多个筷子
  - 环之外的线程可以释放资源解开这个环等待状态



处理死锁

- **忽略问题**
  - 在问题的发生频率很低且发生后代价很小（重启大法）直接忽略(Ostrich Algorithm).
- **从源头避免死锁发生**
  - **Prevention**, by **structurally** negating one of the four required conditions.
  - **Dynamic avoidance** by careful resource allocation.
- **检测并恢复**
  - **Let deadlocks occur, detect them, and take action.**

死锁避免：必要条件的破坏

- 想法：死锁的四个必要条件，破快其一即可！
  - 互斥的破快?
    - 不太容易，互斥是并发能够保证安全性的重要机制！
    - 不过人类还是开发出了一系列无锁算法：如RCU、一些基于硬件支持（如CAS原子指令）的数据结构和算法
    - 但写出这样无锁的算法涉及非常小心的指令顺序，非常容易出错

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

无锁的加某个值

死锁避免：必要条件的破坏

- 持有并等待的破坏
  - 要么能一次得到所有锁，要么什么锁也不获取
    - 不现实，因为这需要线程提前就知道需要哪些锁，如果是后面系统根据需要才新建一些锁，那么其难以提前获知
    - 此外，这也减少了一些并发度，因为如果程序很长时间运行中其实不太需要某些锁（只有在中间某段很短的时间内才需要某个锁），那么一开始就持有这把锁会限制其他线程得到这把锁并运行

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

死锁避免：必要条件的破坏

- 持有并等待的破坏
  - 尝试2: 如果此刻无法获得想要的锁，就释放其所有持有的锁

但如果锁是被封装在某个地方（不是显式地调用lock所得），便很难察觉需要释放这个锁

```
void philosopher(int i){
  while(1){
    think();
    while(1){
        P(forks[left(i)]);// Pick up left fork
        int ret = tryP(forks[right(i)]);//Try to pick up right fork;
        if (ret!= 0){
             V(forks[left(i)]);//  Put down left fork
             sleep(randomtime);
        }else{
            break;
        }
    }
    eat();
    V(forks[left(i)]);// Put down left fork
    V(forks[right(i)]);//Put down right fork;
  }
}
```

这里使用random的睡眠时间是防止出现同时睡同时醒的状况，这样各个线程会非常“尴尬的”无法行进，也即活锁（livelock）

死锁避免：必要条件的破坏

- 非抢占性的破坏
  - 允许系统直接去“抢”别的线程所持有的资源
    - 比如说一个线程需要更多的内存，但此时没有更多free的内存了，那么它可以去“抢”别的线程的内存：将别的线程先暂时移出内存到磁盘上，然后再去得到这些内存
    - 比如CPU的使用也是可以被抢占（利用context-switch机制）
  - 问题是不是每种资源都可以直接“抢”的
    - 比如打印机资源，如过一个打印机正在打印，那么现在直接去抢?
    - 一个合理的“抢”需要被抢走资源的线程能够恢复到被抢前的状态

死锁避免：必要条件的破坏

- 环形等待条件破快
  - 强制在锁在申请时按照规定的顺序来（Lock ordering）
    - 比如，按照锁在内存中的位置进行编号排序

```
void do_something(mutex t *m1, mutex t *m2) {
       if (m1 > m2) {    // grab in high-to-low address order
         pthread_mutex_lock(m1);
         pthread_mutex_lock(m2);
       } else {
         pthread_mutex_lock(m2);
         pthread_mutex_lock(m1);
       }
}
```

死锁避免：必要条件的破坏

- 为什么按照一个全局的顺序就不会产生死锁?
  - 让我们首先回顾有向图的拓扑排序
  - 对一个有向图 G的节点的线性排序
    - 这个线性排序必须满足如果G存在一条边(u, v)，那么这个线性排序中u必须在v之前
  - 有环的图没有拓扑序
  - 无环的图一定有拓扑序

智能软件与工程学院
School of Intelligent Software and Engineering

深度优先的一些应用
Some application of DFS

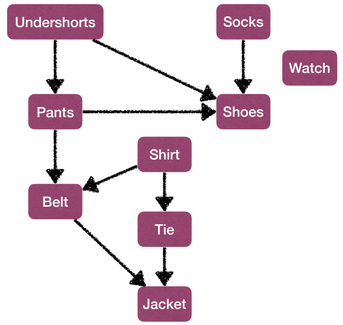
钮鑫涛
Nanjing University
2023 Fall

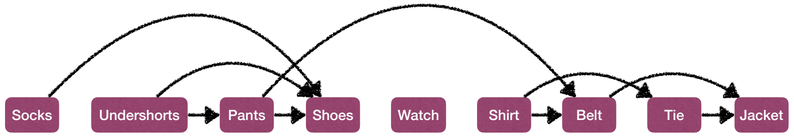
The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

智能软件与工程学院
School of Intelligent Software and Engineering

Topological Sort

- A **topological sort** of a DAG G is a linear ordering of its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.
- $E(G)$ defines a **partial order** over $V(G)$, a **topological sort** gives a **total order** over $V(G)$ satisfying $E(G)$





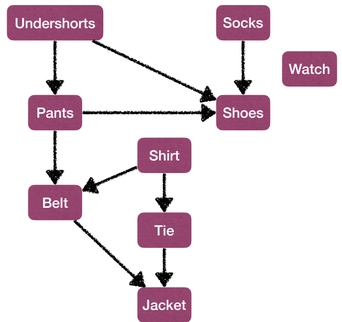
A topological ordering arranges the vertices along a horizontal line so that all edges go “from left to left”.

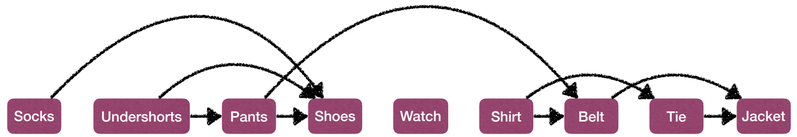
智能软件与工程学院
School of Intelligent Software and Engineering

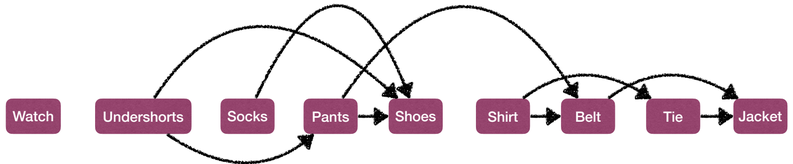
Topological Sort

- **Topological sort** is **impossible** if the graph contains a cycle.
- A given graph may have multiple different valid topological ordering.

How to generate a topological ordering?







智能软件与工程学院
School of Intelligent Software and Engineering

Topological Sort

- A topological sort of a DAG G is a linear ordering of its vertices such that if G contains an edge (u, v) then u appears before v in the ordering.
- **Q**: Does every DAG has a topological ordering?
- **Q**: How to tell if a directed graph is acyclic? If acyclic, how to do topological sort?

Lemma 1 Directed graph G is acyclic iff a DFS of G yields no back edges

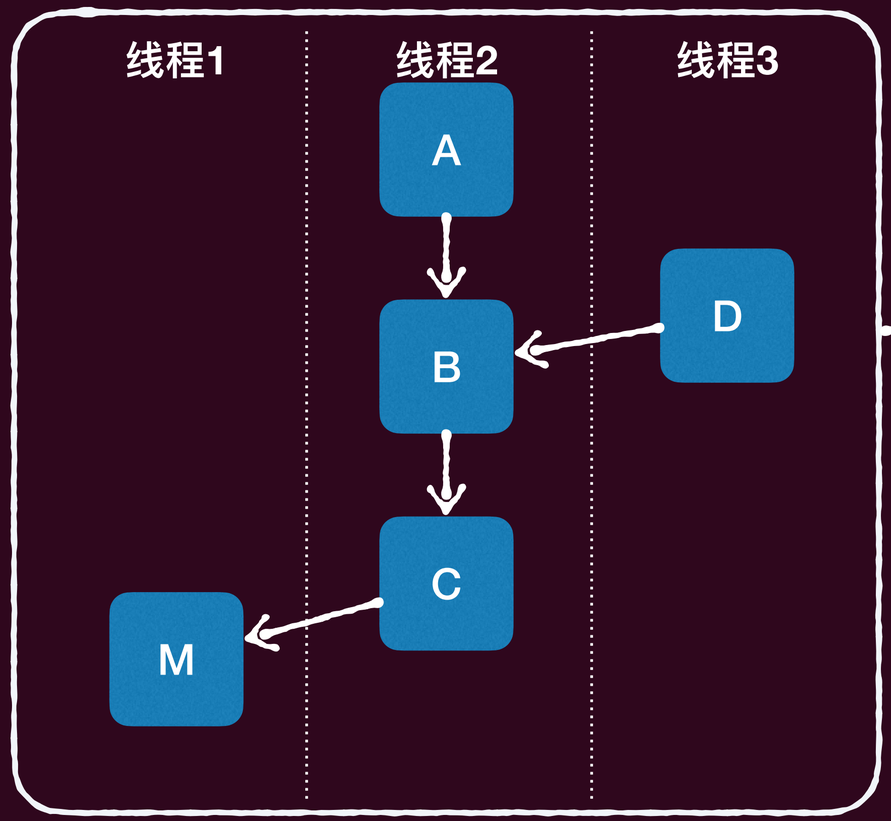
Lemma 2 If we do a DFS in DAG G, then $u.f > v.f$ for every edge (u,v) in G

Theorem Decreasing order of finish times of DFS on DAG gives a topological ordering

Corollary Every DAG has a topological ordering

死锁避免：必要条件的破坏

- 为什么按照一个全局的顺序就不会产生死锁?
  - 锁的申请过程就是构造一个关于锁的有向图G的过程



一个结论：将这个图中的锁按照编号输出就是这个图的一个拓扑序！

➡因为每“大”一点的锁一定都在“小”一点的锁之后，而这个图中的每条边都是“小”一点的锁指向“大”一点的锁

有拓扑序意味着这个锁申请有向图是无环的！
（注意甚至锁的编号不一定要有一个全序都可以做到这一点，只要有偏序关系即可）

死锁避免：必要条件的破坏

- 这个策略被现实操作系统广泛采用
  - 比如：Linux Kernel: mm/rmap.c

```
/*
 * 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)
```

死锁避免：必要条件的破坏

- 然而即使是lock ordering任然有其局限性

Unreliable Guide to Locking: 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 doesnt scale: when I create a new lock, I dont 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 nontrivial 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 dont even need to know you are using a lock.

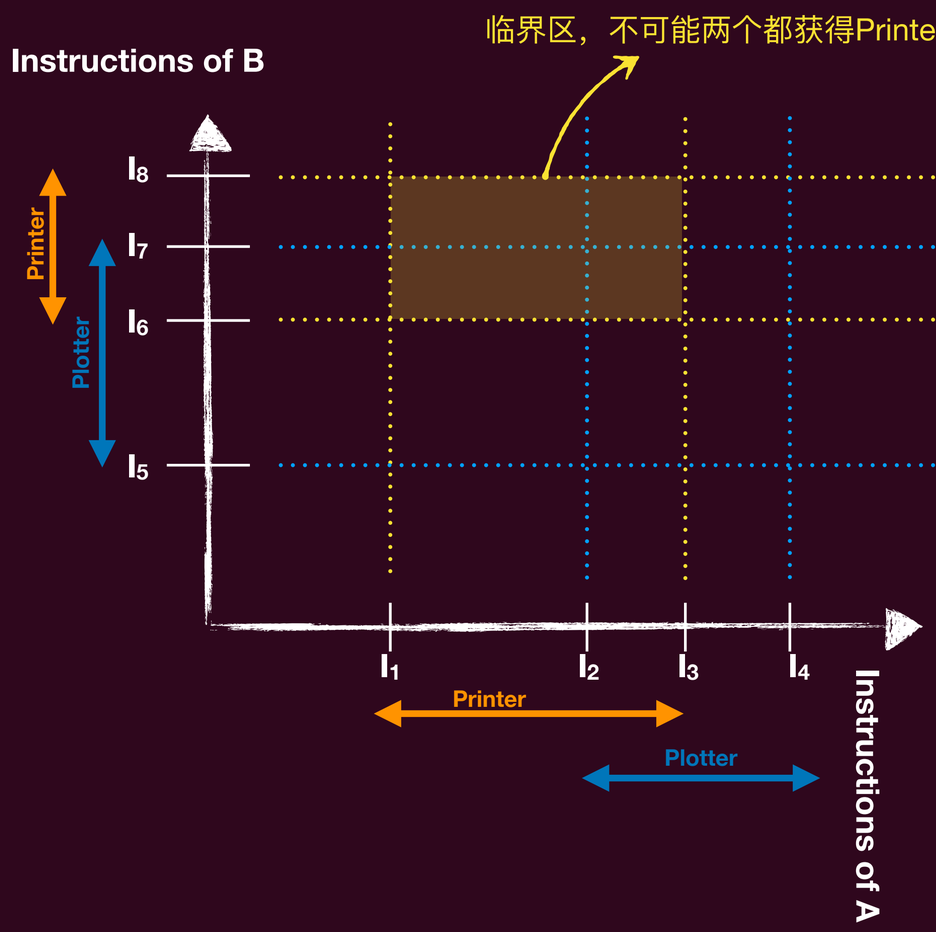
死锁避免：避免陷入不可挽回状态

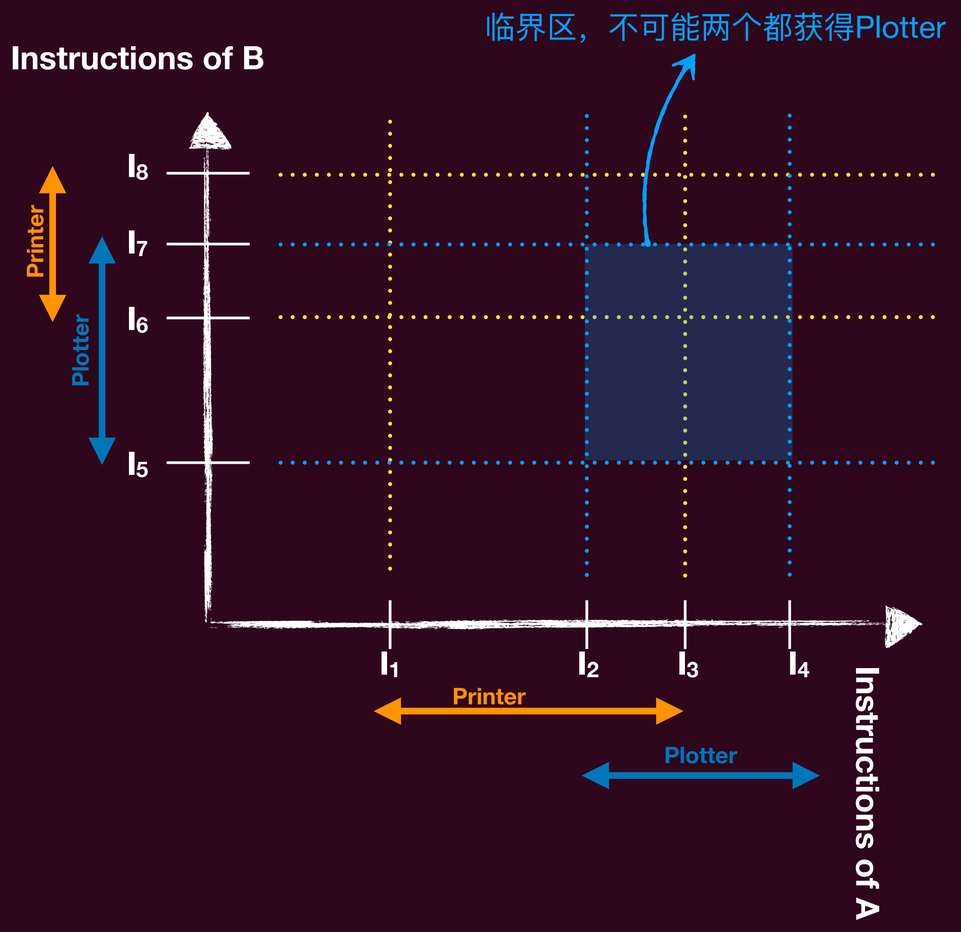
- 死锁关乎程序的活性（liveness）性质
- 回顾活性：
  - 活性: “**好事终将**发生”：
  - 活性要求只要在**最终**能满足要求即可，一个隐含的要求是执行中不能发生“不可挽回”的步骤！
- 什么时候是导致死锁的“不可挽回”事件?

导致死锁的“不可挽回”事件

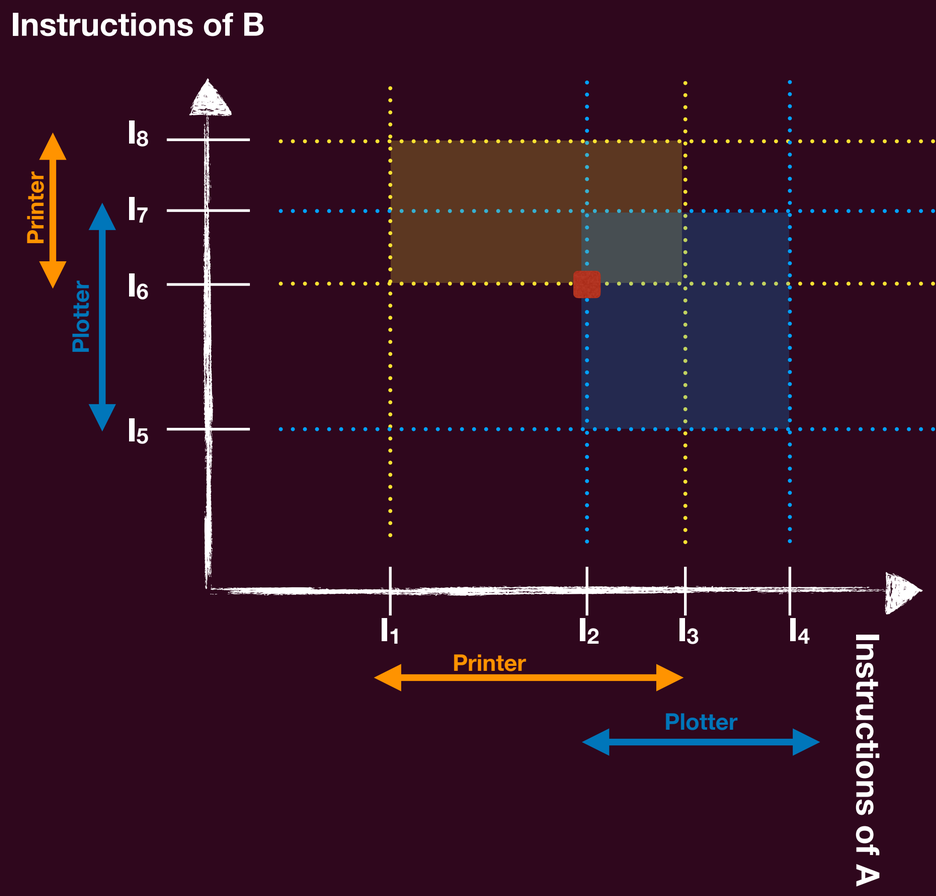
- 让我们从一个简单例子
  - 有两个线程A和B，每个线程都需要用到需要如下的两个互斥的资源：打印机和绘图仪
  - 线程A从指令I_1到I_3之间需要打印机，从指令I_2到I_4之间需要绘图仪
  - 线程B从指令I_5到I_7之间需要绘图仪，从指令I_6到I_8之间需要打印机

导致死锁的“不可挽回”事件





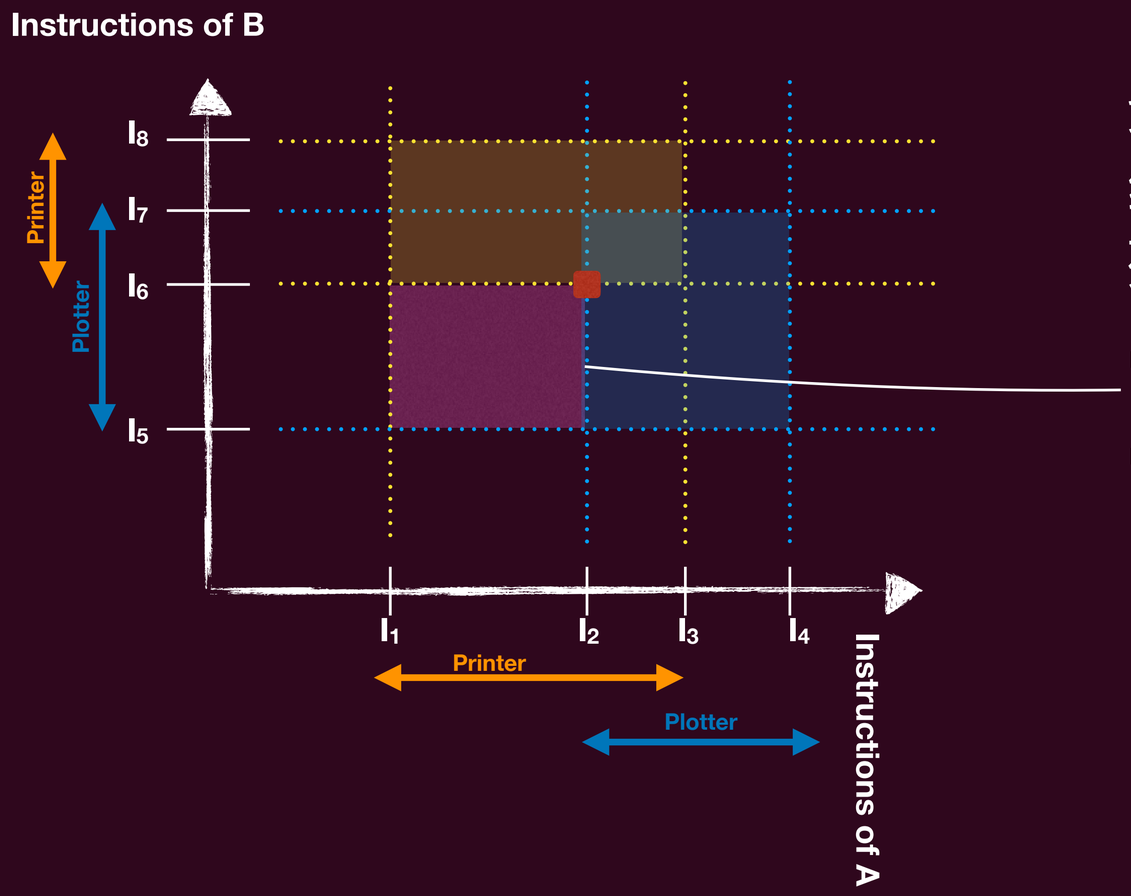
导致死锁的“不可挽回”事件



- 考察如下红点处(I_2, I_6)，试问如果系统状态处于该点（即线程A得到了Printer，请求Plotter，线程B得到了Plotter，请求Printer）
  - 典型的ABBA型死锁，系统无法再行进
  - 从图上也可知道此时系统无论向上还是向下都是“不可达”状态！

注：线程的行进无法倒退，因此整体上系统的“点”的行进向量只能是指向“第一象限”的方向

导致死锁的“不可挽回”事件



紫色区域已经是“不可挽回了”，因为处在这个区域的系统，随着整体能够行进的方向最终都会无可避免的走向死锁状态

- 只要能避免进入该不可挽回的区域就不会发生死锁！
- 这需要在系统尝试进入该区域时，调度系统直接拒绝相应的线程得到想要的资源

导致死锁的“不可挽回”事件

- 如果要按照上述的情形来给出不可挽回区域，你必须提前知道每个线程在未来哪段指令间需要什么锁或其他互斥资源
  - 这显然是不现实的
- 我们将条件稍稍放宽，假设我们能够知道每个线程在其整个执行期间（而不是具体到哪个指令段）需要哪些互斥资源
  - 更一般的，假设资源数不只为1（信号量），我们假设能够知道每个线程在其执行期间需要哪些互斥资源的最大数量，即每个线程i有一个最大需求向量M_i
  - 所有线程的最大需求向量形成一个最大需求矩阵M

导致死锁的“不可挽回”事件

- 此外相应的，我们也可以刻画当前系统状态的互斥资源的线程持有状态
  - 每个线程i有一个当前持有资源C_i, 所有线程的当前持有资源为矩阵C
- 系统初始每个资源数向量E
- 系统的行进伴随着线程对互斥资源的申请和释放，释放资源是平凡的事件，因为释放资源不会让系统进入“不可挽回”状态，但申请资源会
  - 每个线程i有一个当前资源申请向量R_i,所有线程的当前申请资源为矩阵R

导致死锁的“不可挽回”事件

- 什么样的资源申请矩阵会导致系统进入“不可挽回”的状态呢?
  - 如果接受了这个申请，导致无法满足未来的“最大”申请，即不可挽回
    - 即目前虽然没有死锁，可以满足，但既定的未来不可满足
    - 这个“不可挽回”状态也有人称为“不安全”状态
- 正确的决策：如果当前申请矩阵会导致系统进入“不可挽回”状态，拒绝，否则接受申请，更新系统状态
- 这就是由Dijkstra提出著名的“银行家算法”(Banker’s Algorithm)

导致死锁的“不可挽回”事件

- 银行家算法的关键—判断未来是否可以满足：
  - Step 1: 找一个线程i，看看其未来还需要的最大资源数（最大资源需求数-目前持有资源数）是否能够被目前系统尚存的资源数所满足（系统初始资源数-被所有线程所持有的资源数），如果所有线程都不能被满足，就是一个“不可挽回”的不安全状态，最终会进入死锁状态
  - Step 2: 对满足需求的线程i，标记其为未来可满足状态，即其可以在目前状态下存在一个分配方式（立即全部分配其所有资源）终止. 那么也就存在这样的一个好的局面：我们可以将其资源都回收.
  - Step 3: 在步骤2的更好的局面上，重复之前的步骤，一直到所有线程都可以终止，那么该系统状态就是“可挽回”的状态，否则就是“不可挽回”

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

每一格代表一类资源数

尚存资源向量 （初始资源-已持有资源）

1	0	2	0

线程	资源已持有矩阵			
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

我们应该满足这个请求吗?

资源申请矩阵

0	0	0	0
0	0	**1**	0
0	0	0	0
0	0	0	0
0	0	0	0

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

1	1	0	0
0	1	1	2
3	1	0	0
0	0	1	0
2	1	1	0

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

1	0	2 - 1	0

线程	资源已持有矩阵			
A	3	0	1	1
B	0	1	0 + 1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

1	1	0	0
0	1	1 - 1	2
3	1	0	0
0	0	1	0
2	1	1	0

假定满足这个需求，那么系统会进入该状态

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

1	0	1	0

线程	资源已持有矩阵			
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

在该状态下，逐个检查线程剩余需求是否能够满足

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

1	1	0	0
0	1	0	2
3	1	0	0
0	0	1	0
2	1	1	0

可以满足

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

1+1	0+1	1+0	0+1

线程	资源已持有矩阵					最大需求矩阵			
A	3	0	1	1		4	1	1	1
B	0	1	1	0		0	2	1	2
C	1	1	1	0		4	2	1	0
D	1 - 1	1 - 1	0 - 0	1 - 1	回收其资源	1	1	1	1
E	0	0	0	0		2	1	1	0

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

1	1	0	0
0	1	0	2
3	1	0	0
-	-	-	-
2	1	1	0

在该状态下，逐个检查线程剩余需求是否能够满足

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

2	1	1	1

线程	资源已持有矩阵			
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	-	-	-	-
E	0	0	0	0

最大需求矩阵			
4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

在该状态下，逐个检查线程剩余需求是否能够满足

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

1	1	0	0
0	1	0	2
3	1	0	0
-	-	-	-
2	1	1	0

可被满足，重复之前操作

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

5	1	2	2

线程	资源已持有矩阵			
A	-	-	-	-
B	0	1	1	0
C	1	1	1	0
D	-	-	-	-
E	0	0	0	0

最大需求矩阵			
4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

在该状态下，逐个检查线程剩余需求是否能够满足

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

-	-	-	-
0	1	0	2
3	1	0	0
-	-	-	-
2	1	1	0

可被满足，重复之前操作

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

5	2	3	2

线程	资源已持有矩阵			
A	-	-	-	-
B	-	-	-	-
C	1	1	1	0
D	-	-	-	-
E	0	0	0	0

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

在该状态下，逐个检查线程剩余需求是否能够满足

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

-	-	-	-
-	-	-	-
3	1	0	0
-	-	-	-
2	1	1	0

可被满足，重复之前操作

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

6	3	4	2

线程	资源已持有矩阵			
A	-	-	-	-
B	-	-	-	-
C	-	-	-	-
D	-	-	-	-
E	0	0	0	0

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

在该状态下，逐个检查线程剩余需求是否能够满足

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-
2	1	1	0

可被满足，重复之前操作

银行家问题样例

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

6	3	4	2

线程	资源已持有矩阵			
A	-	-	-	-
B	-	-	-	-
C	-	-	-	-
D	-	-	-	-
E	-	-	-	-

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-

资源申请矩阵

0	0	0	0
0	0	**1**	0
0	0	0	0
0	0	0	0
0	0	0	0

接受该资源申请后，存在一个方案使得所有线程都满足未来申请需求，因此该申请可以被接受！

银行家问题样例2

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

1	0	2	0

线程	资源已持有矩阵			
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

资源申请矩阵

0	0	0	0
0	0	**1**	0
0	0	0	0
0	0	**1**	0
0	0	0	0

我们应该满足这个请求吗?

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

1	1	0	0
0	1	1	2
3	1	0	0
0	0	1	0
2	1	1	0

银行家问题样例2

初始资源向量（即系统资源总数）

6	3	4	2

尚存资源向量 （初始资源-已持有资源）

1	0	2 - 2	0

线程	资源已持有矩阵			
A	3	0	1	1
B	0	1	0 + 1	0
C	1	1	1	0
D	1	1	0 + 1	1
E	0	0	0	0

最大需求矩阵

4	1	1	1
0	2	1	2
4	2	1	0
1	1	1	1
2	1	1	0

每个线程未来需要的资源矩阵（最大需求矩阵-已持有资源矩阵）

1	1	0	0
0	1	1 - 1	2
3	1	0	0
0	0	1 - 1	0
2	1	1	0

假定满足这个需求，那么系统会进入该状态
然而未来需求矩阵中没有一个线程的需求可以被满足：这是一个“不可挽回”状态
因此该需求不应该被满足

银行家算法是不实用的

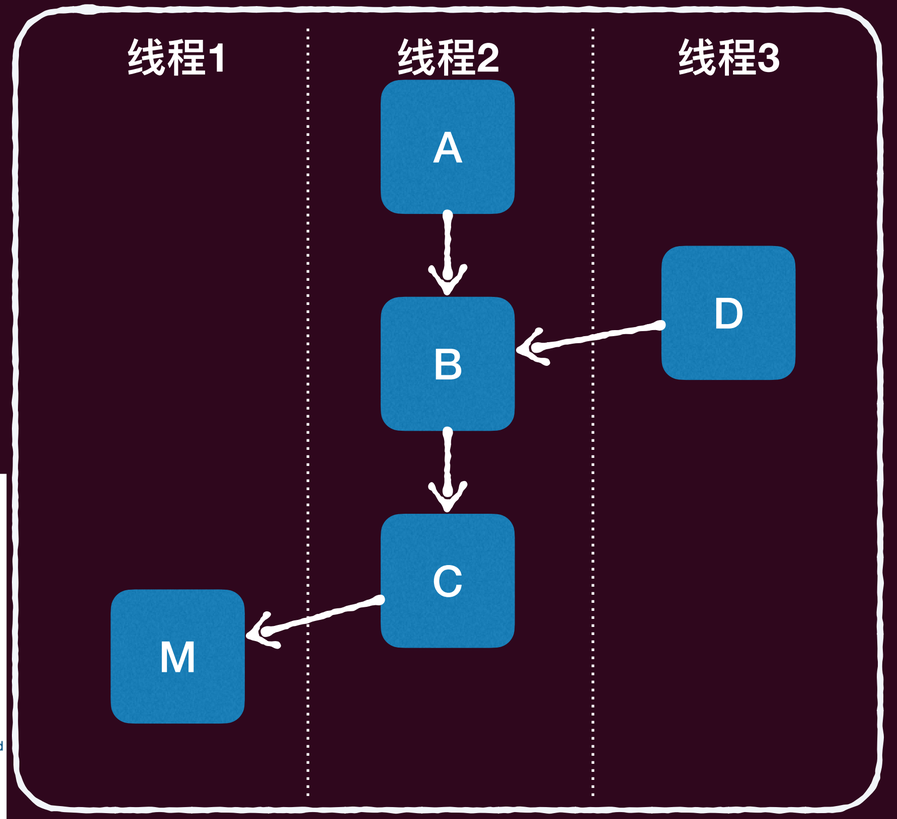
- 然而，这个算法是不实用的
  - 我们一般没法知道“最大”需求矩阵
  - 系统的资源是动态变化的
    - 线程数会变化
    - 资源数可能会变化（线程可以自己释放所持有的锁）

死锁检测并处理：让他发生吧！

- 之前的方法大多不实用或者并不能真正避免人类“犯错”
- 那么一个办法就是，让他发生吧，然后我们检测并处理
- 那么这个方案的核心就是如何检测系统出现了死锁！
  - 死锁是由于多个线程形成一个等待环，一个线程的行进需要环内的另外一个线程所持有的锁，从而所有线程都无法行进
  - 因此，核心就是检测锁申请环！

死锁的动态侦测

- 如何检测环?
  - 锁的持有和申请就是一个有向图
  - 有向图的环检测
    - 深度优先算法，出现回边就是出现了环

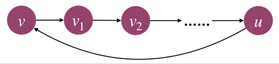


智能软件与工程学院
School of Intelligent Software and Engineering

Topological Sort

Lemma 1 Directed graph G is acyclic iff a DFS of G yields no **back** edges

- Proof of [$\Longrightarrow$] (Directed graph G is acyclic $\Longrightarrow$ a DFS of G yields no back edges)
  - For the sake of contradiction, assume DFS yields back edge (u, v).
  - So v is ancestor of u in DFS forest, meaning there's a path from v to u in G.
  - But together with edge (u, v) this creates a cycle. Contradiction!

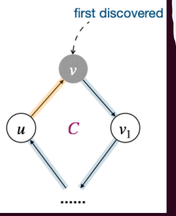


智能软件与工程学院
School of Intelligent Software and Engineering

Topological Sort

Lemma 1 Directed graph G is acyclic iff a DFS of G yields no **back** edges

- Proof of [$\Longleftarrow$] (Directed graph G is acyclic $\Longleftarrow$ a DFS of G yields no back edges)
  - For the sake of contradiction, assume G contains a cycle C.
  - Let v be the first node to be discovered in C.
  - By the **White-path** theorem, u is a descendant of v in DFS forest.
  - But then when processing u, (u, v) becomes a back edge!

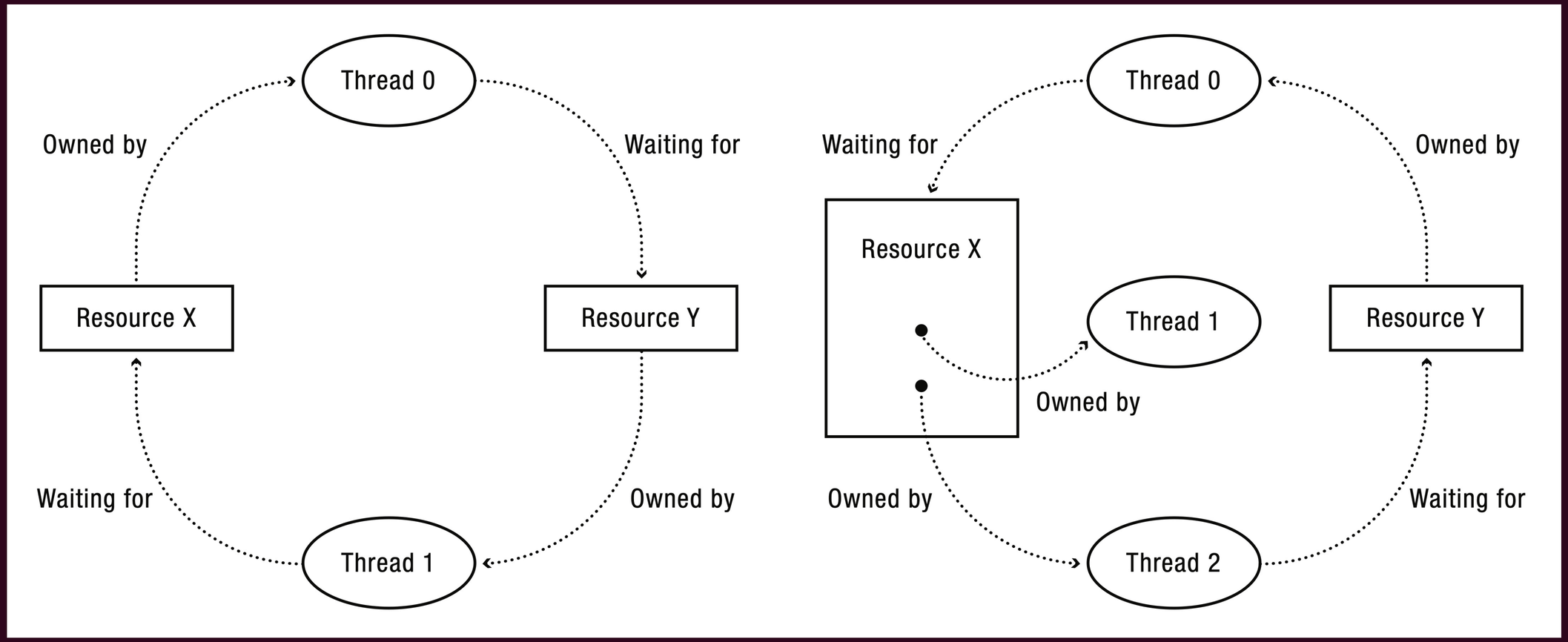


死锁的动态侦测

- 一个更加具体的实现(lockdep的简单原理)：
  - 每次锁的acquire/release都记录 tid 和 lock name，动态构建锁的有向图$G(V, E)$，并Assert该图没有环
    - V：为每把锁的名字
    - E：每个线程持有某把锁u之后再去尝试获得某把锁v，就加入边(u, v)
    - 锁的名字可以用地址来唯一绑定 （也可以是锁所在的文件和行，这是一种近似，因为可能同一个行是malloc不同的锁，相应地，代价低）

死锁的动态侦测

- 如果是一类的互斥资源有多个资源数呢？单纯的环已不足来检测死锁



死锁的动态侦测

- 给定：所有线程的当前持有资源为矩阵C、所有线程的当前申请资源为矩阵R、以及系统资源总数向量E

系统资源总数E

6	3	4	2

尚存资源向量 (总数-持有数)

1	0	2	0

线程	资源已持有矩阵C			
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

资源申请矩阵R

2	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
2	0	0	0

- 按照和之前银行家算法类似的思路：
  - 找到可以满足要求的线程，先假定分配给其资源，然后释放其所有资源（创造更好的局面）
  - 不断重复上述步骤，直到所有线程都可满足即没有死锁
  - 否则就是出现死锁

并发bugs的一些动态分析方法

什么是动态分析

- 什么是动态分析
  - 给定一次状态机（程序）的执行历史信息τ（比如日志log、covering lines、memory access…）
    - 当然这种记录往往需要额外的运行成本（比如插桩）
  - 动态分析即为根据这个信息的分析函数$f(\tau): \tau \rightarrow \{0,1\}$，0代表关心问题的答案为否（比如没有bug），1代表关心问题的答案为是（比如有bug），当然答案还可以有更多可能，那么这个值域会增大

本质上，只要你定义了什么是正确、什么是错误，并知道这种错误会在运行时产生有什么后果，就能检测出来

AddressSanitizer: 非法内存访问

- 通过编译器自动插入和内存相关的断言（比如每次分配内存时，额外分配一写不可写的内存（投毒），一旦访问到这些被投毒的内存，就知道越界了），实现代码正确性的检查。可以检测如Buffer (heap/stack/global) overflow, use-after-free, use-after-return, double-free, ...;

```c
#include <stdio.h>
#include <string.h>

void mystrcpy(char *dest, const char *src) {
    while (*src) {
        *dest = *src;
        dest++;
        src++;
    }
}
```

```c
int main() {
    char hello[] = "Hello, world!";
    char goodbye[] = "Goodbye, world!";
    char buf[13]; // Buggy!

    mystrcpy(buf, hello);

    printf("%s\n", buf);
    printf("%s\n", goodbye);
    return 0;
}
```

试试编译选项 -fsanitize=address

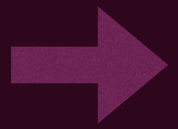
ThreadSanitizer: 运行时的竞态条件检测

- 回顾竞态条件(数据竞争)：不同的线程同时访问同一内存，且至少有一个是写
  - 基于这个定义，就能动态的检测出竞态条件
  - 这些条件中，不同线程是容易观察到的（记录thread id即可），同一内存也是容易观察到的（内存的地址），至少一个是写是容易观察到的（load和store指令）
  - 关键问题是，什么是同时?
    - 让我们问一个反向的问题：什么不是同时?

happens-before关系！

ThreadSanitizer: 运行时的竞态条件检测

- ThreadSanitizer可以为所有事件建立happens-before关系图
  - 比如对于下面两个事件
    - x: mutex_lock(A); load(x); mutex_unlock(A);
    - y: mutex_lock(A); store(x); mutex_unlock(A);



$$x \prec y \vee y \prec x$$

  - 比如对于wait和signal有显著的先后顺序
- 一个系统的刻画并发系统事件happens-before的方法：
  - Lamport's Vector Clock: Time, clocks, and the ordering of events in a distributed system
- 而对于发生在不同线程且至少有一个是写的检查，但是没有出现在上述的关系中，便是一个潜在的竞态条件！

试试编译选项 -fsanitize=thread

Sanitizers：现代复杂软件系统必备的支撑工具

- AddressSanitizer (asan): 非法内存访问
  - Linux Kernel 大量依赖于 KASAN
- ThreadSanitizer (tsan): 数据竞争
  - Kernel Concurrency Sanitizer: Concurrency bugs should fear the big bad data-race detector
- MemorySanitizer (未初始化的读取)
- UBSanitizer (undefined behavior如整数溢出、对象越界、溢出等一系列问题), …

低配版实现

- 对于需要手动实现操作系统的你们来说
  - 没有现成的Sanitizer
  - 也很难实现Full Sanitizer（各种依赖的库我们的实验框架都没有）
- 但可以给出一些低配的版本，保障你们实验的高效实现

Canary

- Canary: “牺牲” 内存单元，预警 memory error
  - 比如stack guard

```
#define MAGIC 0x55555555
#define BOTTOM (STK_SZ / sizeof(u32) - 1)
struct stack { char data[STK_SZ]; };

void canary_init(struct stack *s) {
    u32 *ptr = (u32 *)s;
    for (int i = 0; i < CANARY_SZ; i++)
        ptr[BOTTOM - i] = ptr[i] = MAGIC;
}
void canary_check(struct stack *s) {
    u32 *ptr = (u32 *)s;
    for (int i = 0; i < CANARY_SZ; i++) {
        panic_on(ptr[BOTTOM - i] != MAGIC, "underflow");
        panic_on(ptr[i] != MAGIC, "overflow");
    }
}
```

Canary

- Canary: “牺牲” 内存单元，预警 memory error
  - 比如buffer overflow

```
int foo() {
    // 一段连续内存；位于局部变量和返回地址之前
    u32 canary = SOME_VALUE;
    ... // 实际函数
    canary ^= SOME_VALUE; // 如果程序被攻击或出错
                          // canary 就不会归零了
    assert(canary == 0);
    return ret;
}
```

低配版lockdep

- 统计当前的 spin count
- 如果超过某个明显不正常的数值 (e.g., 100,000,000) 就报告
  - 来代替你对于当前系统发生死锁直觉

```
int spin_cnt = 0;
while (xchg(&lk, ❌) == ❌) {
    if (spin_cnt++ > SPIN_LIMIT) {
        panic("Spin limit exceeded @ %s:%d\n",
            __FILE__, __LINE__);
    }
}
```

配合debugging工具找到合适的断点进行调试，会大大缩减你需要调试的时间

低配版 AddressSanitizer

- 实验L1 内存分配器的 specification
  - 已分配内存$S = [l_0, r_0) \cup [l_1, r_1) \cup \ldots$
  - kalloc(s)返回的$[l, r)$必须满足$[l, r) \cap S = \varnothing$

并发的分配很可能破坏这个 specification

```
// allocation
for (int i = 0; (i + 1) * sizeof(u32) <= size; i++) {
    panic_on(((u32 *)ptr)[i] == MAGIC, "double-allocation");
    arr[i] = MAGIC;
}

// free
for (int i = 0; (i + 1) * sizeof(u32) <= alloc_size(ptr); i++) {
    panic_on(((u32 *)ptr)[i] == 0, "double-free");
    arr[i] = 0;
}
```

低配版 ThreadSanitizer

- 竞态条件会产生什么后果?
  - 线程的读写一个共享数据不是原子的，即在中间可能被其他线程插入
    - 但是这个错误一般很难观测，因为指令的读写太快了
  - 想法：通过拖慢线程读写速度，放大原子性破坏的可能性（执行时间越长，被其他线程干涉的可能性越大）
- Effective data-race detection for the Kernel (OSDI'10)

```
// Suppose x is lock-protected
...
int observe1 = x;
delay();
int observe2 = x;
assert(observe1 == observe2);
...
```

低配版的意义

- 给定一个程序正确运行的specification，原则上我们能够写出动态分析的方法验证程序运行时有没有破坏这个specification
- 但有时候验证一个完整的specification过于复杂或者条件不允许（比如我们的实验，比如计算资源受限）
- 但此时如果能够从这个full specification推出一些弱化的specification，从而更加容易的实现一些Sanitizer，那么即使精度可能没那么高，但实际中会有很好的效果！
  - 比如:你能给出图像识别软件的full specification吗?

总结

- 即使已经学了很多并发控制的工具，人类依然会犯错
- 死锁就是其中一大类
  - 死锁可以通过破坏其必要条件来避免
  - 也可以通过动态的合理分配来避免
  - 也可以通过动态分析来检测
- 并发的其他bugs也有一些非常实用的动态分析技术来检测
  - 各种Sanitizer

阅读材料

- [OSTEP] 第32章