

6. 虚拟链路

1

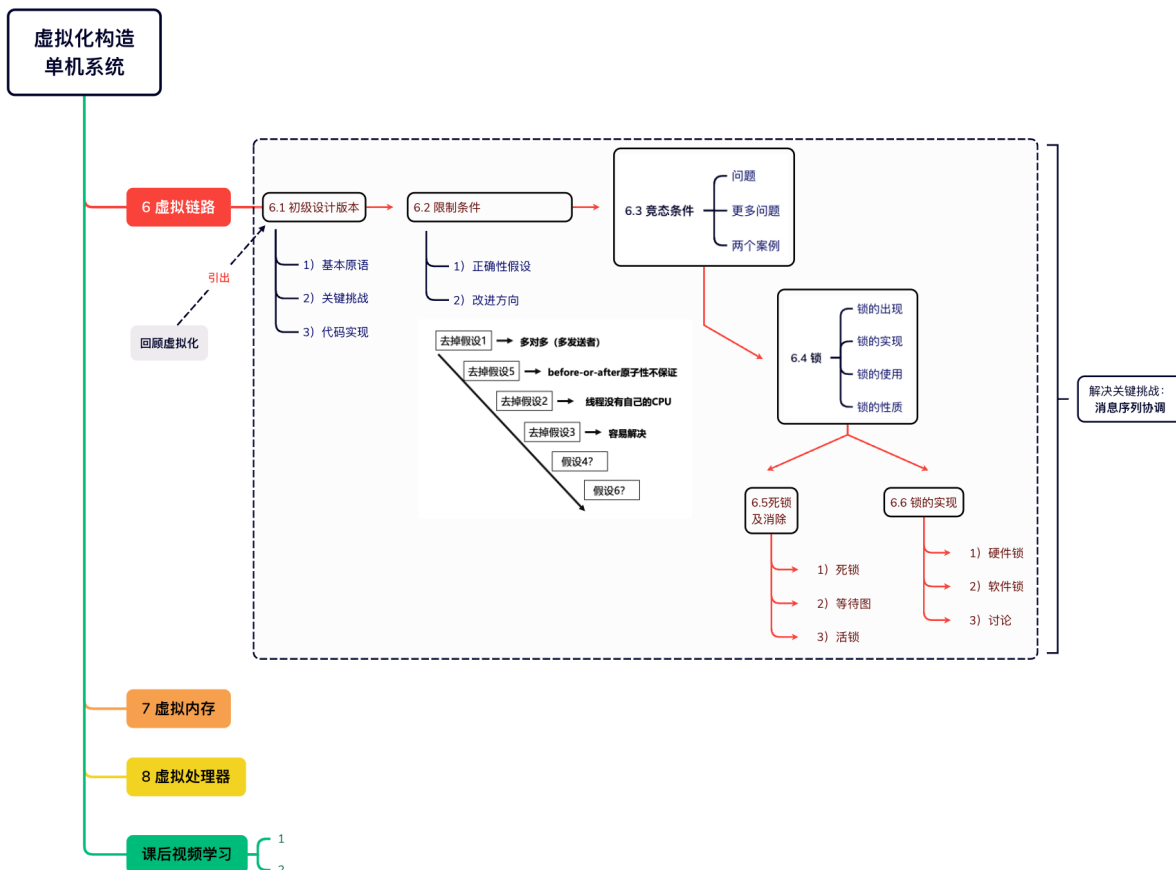
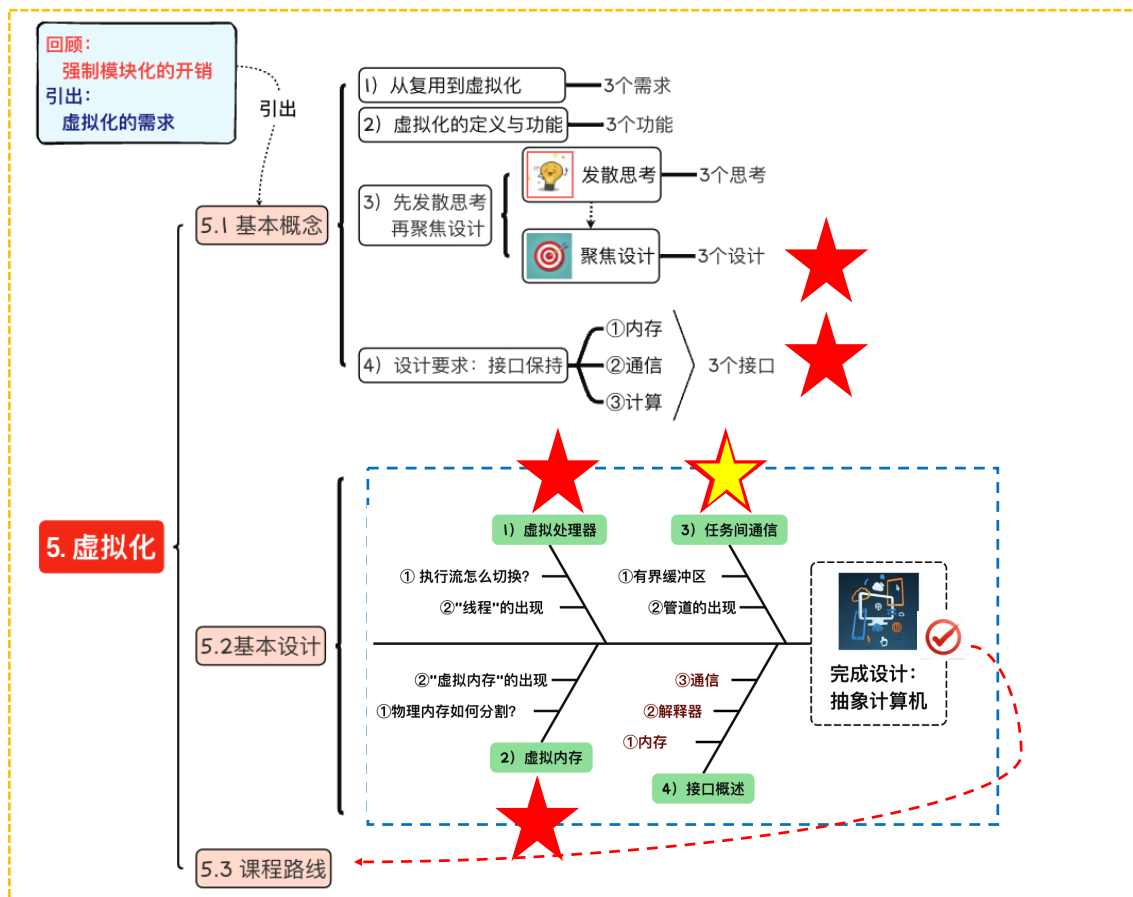


本章主要参考文献

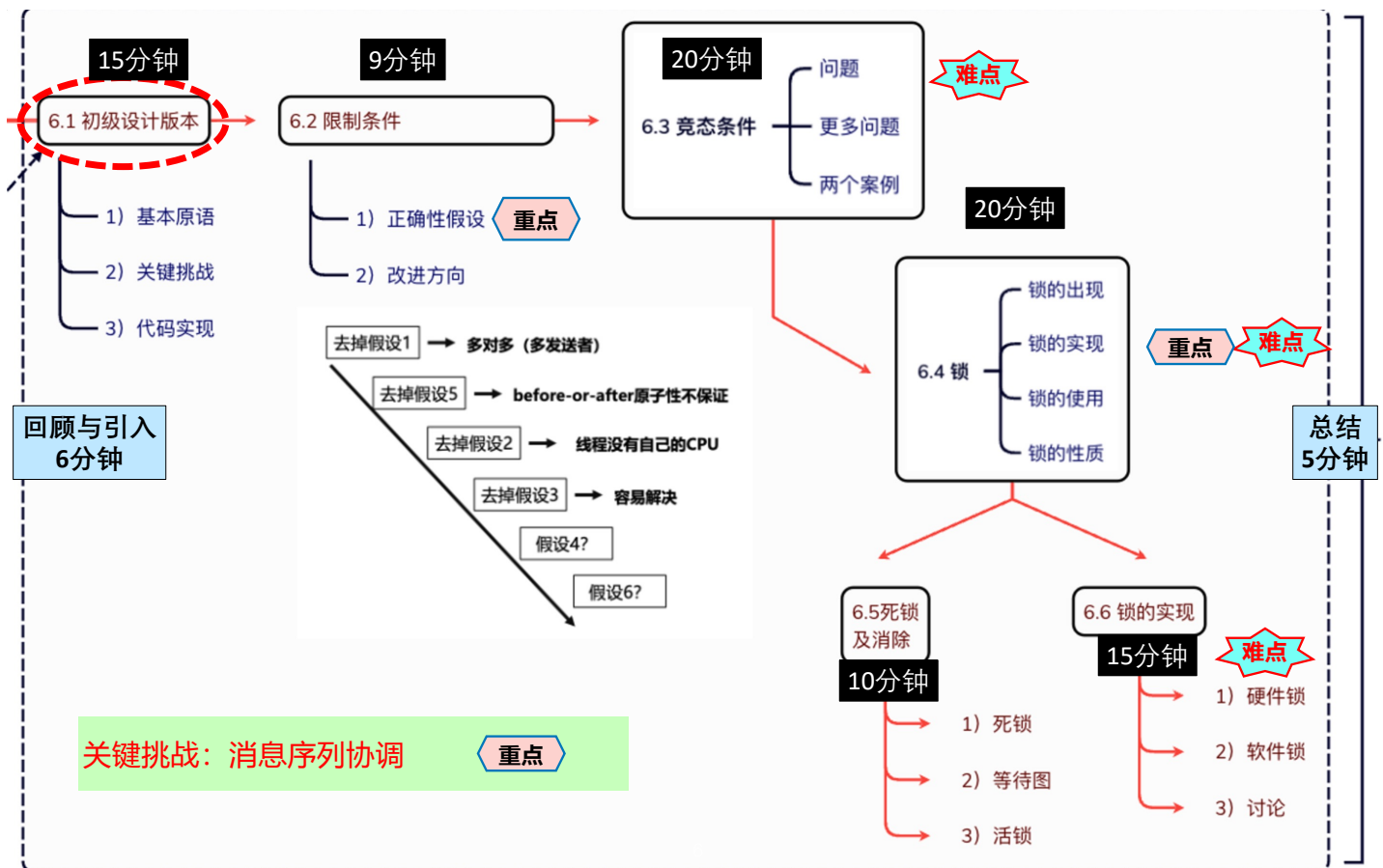


- Anderson T E, Bershad B N, Lazowska E D, et al. Scheduler activations: Effective kernel support for the user-level management of parallelism[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 53-79.
- Clark D D. The structuring of systems using upcalls[C]//Proceedings of the tenth ACM symposium on Operating systems principles. 1985: 171-180.
- Saltzer J H. Traffic control in a multiplexed computer system[D]. Massachusetts Institute of Technology, 1966.

回顾



6.1 初级设计版本



回顾：线程间通信

(第7次理论课：5.2 有界缓冲区)

- 一段固定大小的内存，首尾两个指针

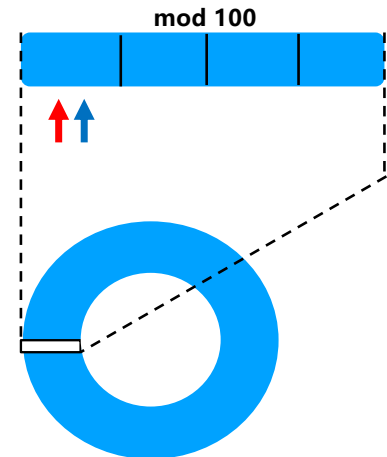
- 发送者将数据写入
- 数据首部指针：in+1

首-尾=N
则等待

- 接收者读取数据，并删除
- 数据尾部指针：out+1

首=尾
则等待

0	1	10	11
100	101	110	111
1000	尾	✓	✓
头	1101	1110	1111



——《数据结构》循环队列

1) 基本原语（初级设计版本）

● 4个原语

- 1) 分配：分配大小为n的缓冲区

1) $buffer \leftarrow allocate_bounded_buffer(n)$

- 2) 释放：释放缓冲区

2) $deallocate_bounded_buffer(buffer)$

- 3) 发送：插入消息，缓冲区满则等待

3) $send(buffer, message)$

- 4) 接收：读走消息，缓冲区空则等待

4) $message \leftarrow receive(buffer)$

● 实现：用stub实现本地RPC，在调用者来看与RPC无差别

2) 关键挑战

如果两线程并发访问相同内存

能否确保结果正确?

如何确保结果正确?



消息序列协调

如何协调线程之间的事件的先后顺序?

谁来协调关键挑战?

- 计算机系统不同领域的思维差异
 - 操作系统(内核)领域
 - 用户是Genius
 - 其他领域
 - 用户是Mortal

其他领域的自动化机制是谁建立的呢?

3) 代码实现

- Send
- Receive

什么情况下，线程会循环等待？

空/满。

循环等待对程序有什么影响？

```
1 shared structure buffer
2   message instance message[N]
3   long integer in initially 0
4   long integer out initially 0
5   procedure SEND (buffer reference p, message instance msg)
6     while p.in - p.out = N do nothing //
7     p.message[p.in modulo N] = msg    // in和out:
8     p.in = p.in + 1                    in模N指向队头+1的单元;
9   procedure RECEIVE (buffer reference p)
10    while p.in = p.out do nothing        out模N指向队尾的单元。
11    msg = p.message[p.out modulo N]    // in==out: 队头在队尾之后 = 空
12    p.out = p.out + 1                  in-out==N: 队头离队尾N = 满
13    return msg
```

数组模N：循环使用内存

例：.....n-2, n-1, 0, 1.....

*in*和out:

*in*模N指向队头+1的单元;

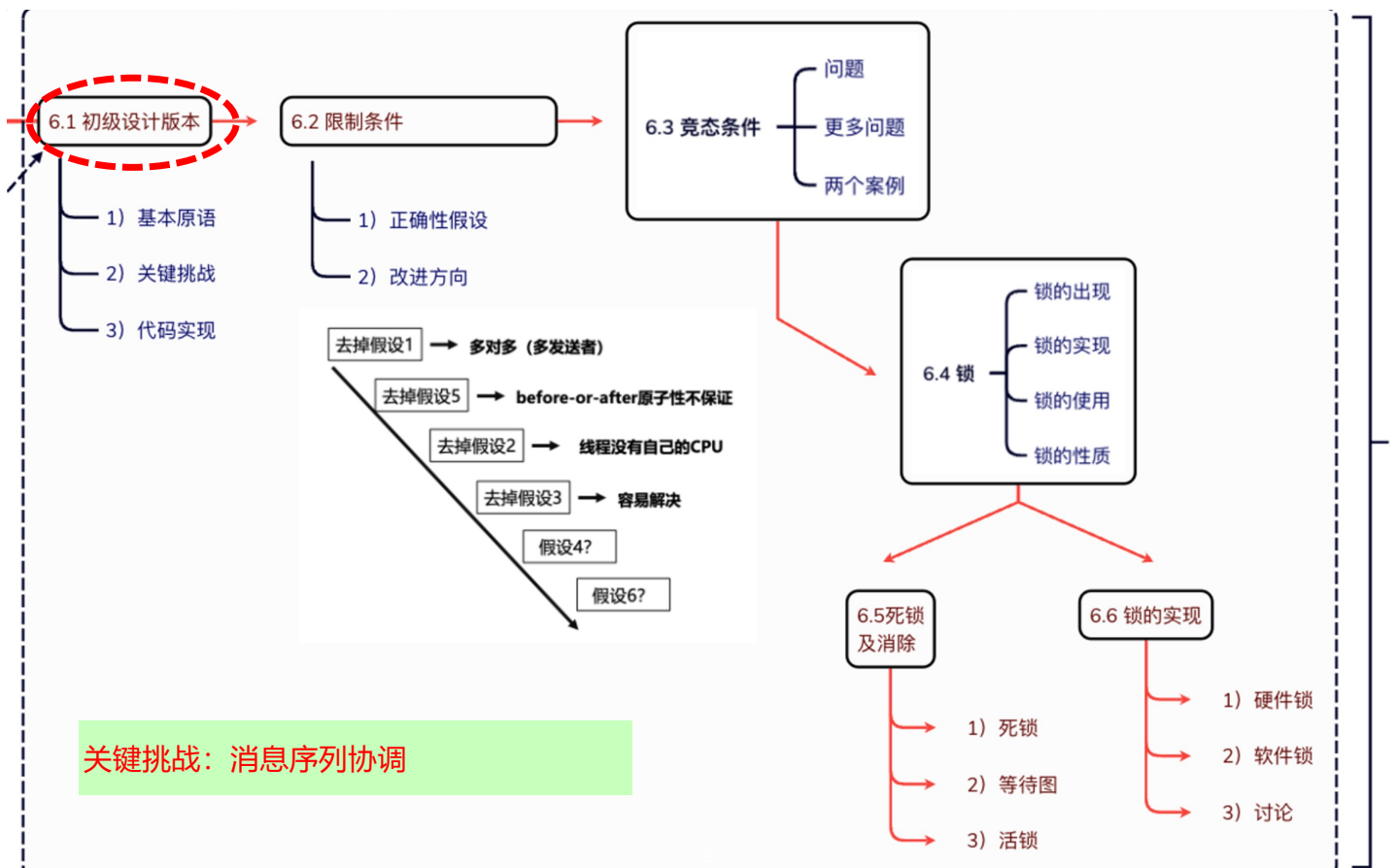
*out*模N指向队尾的单元。

in==*out*: 队头在队尾之后 = 空

in-*out*==N: 队头离队尾N = 满

这段程序的正确性，有哪些限制？

计算机系统工程导论



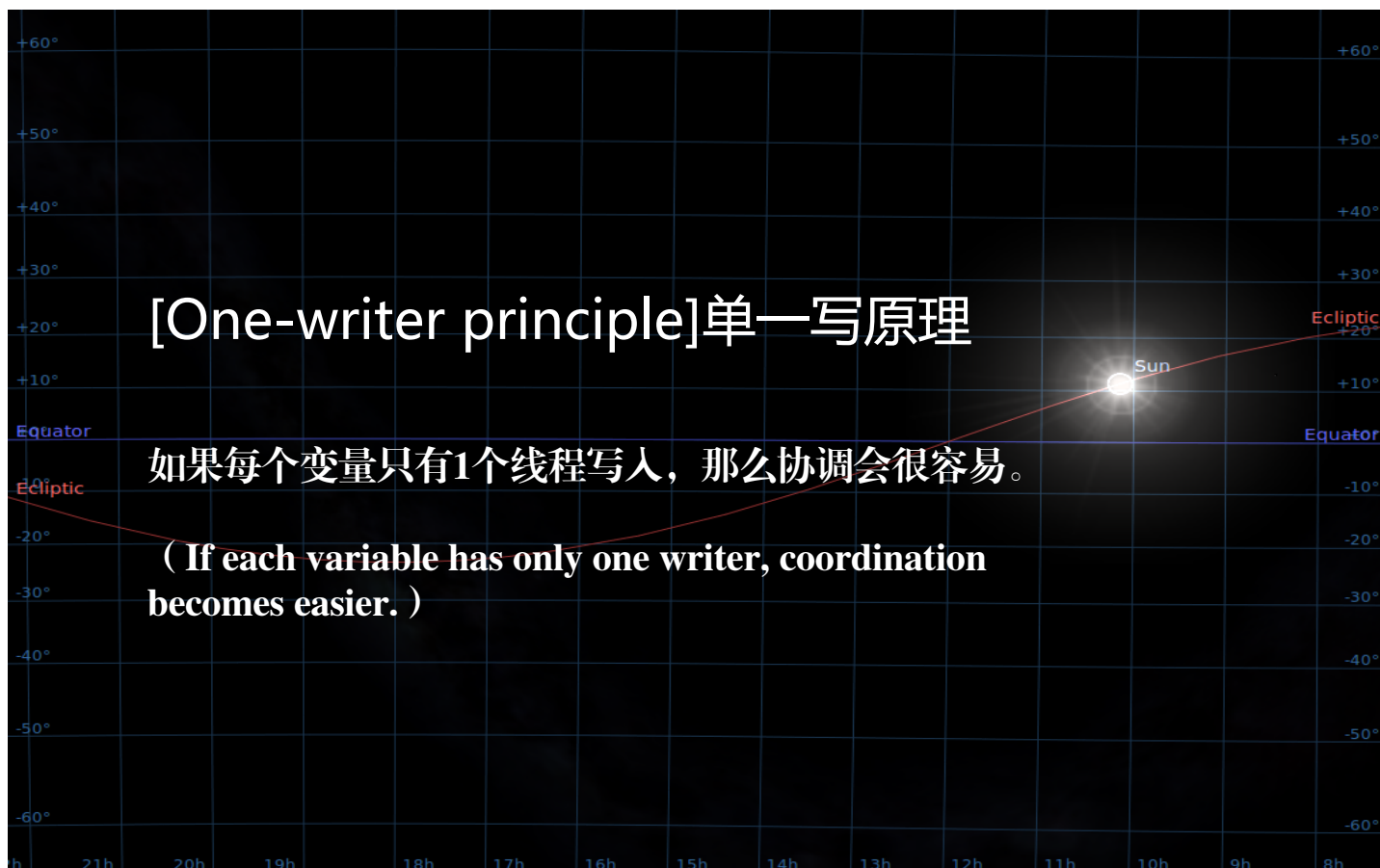
1) 正确性假设

1. 收发各1个线程，不支持多发送/多接收
 - [单一写原理]
 - (课下思考：如果用链表实现，有问题吗？)
2. 线程拥有自己的处理器
 - 可以自旋 (spin)
3. in和out不出现溢出
 - 实现：足够大，64位以上
 - 变体：in和out直接模N，如何禁止套圈？

[One-writer principle]单一写原理

如果每个变量只有1个线程写入，那么协调会很容易。

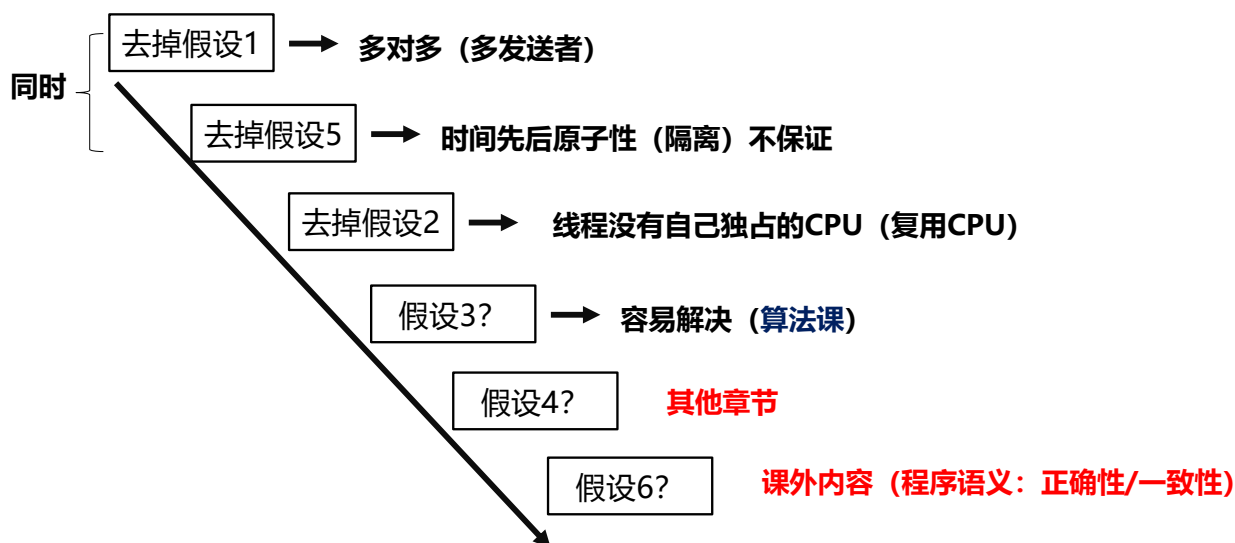
(If each variable has only one writer, coordination becomes easier.)

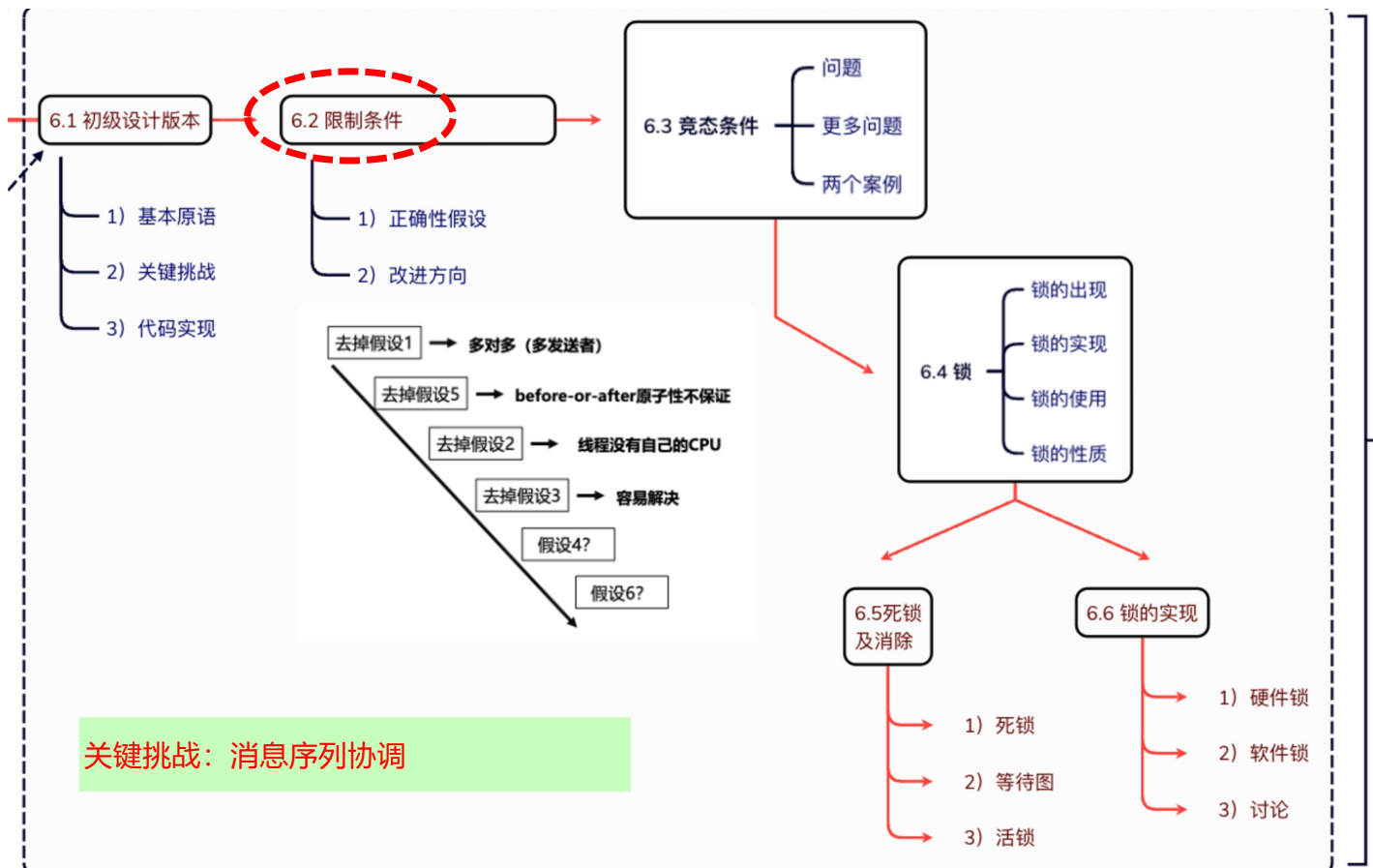


1) 正确性假设

4. 读/写连贯性 (read/write coherence) 已经得到保证**
5. 时间先后原子性 (before-or-after atomicity-隔离) 已得到保证
 - <buffer , in/out> 的写入
 - 如果in/out变量使用多个机器字长, in/out的写入
 - 如果buffer的单元是多个机器字长, buffer的写入
6. 程序指令没有因优化被重新排序 (reorder) **
 - 优化例如: 将in读到寄存器, 一口气读出若干消息, 最后修改in值写回去。(难点: 除非这个过程为原子过程, 否则出错)

改进方向





6.3 竞态条件 (Race condition)

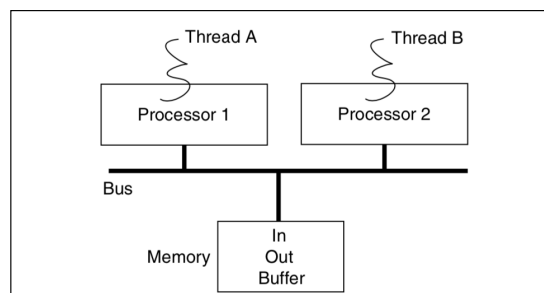
● 去除假设1和5会导致什么问题？

1.多发送者

5.时间先后原子性不保证

● 并发问题：

- 1 → 产生并发条件 5 → 产生并发错误
- 海森堡bug (Heisenbug)



实例 → 错误

- 实例：
 - $N=20$, $out=0$; $in=0$
 - 线程A 和 线程B (分别拥有处理器)
 - 由于中断、缓存等, 无法预估A和B的指令执行时间之间的关系
- (回顾第2章的异步解释器——asynchronous interpreter)

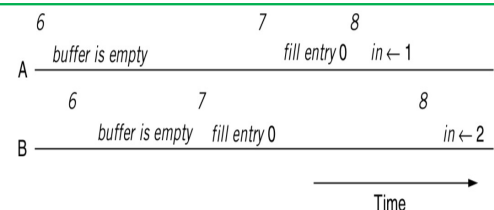
实例 → 错误

- A6表示A执行了第6行代码, 按时间排序
 - A6 A7 A8 B6 B7 B8 → 正确!
 - A6 B6 B7 A7 A8 B8 → 正确? 错误!

竞态条件/竞争条件 (race condition)

多线程访问共享资源, 操作时序导致的错误情况。

特点: 难以重现 - Heisenbug



```
5 procedure SEND (buffer reference  $p$ , message instance  $msg$ )
```

```
6   while  $p.in - p.out = N$  do nothing    // 如果缓冲区满, 等待缓冲区腾出空间
```

```
7    $p.message[p.in \text{ modulo } N] = msg$     // 把信息存入缓冲区
```

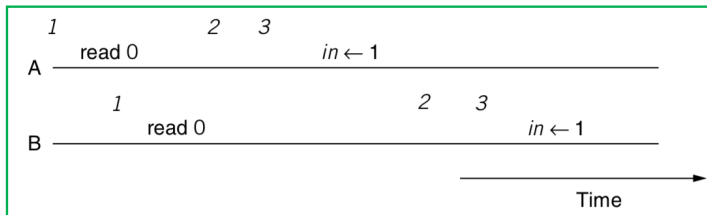
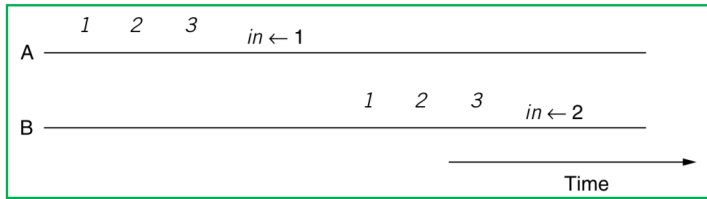
```
8    $p.in = p.in + 1$                       //  $in$ 增加
```

更多问题

- line 8 的语句 $p.in = p.in + 1$ 的汇编代码:

```
1 load in, r0
2 add r0, 1
3 store r0, in
```

- 考虑以下情况:



- 正确? 错误?

只去除假设5是否能避免问题?

5.时间先后原子性不保证

实例:

- in和out使用2倍机器字长的存储
- $in \bmod 20$ 求得位置, 目前为空。
- 更新in的汇编代码:
 - 1 store r0, $in+1$
 - 2 store r1, in
- 读取in的汇编代码:
 - 3 load $in+1$, r0
 - 4 load in , r1

只去除假设5是否能避免问题？

5.时间先后原子性不保证

错误：

- `in = out = 00000000FFFFFFFFH` (大端)
- A写入消息之后更新in
 - `in+1 = 00000001 00000000H` ($\equiv 16 \bmod 20$)
 - 先高地址 (低字节) : `FFFFFFFF` → `00000000`, 再低地址
- B读取in, 按照in读取消息
 - 读: `0000000000000000`
 - `in-out = ?` 出错!

两个案例

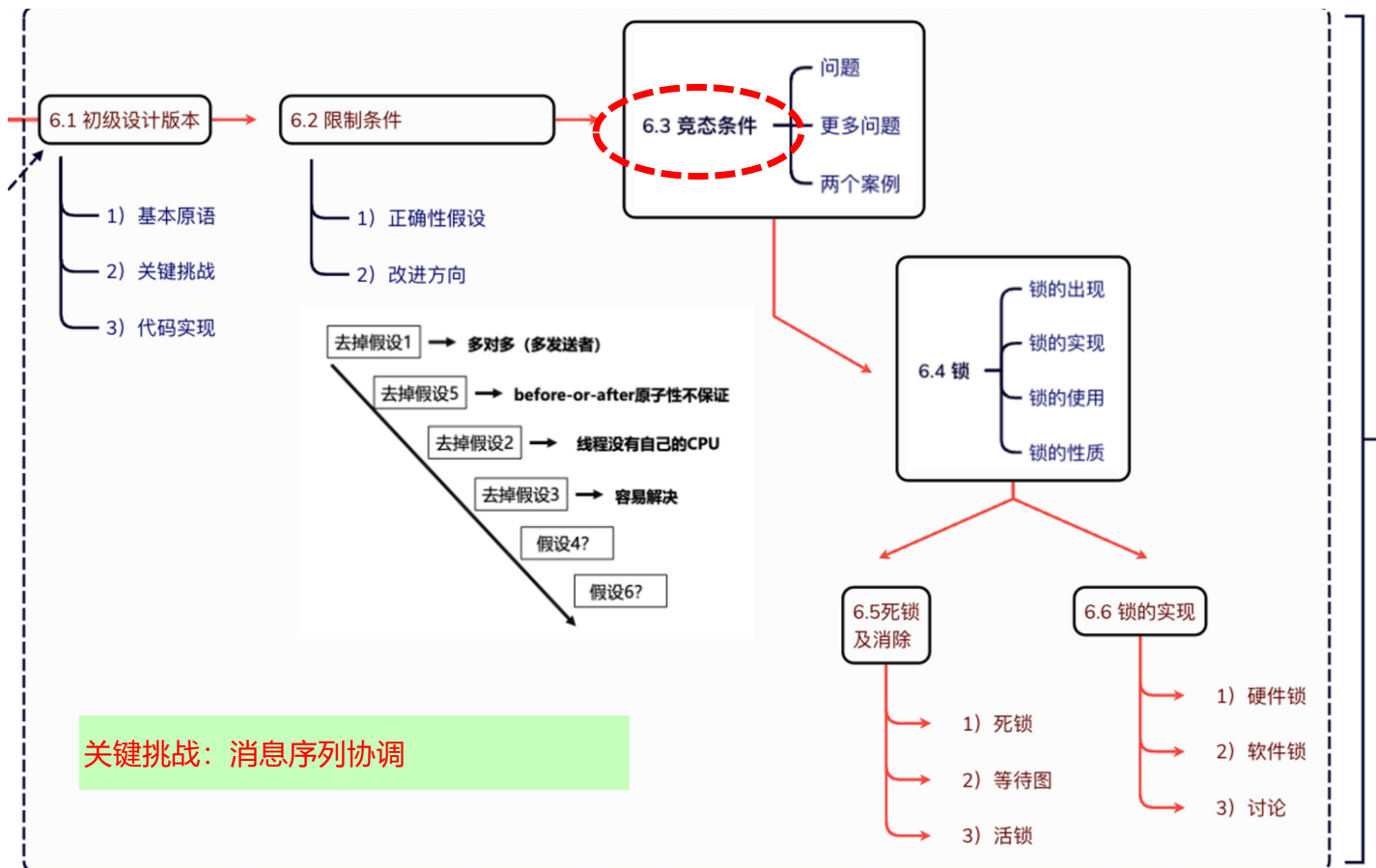
1. CTSS

- 口令文件泄露

2. Therac-25

- 严重医疗事故
- (文献Nancy G. Leveson)

● 习题



6.4 锁 (lock)

- 为什么会有竞态条件 (race condition) ?
 - 时序被交错
- 怎么防止?
 - 禁止时序交错
 - 禁止多线程? 🐱
 - 把关键操作 (临界区) 锁定, 禁止交错!

锁 (lock)

锁的原语

- 可以强制保证假设1和5
 - 为什么?
- 两个原语:
 - ACQUIRE: 在为0的前提下, 设为1
 - 只有 ACQUIRE成功, 方可操作
 - RELEASE: 设为0
- 初级实现
 - 1个共享变量, 用于保护一组对共享资源的操作

锁的实现

```
5 lock instance buffer_lock initially UNLOCKED // 发送方或接收方上锁
6 procedure SEND (buffer reference p, message instance msg)
7   ACQUIRE (p.buffer_lock)
8   while p.in - p.out = N do // 等待, 直到缓冲区有足够空间
9     RELEASE (p.buffer_lock) // 在等待过程中, 释放锁以便进行接收
10    ACQUIRE (p.buffer_lock) // 可以移除一条信息
11    p.message[p.in modulo N] = msg // 把信息存入缓冲区
12    p.in = p.in + 1 // in增加
13  RELEASE (p.buffer_lock)
```

锁的使用

- 预先：锁与资源建立关系
 - 所有对某资源的访问都需要申请对应的锁
- 使用流程
 - 获得锁
 - 操作
 - 释放锁

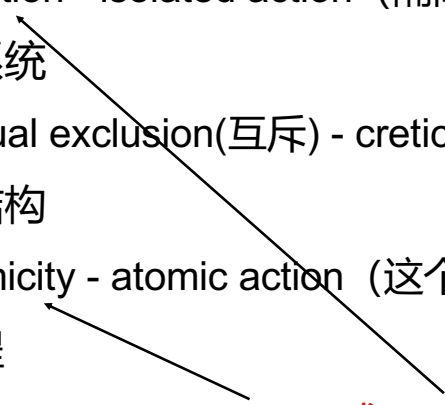
对于多访问下的组合操作，锁实现了“单一写”（one writer）

锁的性质

- 锁是非强制性的
 - 没有acquire到锁，就不能操作了吗？
 - 是一种约定，线程自律
 - 需要**正确使用**！例：锁只保护了11、12行，未从第8行开始

```
6 procedure SEND (buffer reference p, message instance msg)
7   ACQUIRE (p.buffer_lock)
8   while p.in - p.out = N do           // 等待，直到缓冲区有足够空间
9   RELEASE (p.buffer_lock)         // 在等待过程中，释放锁以便进行接收
   ACQUIRE (p.buffer_lock)             // 可以移除一条信息
11  p.message[p.in modulo N] = msg      // 把信息存入缓冲区
12  p.in = p.in + 1                     // in增加
13  RELEASE (p.buffer_lock)
```

“时间先后原子性” 在不同领域的名字

- 数据库与事务
 - isolation - isolated action (隔离)
 - 操作系统
 - mutual exclusion(互斥) - critical section(临界区)
 - 体系结构
 - atomicity - atomic action (这个词在数据库领域有不同含义)
 - 本课程
 - before-or-after atomicity或isolation
 - 避免歧义
- 

解决竞态条件的重要性

- 计算机(系统)领域为之付出了巨大努力!
- 课程仅介绍初步基础, 深入的内容需延伸阅读
- 相关课后阅读已布置

锁原语的实现策略

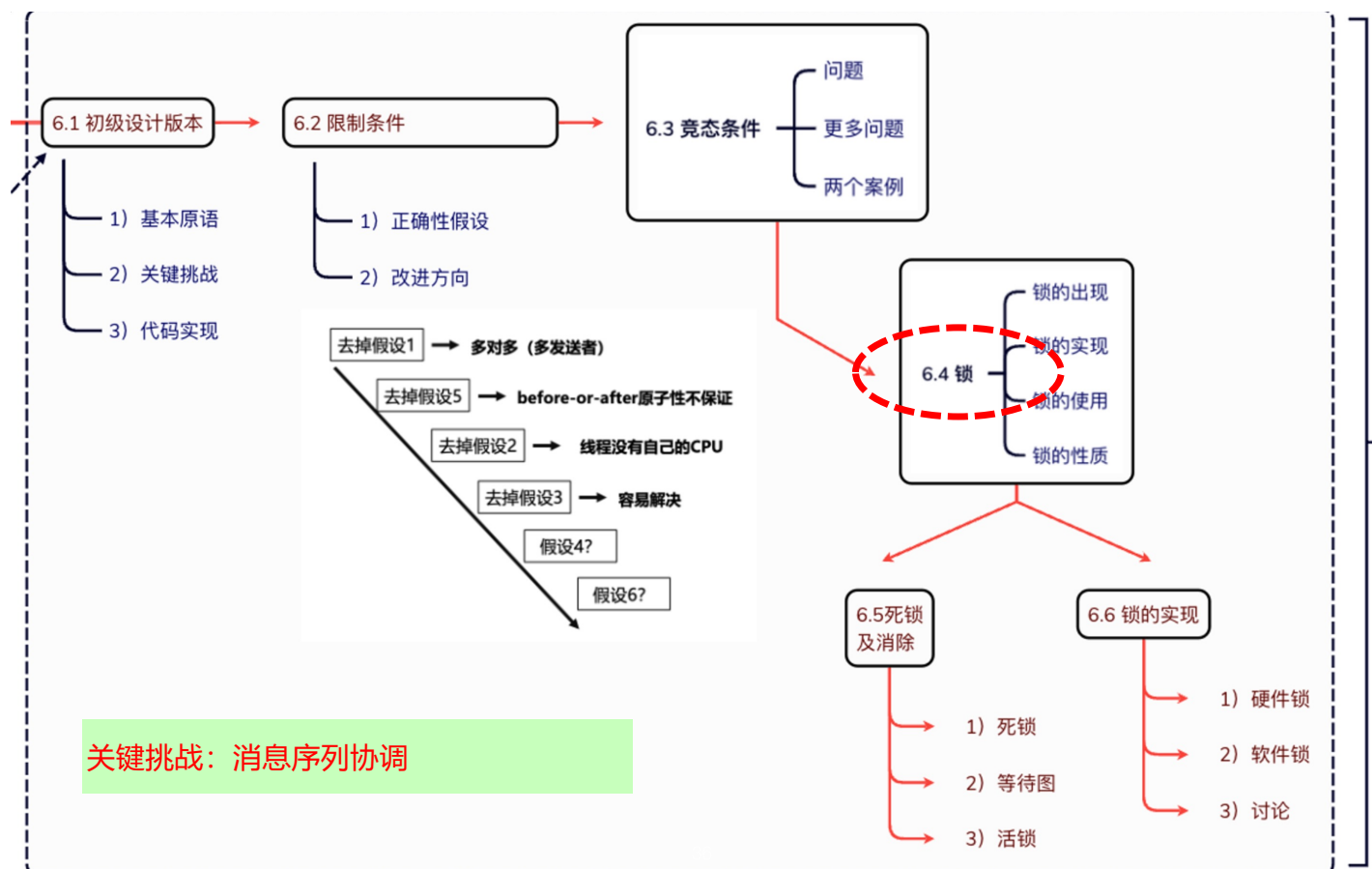
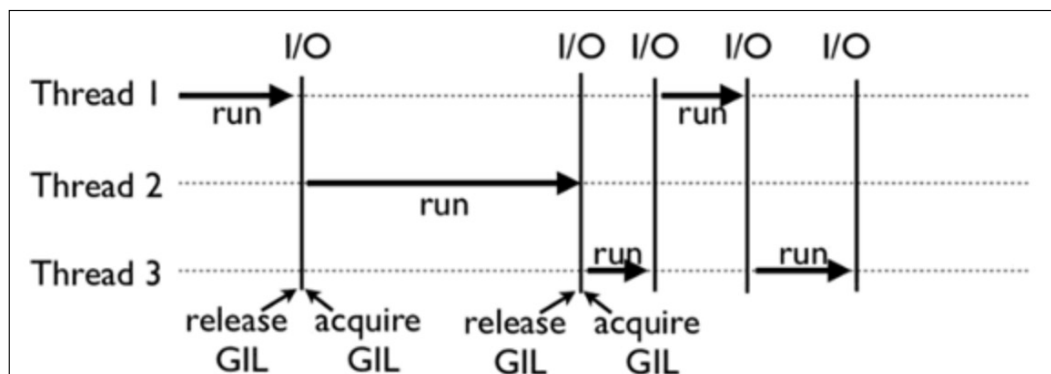
- 常见策略：单一锁协议（single acquire protocol）
 - 任何时候，只有1个线程可获得锁
- 更多策略
 - 如果了解共享变量使用细节，可设计更宽松的协议
 - 例如：读并发

问题：多个共享变量

- A. 使用多个锁，每次保护1个变量的访问，细粒度保护
 - 增加了复杂度，降低了可读性
 - B. 使用1个锁，1次保护所有变量的访问，粗粒度保护
 - 所有共享变量被串行化
 - 全局锁？
- 深入了解（文献Birrell）

思考：为什么Python多线程很慢？

● Python GIL

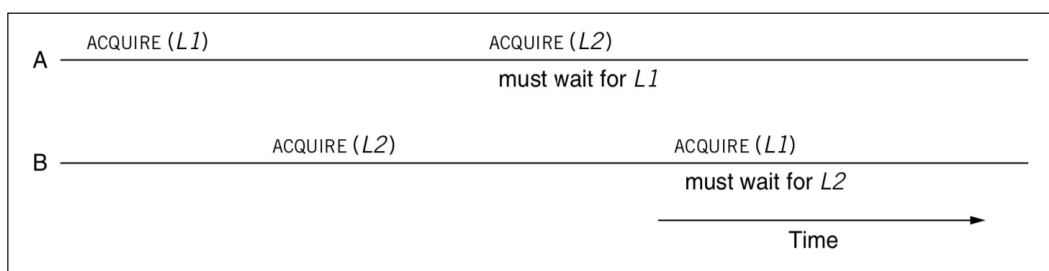


6.5 死锁 (deadlock)



- 死锁的出现:

- 两个线程, 都需要A、B两个锁, 分别拿到了1个
- 无法向前进行



- 定义: 在多线程环境中, 各个线程既在等待其他线程持有的资源, 又不释放已持有的资源, 互相依赖所造成的阻塞局面。

6.5 死锁 (deadlock)



- 问题:

- 两个线程, 都需要A、B两个锁, 分别拿到了1个
- 无法向前进行

静态办法: 给锁排个顺序, 只能按顺序拿

效率如何?

所有的锁都需要排序吗?

部分排序: 细心的编程

依赖关系的分析?

6.5 死锁 (deadlock)



- 问题:

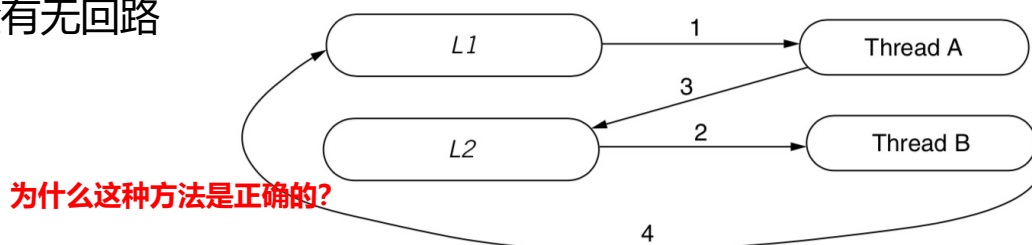
- 两个线程, 都需要A、B两个锁, 分别拿到了1个
- 无法向前进行

**动态方法: 扫描锁的请求, 分析制约关系
出现问题再解决
wait-for graph (等待图)**

等待图方法表示死锁

- 方法

1. 已获得锁: 画线 锁→线程
2. 正等待锁: 画线 线程→锁
3. 检验有无回路



- 事务常用: 使用事务等待列表, 发现死锁进行回滚。

其他类型的死锁

- 忘了释放

```
5  lock instance buffer_lock initially UNLOCKED    // 发送方或接收方上锁

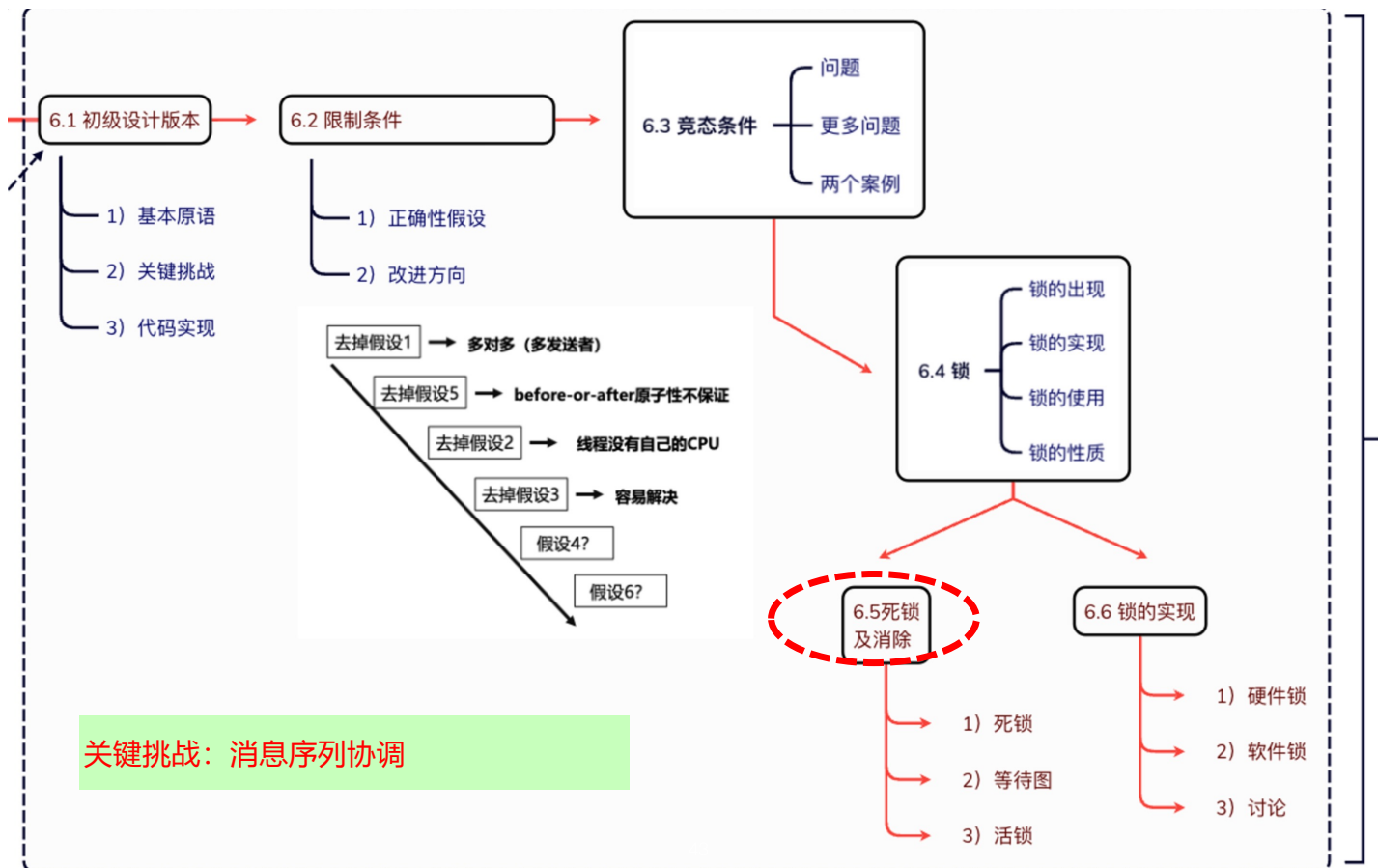
6  procedure SEND (buffer reference p, message instance msg)
7      ACQUIRE (p.buffer_lock)
8      while p.in - p.out = N do                    // 等待，直到缓冲区有足够空间
9          RELEASE (p.buffer_lock)                // 在等待过程中，释放锁以便进行接收
10         ACQUIRE (p.buffer_lock)              // 可以移除一条信息
11         p.message[p.in modulo N] = msg          // 把信息存入缓冲区
12         p.in = p.in + 1                          // in增加
13         RELEASE (p.buffer_lock)
```

等待图怎么画?

活锁 (livelock)

- 重复但不能前进





6.6 锁原语的实现

● 目标：单一获取协议 (single-acquire protocol)

- 不管有几个acquire，只有1个可以成功
- acquire本身必须是before-or-after操作

● 以下实现正确吗？

如何保护lock中的state? 锁? 正确!

```

1 structure lock
2     integer state
3
4 procedure FAULTY_ACQUIRE (lock reference L)
5     while L.state = LOCKED do nothing    //盲等待直到L解锁
6     L.state = LOCKED                    //while循环结束之后上锁
7
8 procedure RELEASE (lock reference L)
9     L.state = UNLOCKED
  
```

从归约到自举

- 归约 (reduction)
 - 假设B正确, A就正确
 - 类比: 数学归纳法
 - 奠基: 锚点
- 自举 (bootstrapping)
 - 系统地将通用问题归约到特定问题, 再用特定方式解决这个问题
 - 常用于: 操作系统启动、可信计算、锁
 - 例如: 将系列操作归约到1个单变量锁, 再用RSM指令实现它。
 - 问题分成了2步: ①归约通用问题, ②解决特定问题。

锁的奠基

- 硬件锁
- 软件锁

RSM指令：硬件锁

- RSM指令-read and set memory

- 一条指令，同时完成读、写内存单元的操作

- 依靠总线仲裁器：确保读写操作满足before-or-after原子性
 - 例：读写在一个总线周期（bus cycle）完成
 - 把问题归约给了一个特殊的硬件——总线仲裁器（bus arbiter）

- 自举（bootstrapping）任意高层锁

用RSM实现高层锁

- 正确性假设：写读连贯性（read/write coherence）

- 第7行代码赋值后，另一线程可马上读到L.state的状态

- 但某些优化机制不遵守写读连贯性语义 如果锁变量在缓存中，还有效吗？

- 需要附加的机制来保证

```
1  procedure ACQUIRE (lock reference L)
2      R1 ← RSM (L.state)           // read and set lock L
3      while R1 = LOCKED do         // was it already locked?
4          R1 ← RSM(L.state)        // yes, do it again, till we see it wasn't
5
6  procedure RELEASE (lock reference L)
7      L.state ← UNLOCKED
```


RSM、test-and-set及其他（线上）

- test-and-set

- 名字来自于早期的版本：先test再set，同时还设置一位成功标志
- 使用：while test_and_set (L) = locked do nothing
- test-and-set，实际上是read-test-set
- read是必须的，但test是多余的，只管set就行
- 使用read不用test，就把test留给了上层实现

RSM、test-and-set及其他（线上）

- test-and-test-and-set

- 因为test-and-set是原子操作（锁总线），先使用test测试可以减少性能下降

- COMPARE_AND_SWAP(v1,m,v2)

- 如果m=v1，那么m←v2
- 乐观non-blocking，直接操作数据而不加锁

- read-copy update

- Linux kernel使用
- 适用于经常读、不常更新的数据结构
- 进一步阅读（文献Paul）

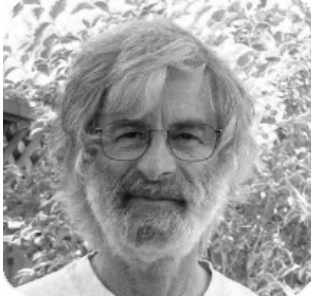
软件锁：借助“单一写原理”

- 使用软件实现RSM（思考？）
- 关键
 - 读是单步操作，修改即读后写是多步操作，单一写是单步操作！
 - “单一写”保护修改操作不发生竞争？
 - 确保只有一个线程可以修改一个变量
- 怎么做？
 - 需要多少变量？

软件RSM

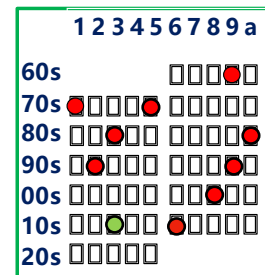
- L. Lamport方案
 - 正确性前提
 - 总线仲裁器保证每个内存访问操作是原子的
 - 保证写读连贯性（read/write coherence）
 - 保证程序顺序执行，没有重排（reorder）

LESLIE LAMPORT



United States – 2013

For fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency.



Lamport方案

```

shared boolean flag[N]                                // one boolean per thread

1  procedure RSM (lock reference L)                    // set lock L and return old value
2    do forever                                         // me is my index in flag
3      flag[me] ← TRUE                                  // warn other threads
4      if ANYONE_ELSE_INTERESTED (me) then             // is another thread warning us?
5        flag[me] ← FALSE                               // yes, reset my warning, try again
6      else
7        R ← L.state                                    // set R to value of lock
8        L.state ← LOCKED                               // and set the lock
9        flag[me] ← FALSE
10     return R
11
12  procedure ANYONE_ELSE_INTERESTED (me)              // is another thread updating L?
13    for i from 0 to N-1 do
14      if i ≠ me and flag[i] = TRUE then return TRUE
15    return FALSE
    
```

Lamport方案

每个线程有自己的编号：me，每个编号对应1个flag

● 步骤（循环直到返回）

- 1.设置me的flag，表达想法
- 2.检查其他线程的flag，如果有其他线程有想法，改回flag
- 3.如果没有线程有想法，读取锁状态，**加锁**

● 分析

- 1.每个线程的返回值和加锁操作是原子性的
- 2.但也可能AB都不成功，一直循环 → 活锁

Peterson算法解决了
2线程的活锁问题

讨论

● Lamport算法看起来比硬RSM更好？

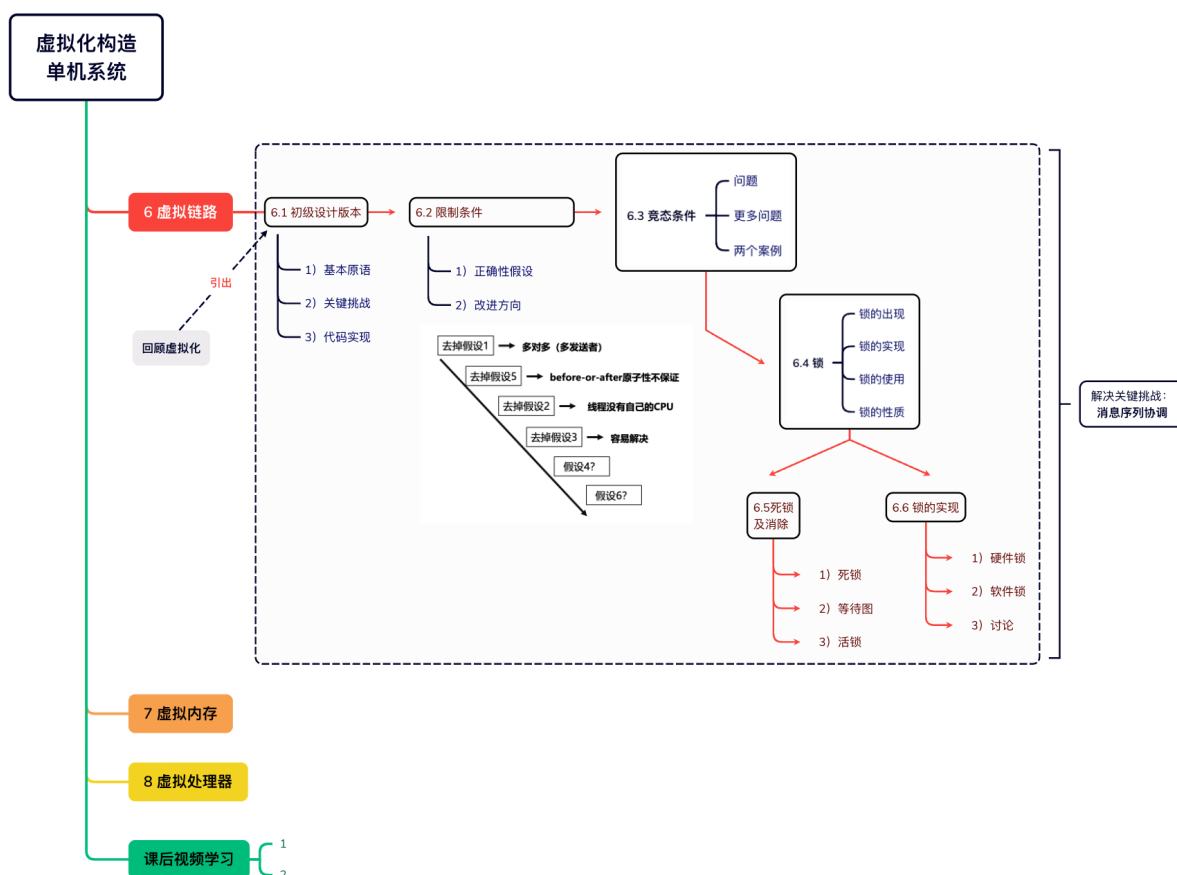
- 开销与收益
- 性能：软件锁可以最优化到2次读+5次写
- 如果线程数是动态变化的，那么每个锁的flag长度都是动态的，需要更复杂的数据结构和算法
- 出于复杂性的考虑，硬件RSM更广泛

为什么要了解系统设计背后的原理？ 仅仅熟练掌握现有的系统是不够的？

- 系统设计者，应了解系统正确的原理和前提，在前提改变时进行审慎的分析。
- 这也是我们经常回顾历史的原因。



总结





重要术语中英文对照



时间先后原子性: before-or-after atomicity

隔离: isolation

竞态条件: race condition

锁: lock

死锁: deadlock

活锁: livelock

等待图: wait-for graph

规约: reduction

自举: bootstrapping



本章主要参考文献



- Anderson T E, Bershad B N, Lazowska E D, et al. Scheduler activations: Effective kernel support for the user-level management of parallelism[J]. ACM Transactions on Computer Systems (TOCS), 1992, 10(1): 53-79.
- Clark D D. The structuring of systems using upcalls[C]//Proceedings of the tenth ACM symposium on Operating systems principles. 1985: 171-180.
- Saltzer J H. Traffic control in a multiplexed computer system[D]. Massachusetts Institute of Technology, 1966.

第6章 结束