聊聊并发

医疗项目部
丁晖

并发

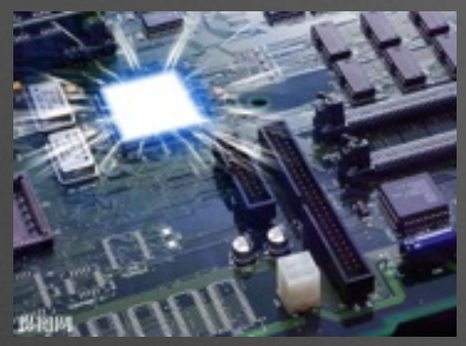
- 并发的常识
- cpu和操作系统
- 并发的问题
- 并发的理论

并发的常识

- 单个主频4GHZ的cpu处理单个进程的效率高于4个主频1GHZ的cpu并发Partition运行
- 由于材料，工艺和功耗等原因，单个cpu性能难以做大幅度突破
- 对于分时多道操作系统，多核cpu具有调度优势（内核调度，中断绑定）
- 好的并发程序要求最大化的利用多核cpu的资源，同时保证运行的性能和正确性

CPU和操作系统

无状态执行指令



PC寄存器 进程2PC

页表寄存器 进程2PT

cpu1

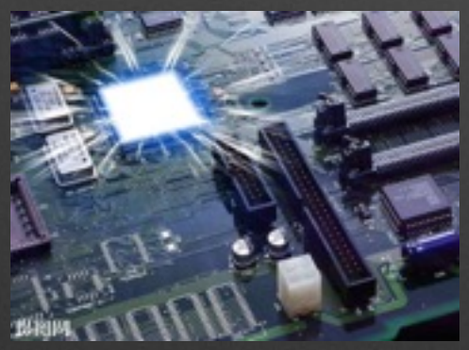
进程1 pcb

进程2 pcb

内核per-cpu变量

CPU1

时钟中断（内核态）



CPU2

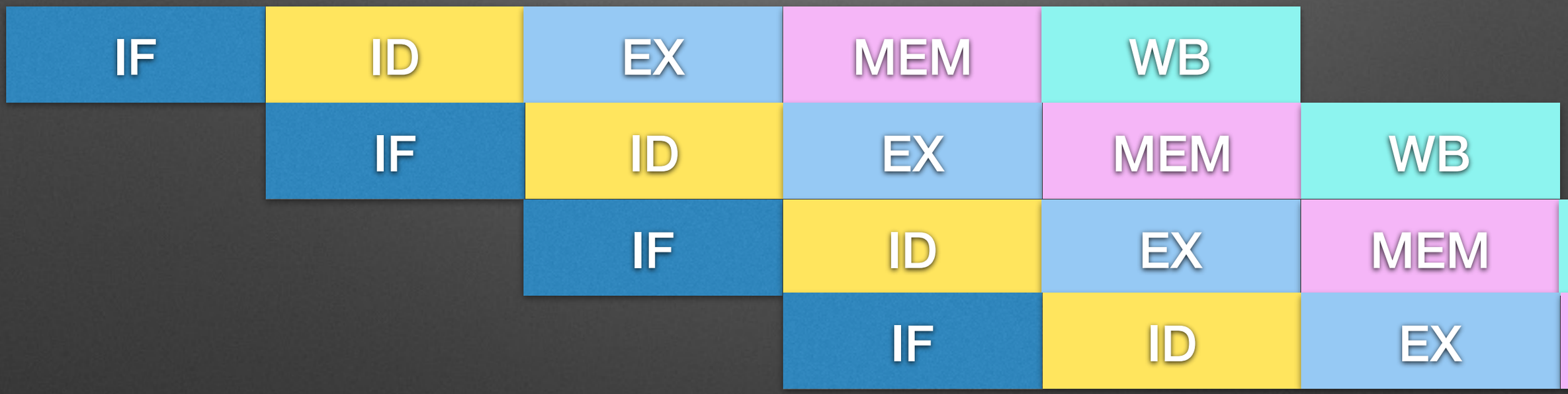
cpu2

内核per-cpu变量

cpu指令周期

- 取指(IF)　取PC寄存器指向的指令
- 译码(ID) 确定执行类别，判断跳转分支
- 执行(EX) 计算有效地址，进行操作数计算
- 访存(MEM) load | store
- 回写(WB) 将EX结果，load结果写入指定寄存器

CPU流水线



指令乱序（编译乱序）

```
int demo(int x){
   int d=3*2;

   if(unlikely(x)){
       x = 5;
       x = x * x;
    }else{
       x=6;
    }

    return x;
}
```

```
Disassembly of section .text:

00000000 <demo>:
0:   55                      push   %ebp
1:   89 e5                   mov    %esp,%ebp
3:   8b 45 08                mov    0x8(%ebp),%eax
6:   85 c0                   test   %eax,%eax
8:   75 07                   jne    16 <demo+0x16>
a:   ba 06 00 00 00          mov    $0x6,%edx
f:   b8 06 00 00 00          mov    $0x6,%eax
14:  c9                      leave
15:  c3                      ret
16:  b8 19 00 00 00          mov    $0x19,%eax
1a:  eb f7                   jmp    14 <demo+0x14>
```

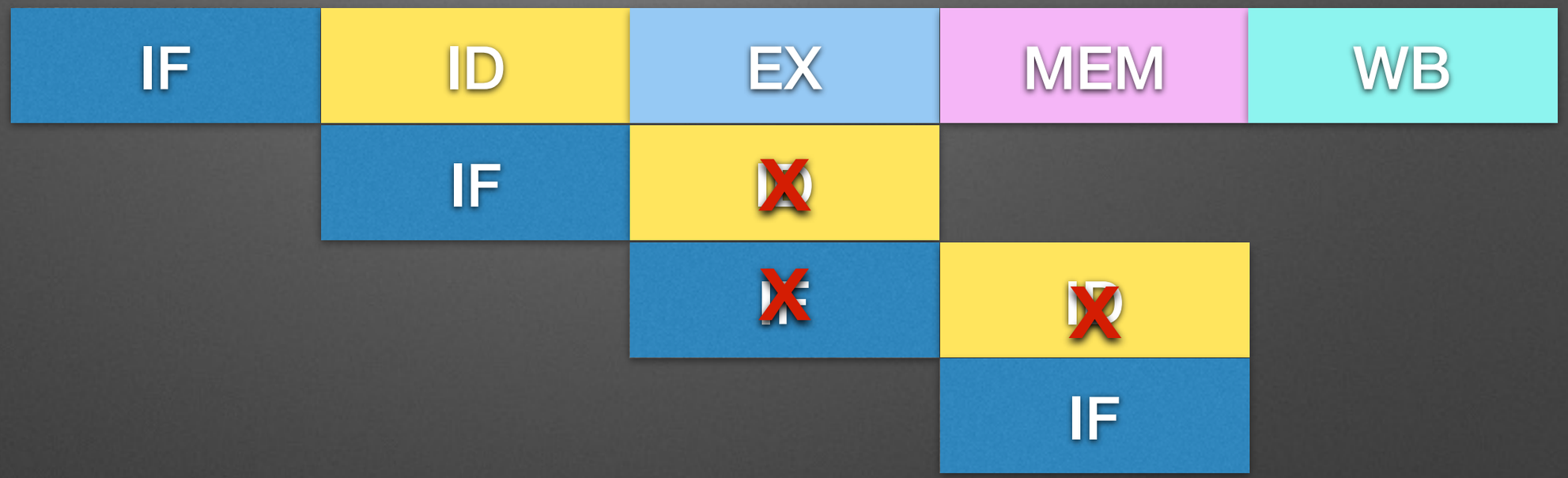
指令乱序（编译乱序）

int d=3*2

x==1

x=6

x=5

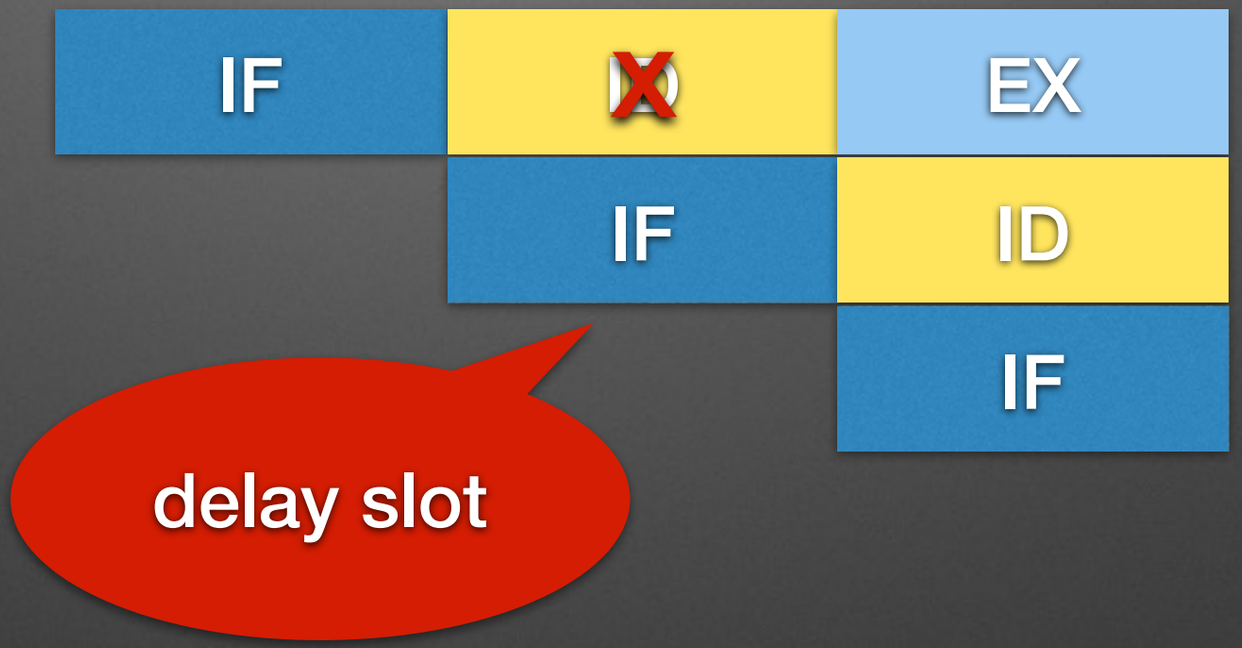


指令乱序（编译乱序）

x==1

int d=3*2

x=5



指令乱序（执行乱序）

```
int b=*a
int c=b+1
int d=6
```

11时钟周期

IF	ID	EX	MEM	MEM	MEM
	IF	ID	STALL	STALL	STALL
		IF	STALL	STALL	STALL

指令乱序（执行乱序）

7时钟周期

指令多发射

int b=*a

int c=b+1

int d=6

int b=*a　LOAD　READY　COMMIT

int c=b+1　EX　READY　COMMIT

int d=6　EX　READY　COMMIT

并发的问题

- 指令乱序
- 缓存一致性
- 伪共享
- 数据竞争

指令乱序

CPU1	CPU1(乱序)	CPU2	CPU1

*a=3
*b=2
*c=true
	*c=true
	*a=3
	*b=2
		if(*c){
		printf(“%d”,
		*a+*b)
		}
			*a=3
			*b=2
			smp_mb()
			*c=true
			smp_mb()

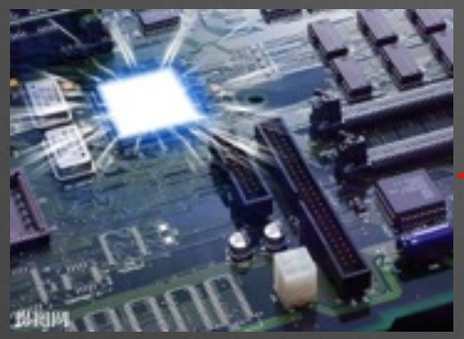
内存屏障

- 全屏障
- 读写屏障
- 写读屏障
- 写写屏障
- 读读屏障

缓存一致性

a=2

volatile a

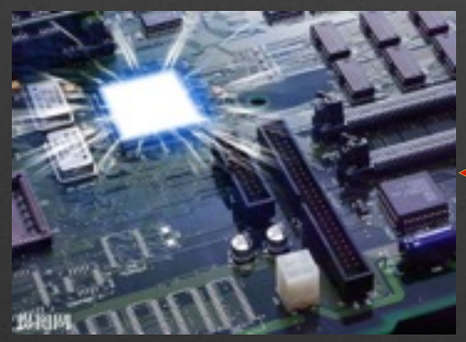


2

Cache

2

b=a

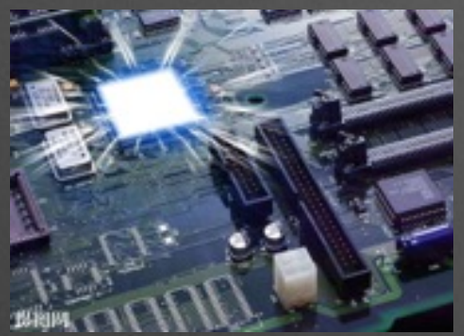


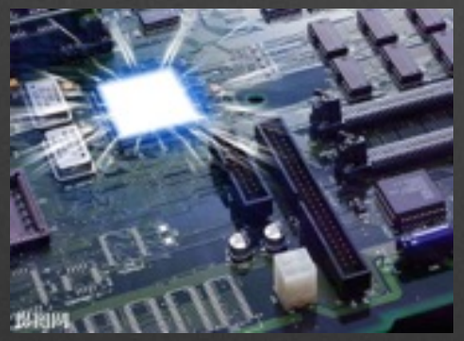
2

Cache

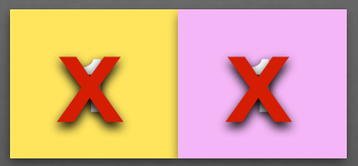
伪共享

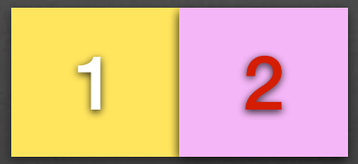
volatile int a,b





cache line





volatile int a,
volatile int alpha[7],
volatile int b

数据竞争

- 哲学家进餐问题
- Counting问题
- 数据结构(java hashmap的死循环问题)
- 解决办法：各种锁或者无锁算法

并发的理论

- 并行与并发比较
- 锁
- 锁同步和通信同步
- 无锁的一致性实现 （RCU）

并行

- 针对运算数据本身进行partition
- 数据竞争，数据通信少
- 执行体行为基本一致
- 根据cpu密集或是IO密集确定执行体个数
- map reduce或者是爬虫服务等

并发

- 针对服务流水线各个阶段进行partition
- 数据竞争，数据通信多，根据应用确定一致性标准
- 流水线各阶段执行体行为基本不同
- 根据各阶段服务压力确定执行体个数（C10K）
- 双十一，秒杀等

锁的类别（互斥锁）

- 一个时间点内通常只能有一个执行体进入临界区
- 事务临界区（同步控制，事务原子性控制）
- 资源临界区（计数和资源调度）

锁的类别（读写锁）

- 读读并行，读写互斥
- 适用于读多写少的应用场景

锁的实现（挂起）

DougLea的Java并发库基于同样原理的AQS实现Lock

锁的实现（spinLock）

- 一直尝试获取锁资源（不挂起）

```
void lock(){
    while(!AtomicBoolean.compareAndSet(False,True)){
                                        //循环
    }
}
```

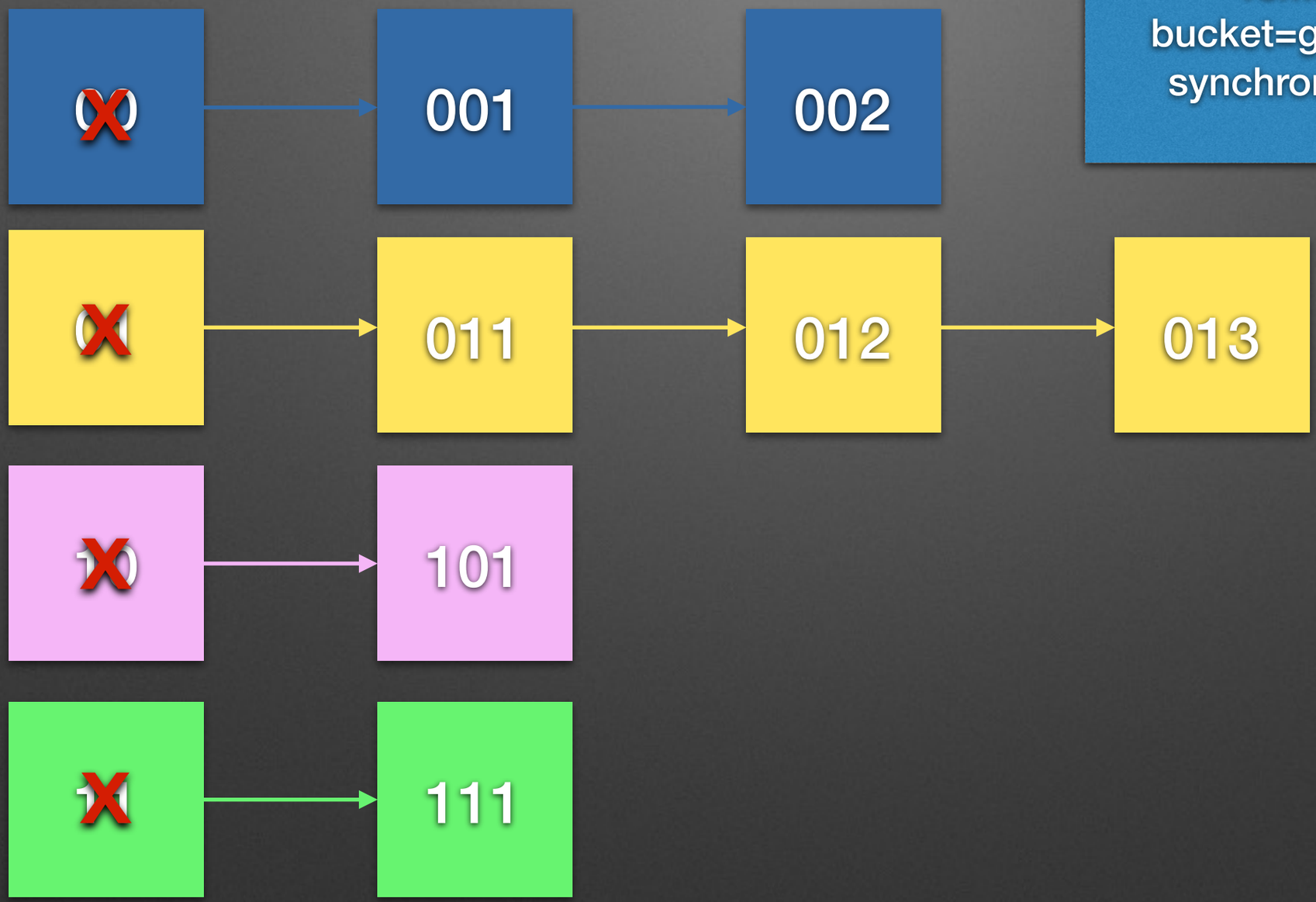
- 基于原子变

- 临界区执行时间短，最好无阻塞

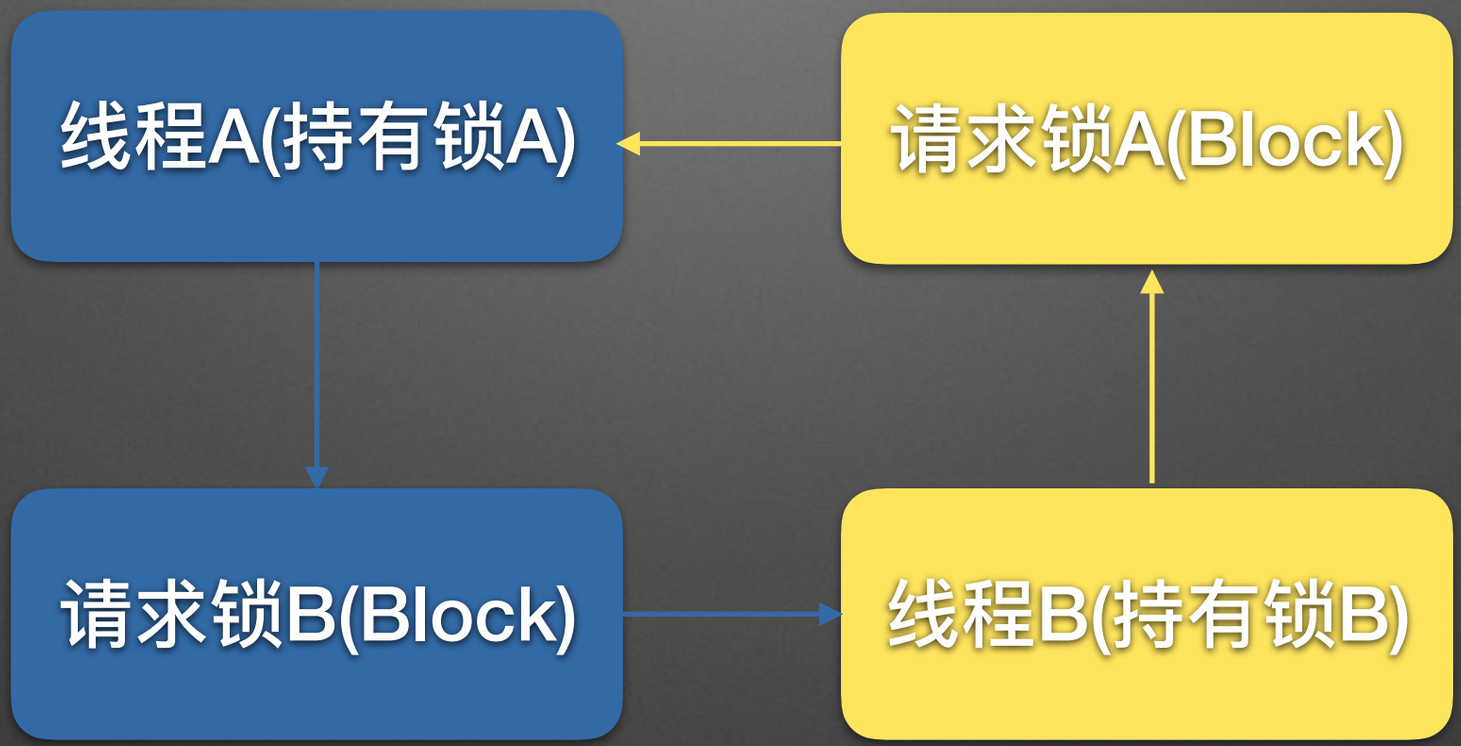
- cpu占用高，但是系统调用和资源切换少，性能更高

锁的粒度

数据锁
remove(011){
bucket=getBucket(011)
synchronize(bucket){}
}



死锁



One Tips For Avoiding DeadLock:

Release all locks before invoking unknown code

锁同步和通信同步

两个人同时吃羊，一共100只羊，吃完为止

两个线程通过锁实现同步操作共享变量

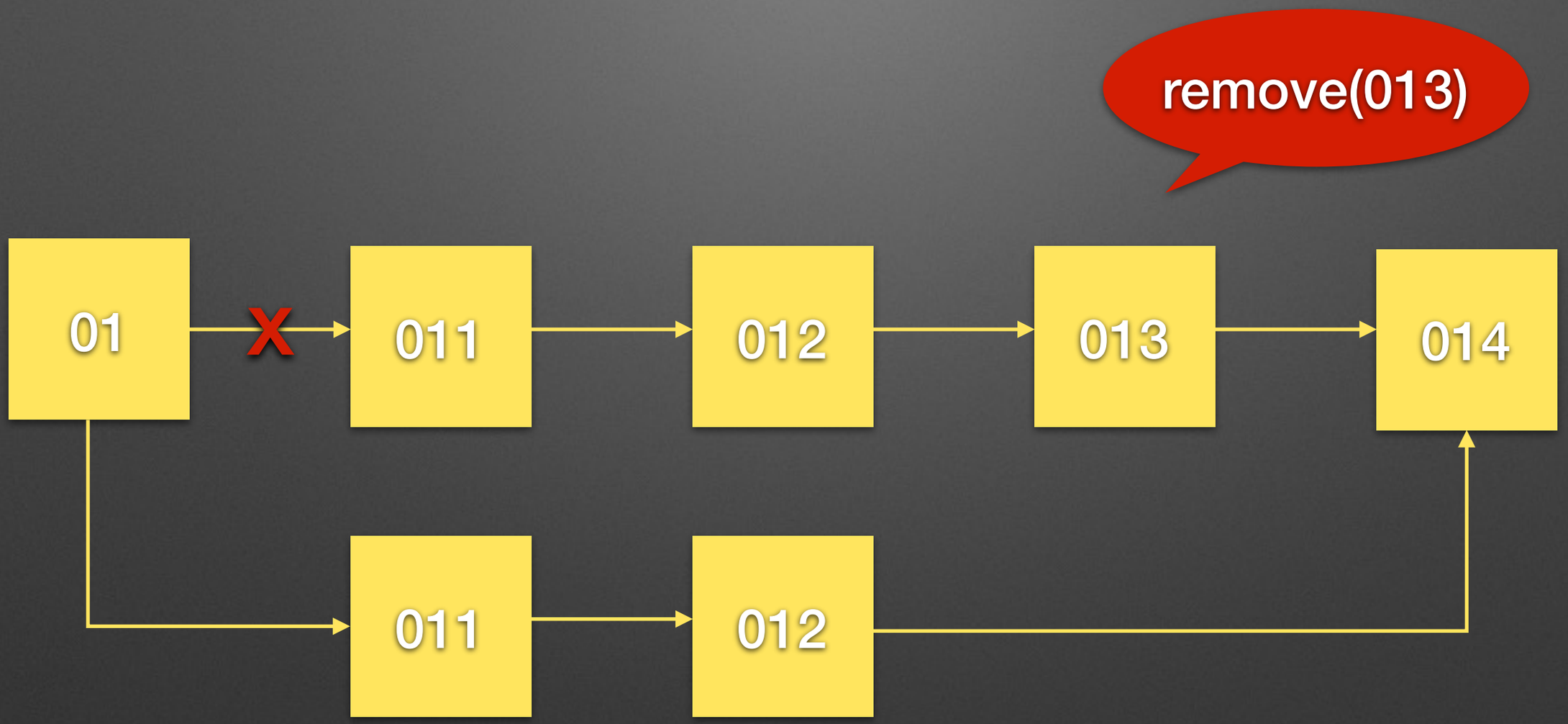
引入三个线程则可转化为通信问题

通信同步(golang)

```
1
2  func consumer(channel chan chan int){
3      xchannel := make(chan int)
4      for{
5          xchannel->channel
6
7          r := <-xchannel
8
9          if r==0 {
10             break
11         }else{
12             fmt.Printf("%d\n",r)
13         }
14     }
15 }
16 func resource(channel chan chan int){
17     R := 100
18     for{
19         xchannel := <-channel
20         if R==0 {
21             0 -> xchannel
22         }else{
23             R -> xchannel
24             R = R-1
25         }
26     }
27 }
28 func main(){
29     channel := make(chan chan int, 2)
30     go consumer(channel)
31     go consumer(channel)
32     resource(channel)
33 }
```

RCU（无锁）

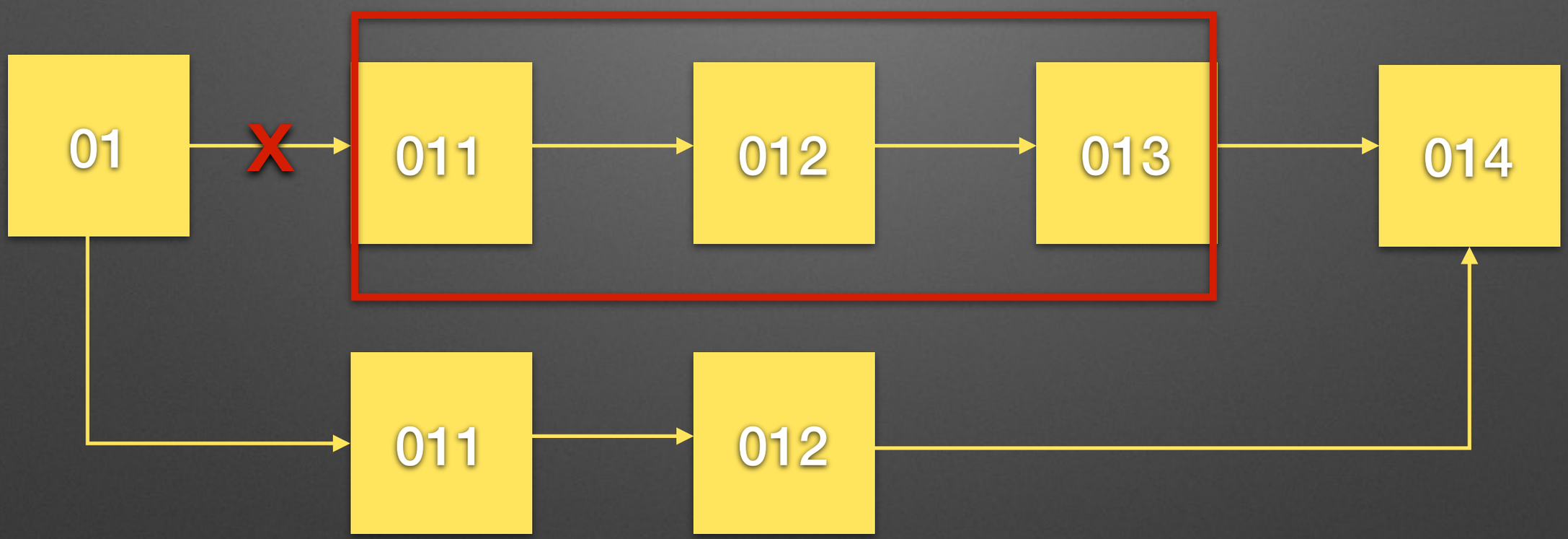
Read Copy Update



ConcurrentHashMap

内存回收问题

内存回收(GC)



内存回收问题(时序)

- rcuLock注册读开始顺序(AtomicInt)
- rcuUnlock通知读结束
- 写操作注册写开始顺序，注册回调函数释放内存
- 在写操作之前的读操作全部结束后执行回调函数

内存回收问题(时序)

读A

读B

写A

读C

回收内存

SEQ3

rculock

SEQ1

rculock

SEQ2

rculock

SEQ4

SEQ1

SEQ2

rcuunlock

rcuunlock

RCU（无锁）

- 读写并行，写写互斥
- 通过数据拷贝完成数据修改，同一时刻可能存在多份数据
- 寻求最终一致性以换取读写性能
- 需要妥善处理资源回收问题

体会

- 并发不是技术手段，是一种思想
- 并发的载体和元素构成各种各样
- 并发的设计要根据业务场景来定，没有绝对完美的并发
- 站在技术之上看技术

3X