

Java程序员需要掌握的 计算机底层知识 v1.0

微机原理 计算机组成原理 操作系统 汇编语言

马士兵

revision

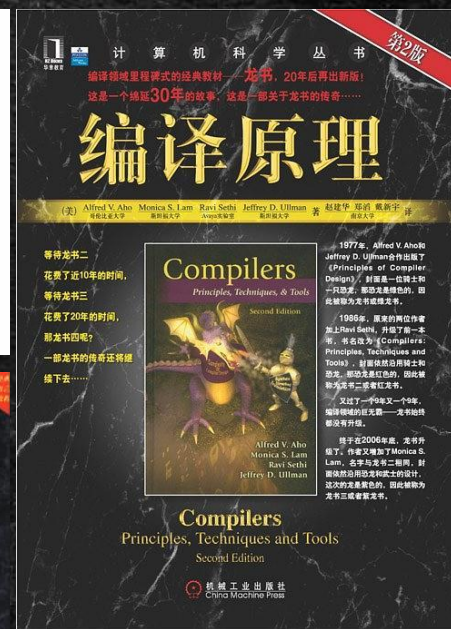
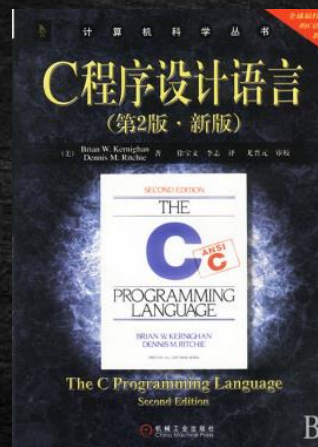
- v1 – 2020/03/08 by 马士兵

Content

- 硬件基础知识
 - Java相关硬件知识
- 汇编语言的执行过程 (时钟发生器 寄存器 程序计数器)
- 计算机启动过程
- 操作系统基本知识
- 进程线程纤程的基本概念 (面试高频)
 - 纤程的实现
- 内存管理
- 进程管理与线程管理 (进程与线程在Linux中的实现)
- 中断与系统调用 (软中断)
- 内核同步基础知识
- 关于硬盘 IO DMA

假如我是一个计算机系讲师

- 《编码：隐匿在计算机软硬件背后的语言》
- 《深入理解计算机系统》
- 语言：C JAVA
- 数据结构与算法：
 - 《Java数据结构与算法》《算法》
 - 《算法导论》《计算机程序设计艺术》
- 操作系统：Linux内核源码解析 30天自制操作系统
- 网络：机工《TCP/IP详解》卷一 翻译一般
- 编译原理：机工 龙书 编程语言实现模式
- 数据库：SQLite源码 Derby



硬件基础知识

关于底层的细节：

适度打开

很多情况下保持黑箱即可，因为打开这个黑箱，你就会发现黑箱会变成黑洞，吞噬你所有的精力和时间，有可能使你偏离原来的方向，陷入到不必要的细节中无法自拔。

cpu的制作

一堆沙子 + 一堆铜 + 一堆胶水 + 特定金属添加 + 特殊工艺

沙子脱氧 -> 石英 -> 二氧化硅 -> 提纯 -> 硅锭 -> 切割 ->
晶圆 -> 涂抹光刻胶 -> 光刻 -> 蚀刻 -> 清除光刻胶 -> 电镀
-> 抛光 -> 铜层 -> 测试 -> 切片 -> 封装

Intel cpu的制作过程

<https://haokan.baidu.com/v?vid=11928468945249380709&pd=bjh&fr=bjhauthor&type=video>

CPU是如何制作的

https://www.sohu.com/a/255397866_468626

硅 -> 加入特殊元素 -> P半导体 N半导体 -> PN结 -> 二极管 -> 场效应晶体管
-> 逻辑开关

与门 或门 非门 或非门 (异或) -> 基础逻辑电路

加法器 累加器 锁存器...

实现手动计算 (通电一次, 运行一次位运算)

加入内存 实现自动运算 (每次读取内存指令, (高电低电))

《编码》17章

晶体管是如何工作的:

<https://haokan.baidu.com/v?vid=16026741635006191272&pd=bjh&fr=bjhauthor&type=video>

晶体管的工作原理:

<https://www.bilibili.com/video/av47388949?p=2>

继续：

让计算机看懂计算：01000010 + 00101100 ..

手工输入：纸带计算机

助记符：01000010 – mov sub ...

高级语言 -> 编译器 -> 机器语言

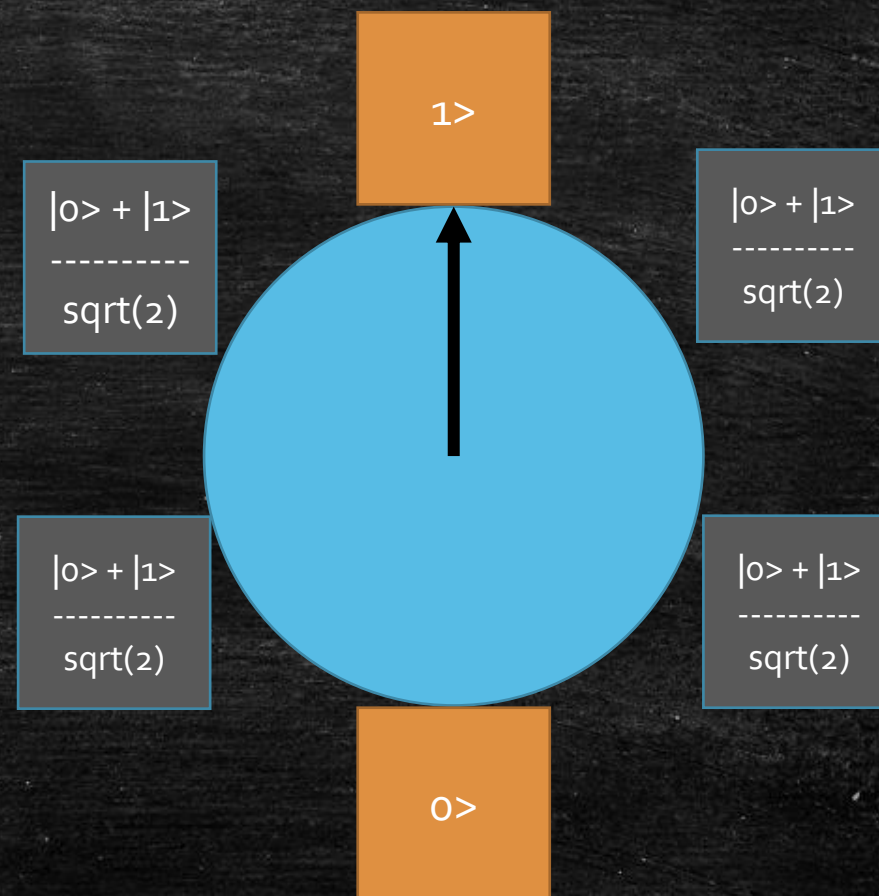
汇编执行过程

计算机通电 -> CPU读取内存中程序（电信号输入）
->时钟发生器不断震荡通断电 ->推动CPU内部一步一步执行
（执行多少步取决于指令需要的时钟周期）
->计算完成->写回（电信号）->写给显卡输出（sout, 或者图形）

01001000 mov
10110011 add sub
汇编的本质：助记符

量子计算机

量子比特



同时表示

1位可以同时表示 1 0

2位可以同时表示 00 01 10 11

3位可以同时表示 000 001 ...

...

18位可以同时表示 2^{18} 个数

比如你要猜一个数，原来只能一个一个猜

而现在，你可以在瞬时同时暴力测试 2^{18} 个数

...

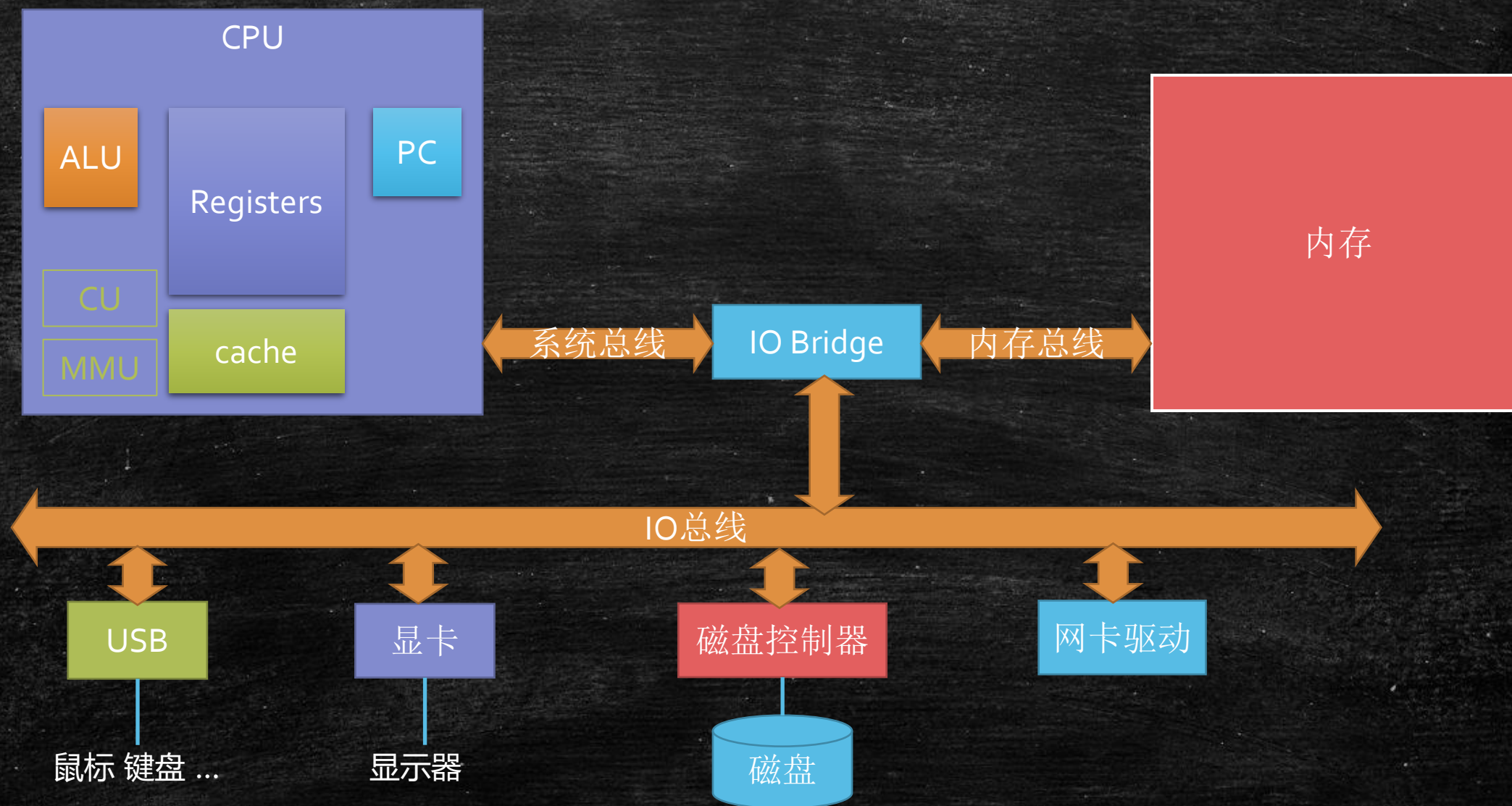
而很多密码学算法是根据单向函数的原理
认为不可短时间穷举密钥而设计的

...

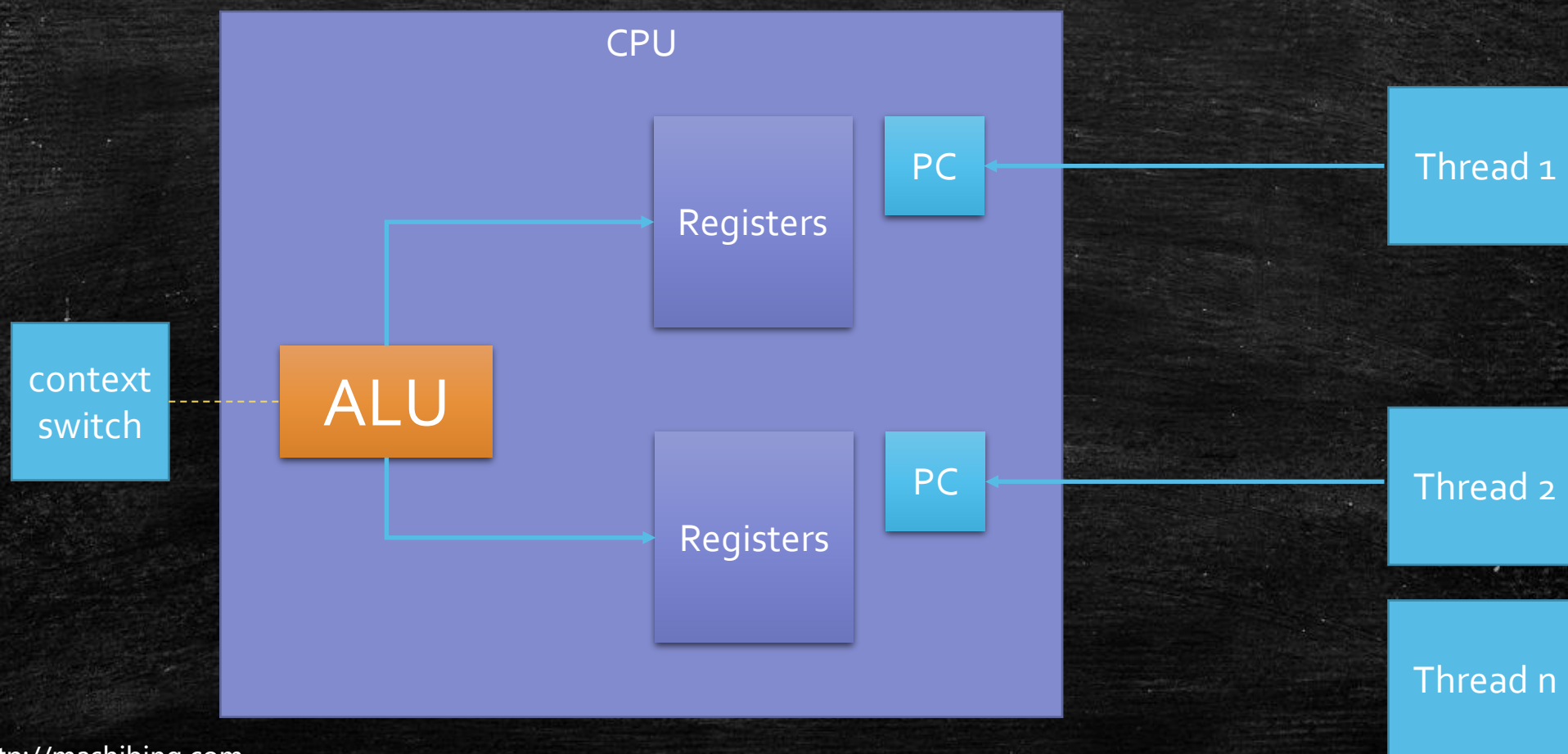
Java相关硬件知识

cpu和内存，是计算机的核心

计算机的组成



超线程：一个ALU对应多个PC | Registers
所谓的四核八线程



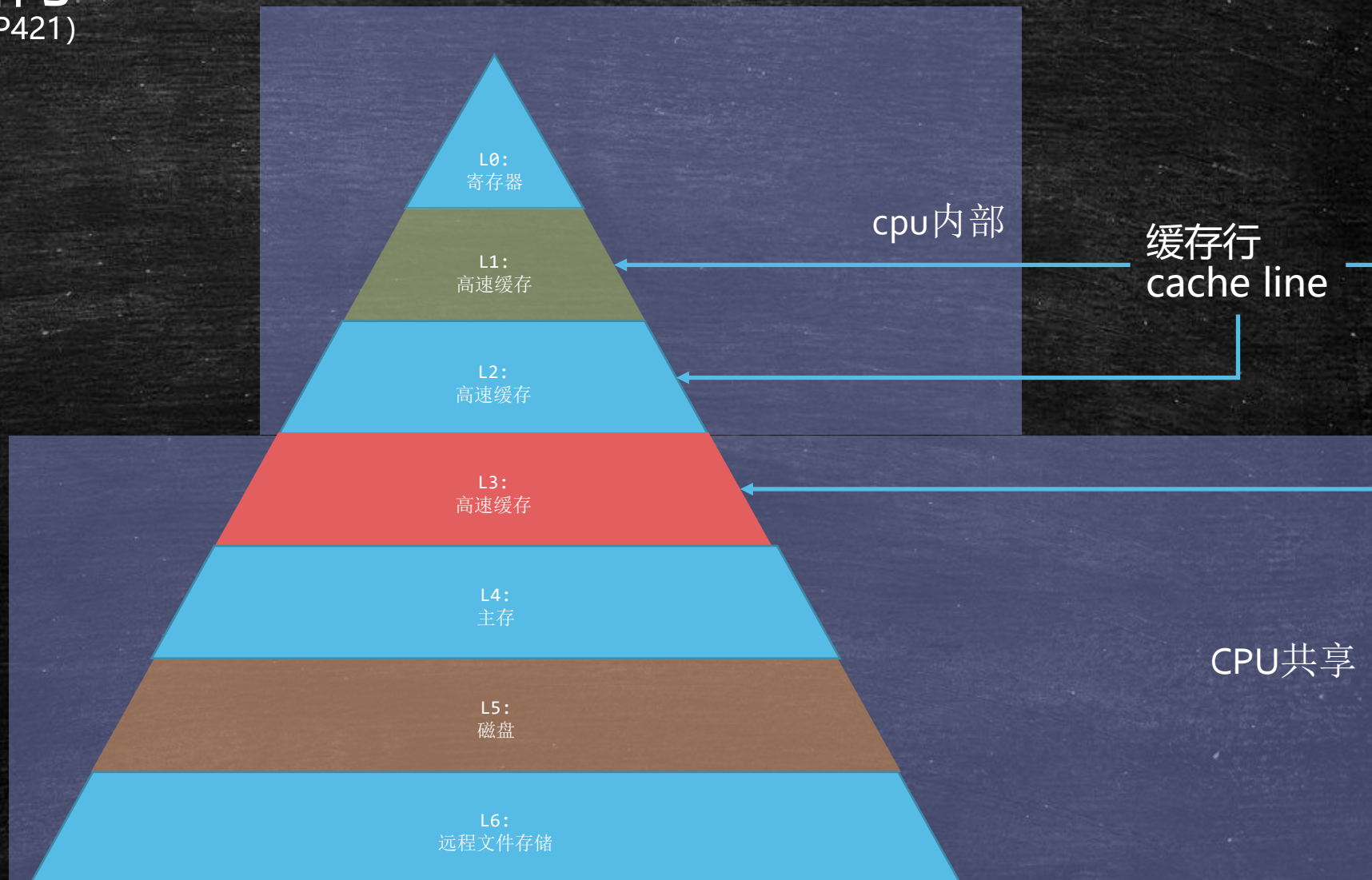
存储器的层次结构

(深入理解计算机系统 原书第三版 P421)

更小 更快 成本更高



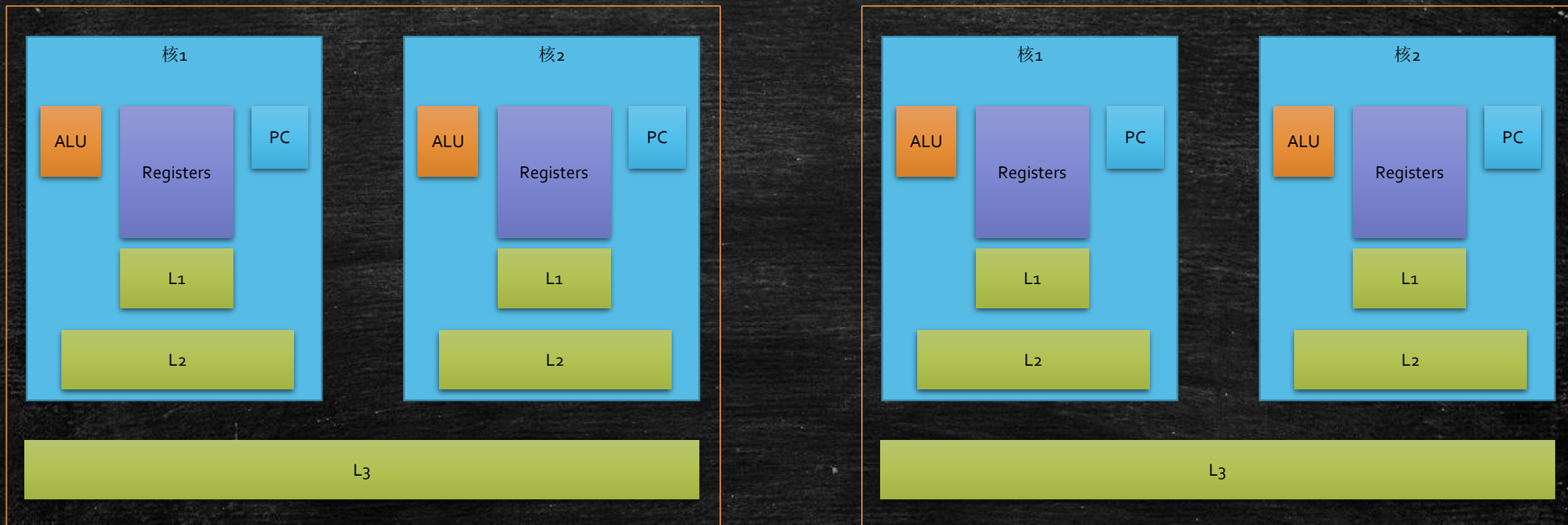
更大 更慢 成本更低



从CPU的计算单元 (ALU) 到:

Registers	< 1ns
L1 cache	约1ns
L2 cache	约3ns
L3 cache	约15ns
main memory	约80ns

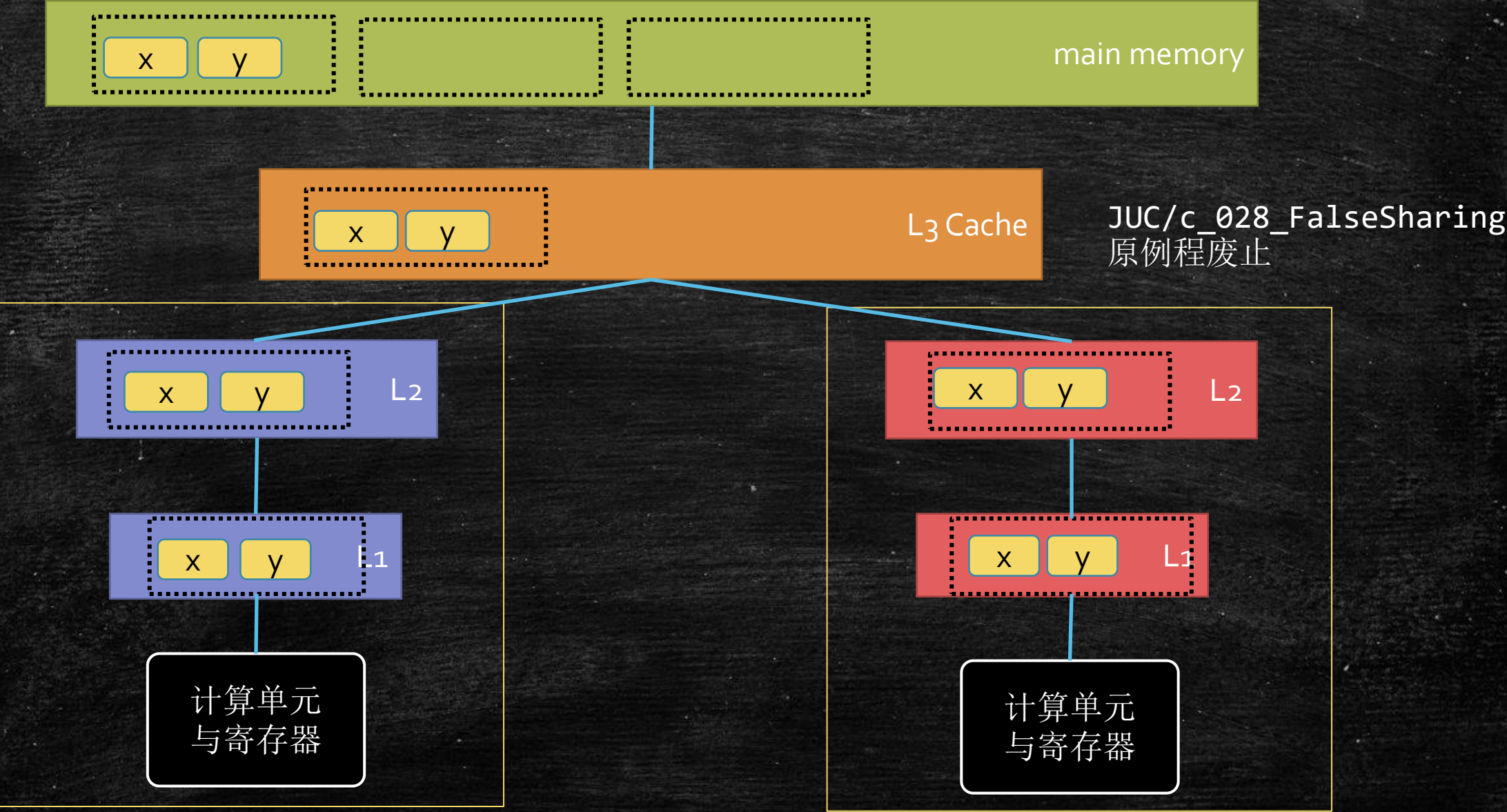
多核CPU



内存

按块读取
程序局部性原理，可以提高效率
充分发挥总线 CPU针脚等一次性读取更多数据的能力

cache line的概念 缓存行对齐 伪共享



缓存行：

缓存行越大，局部性空间效率越高，但读取时间慢

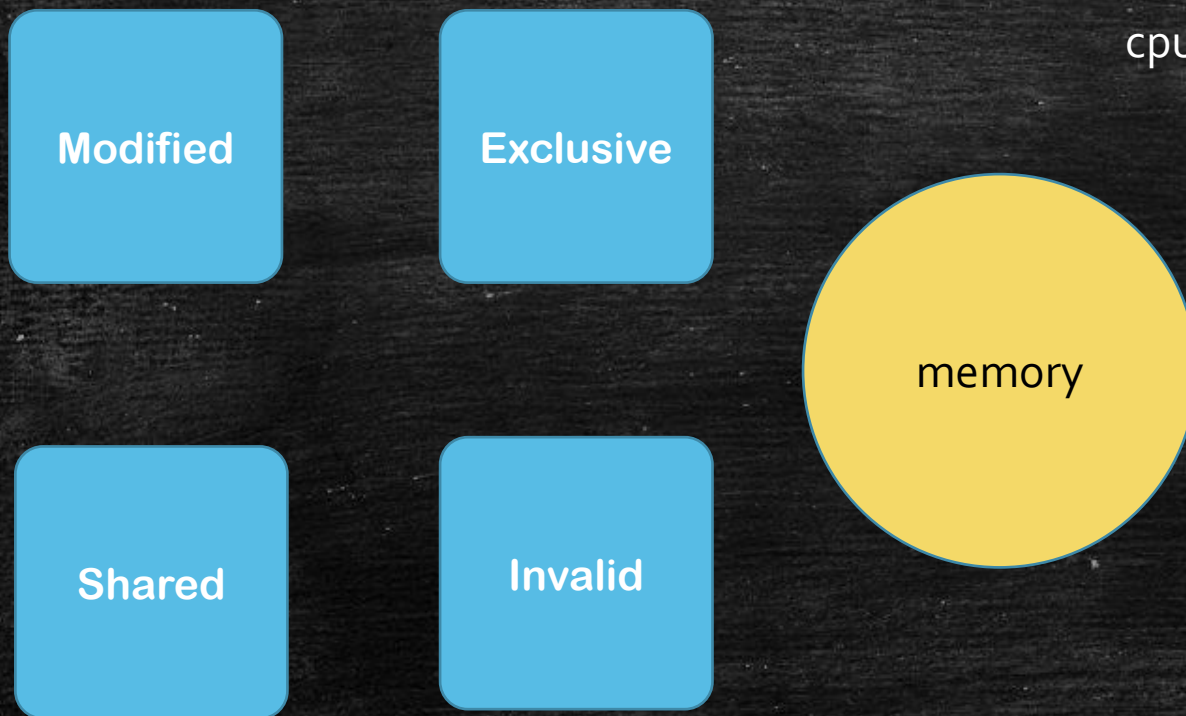
缓存行越小，局部性空间效率越低，但读取时间快

取一个折中值，目前多用：

64字节

MESI Cache一致性协议

<https://www.cnblogs.com/zoo377750/p/g180644.html>



cpu每个cache line标记四种状态（额外两位）

缓存锁实现之一
有些无法被缓存的数据
或者跨越多个缓存行的数据
依然必须使用总线锁

MSI MESI MOSI Synapse Firefly Dragon

disruptor

```
public long p1, p2, p3, p4, p5, p6, p7; // cache line padding  
    private volatile long cursor = INITIAL_CURSOR_VALUE;  
    public long p8, p9, p10, p11, p12, p13, p14; // cache line padding
```


CPU的乱序执行

CPU在进行读等待的同时执行指令，是CPU乱序的根源
不是乱，而是提高效率



乱序执行的证明: jvm/jmm/Disorder.java

<https://preshing.com/20120515/memory-reordering-caught-in-the-act/>

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
```

```
第2728842次 (0,0)
```

```
Process finished with exit code 0
```

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
```

```
第113299次 (0,0)
```

```
Process finished with exit code 0
```

乱序执行可能会产生问题：

DCL单例为什么要加volatile

1: 对象的创建过程

源码:

```
class T {  
    int m = 8;  
}
```

```
T t = new T();
```

汇编码:

```
0 new #2 <T>  
3 dup  
4 invokespecial #3 <T.<init>>  
7 astore_1  
8 return
```

t

m = 8

2:加问: DCL单例 (Double Check Lock) 到底需不需要volatile?



CPU层面如何禁止重排序？

答：内存屏障

对某部分内存做操作时前后添加的屏障
屏障前后的操作不可以乱序执行

指令1 – 去内存读数据

指令2 – 优先执行

intel:

lfence mfence sfence 原语
当也可以使用总线锁来解决

内存屏障
指令1 2不可互换

有序性保障

X86 CPU内存屏障

sfence:在sfence指令前的写操作当必须在sfence指令后的写操作前完成。
lfence: 在lfence指令前的读操作当必须在lfence指令后的读操作前完成。
mfence: 在mfence指令前的读写操作当必须在mfence指令后的读写操作前完成。

有序性保障 intel lock汇编指令

原子指令，如x86上的” lock ...” 指令是一个**Full Barrier**，执行时会锁住内存子系统来确保执行顺序，甚至跨多个CPU。**Software Locks**通常使用了内存屏障或原子指令来实现变量可见性和保持程序顺序

JSR内存屏障

LoadLoad屏障:

对于这样的语句Load1; LoadLoad; Load2,

在Load2及后续读取操作要读取的数据被访问前, 保证Load1要读取的数据被读取完毕。

StoreStore屏障:

对于这样的语句Store1; StoreStore; Store2,

在Store2及后续写入操作执行前, 保证Store1的写入操作对其它处理器可见。

LoadStore屏障:

对于这样的语句Load1; LoadStore; Store2,

在Store2及后续写入操作被刷出前, 保证Load1要读取的数据被读取完毕。

StoreLoad屏障: 对于这样的语句Store1; StoreLoad; Load2,

在Load2及后续所有读取操作执行前, 保证Store1的写入对所有处理器可见。

volatile的实现细节 二：JVM层面

---- StoreStoreBarrier ----

volatile 写

---- StoreLoadBarrier ----

volatile 读

---- LoadLoadBarrier ----

---- LoadStoreBarrier ----

happens-before原则（JVM规定重排序必须遵守的规则）

JLS17.4.5

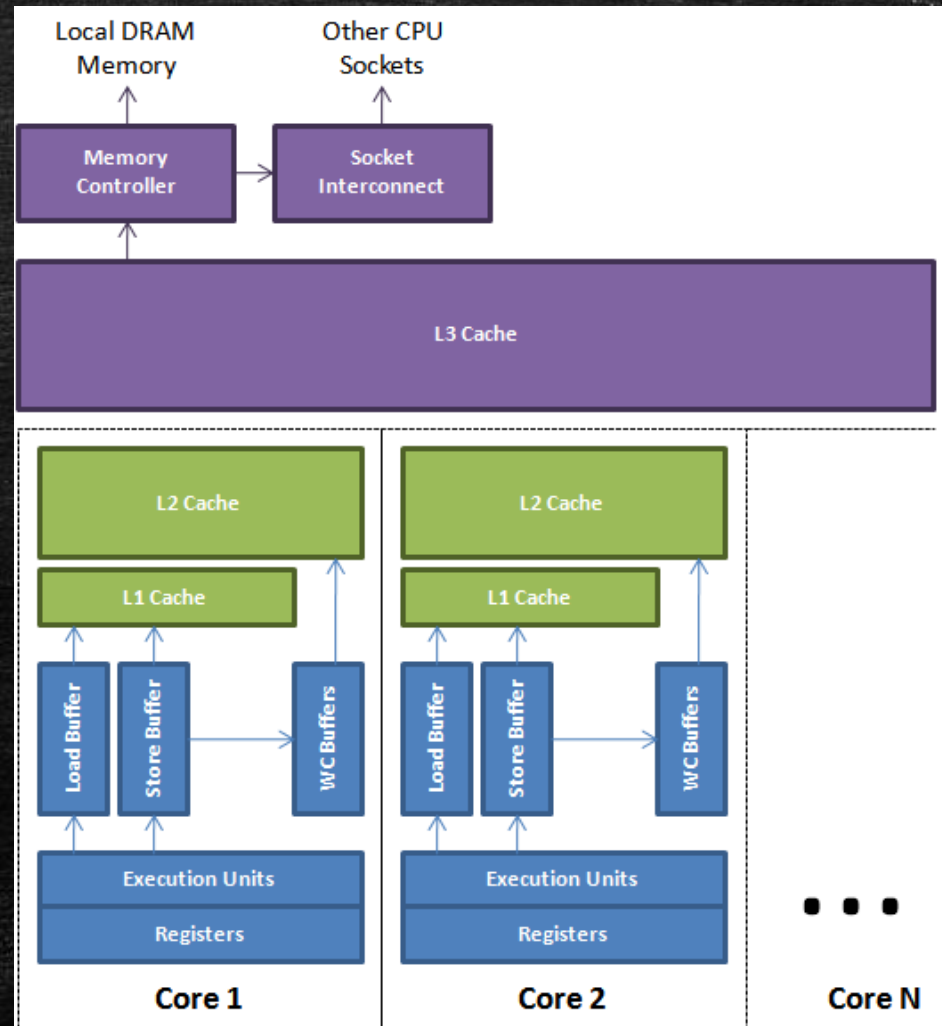
- 程序次序规则：同一个线程内，按照代码出现的顺序，前面的代码先行于后面的代码，准确的说是控制流顺序，因为要考虑到分支和循环结构。
- 管程锁定规则：一个unlock操作先行发生于后面（时间上）对同一个锁的lock操作。
- volatile变量规则：对一个volatile变量的写操作先行发生于后面（时间上）对这个变量的读操作。
- 线程启动规则：Thread的start()方法先行发生于这个线程的每一个操作。
- 线程终止规则：线程的所有操作都先行于此线程的终止检测。可以通过Thread.join()方法结束、Thread.isAlive()的返回值等手段检测线程的终止。
- 线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupt()方法检测线程是否中断
- 对象终结规则：一个对象的初始化完成先行于发生它的finalize()方法的开始。
- 传递性：如果操作A先行于操作B，操作B先行于操作C，那么操作A先行于操作C

as if serial

不管如何重排序，单线程执行结果不会改变

WC - Write Combining 合并写技术

为了提高写效率：CPU在写入L1时，同时用WC写入L2

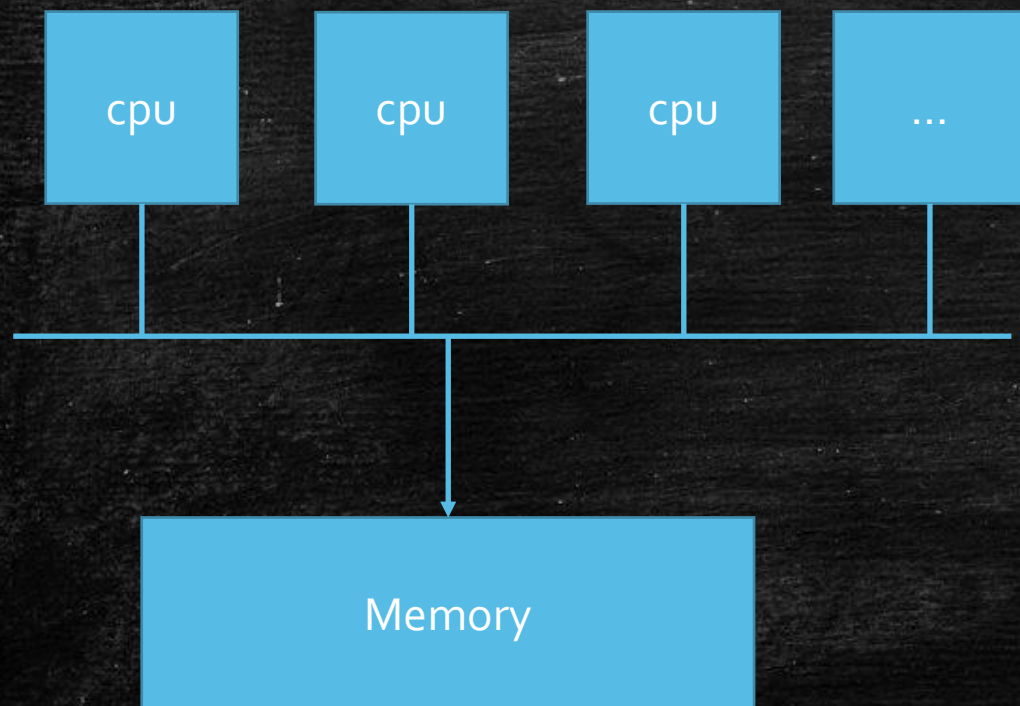


JUC/c029

ZGC的NUMA Aware

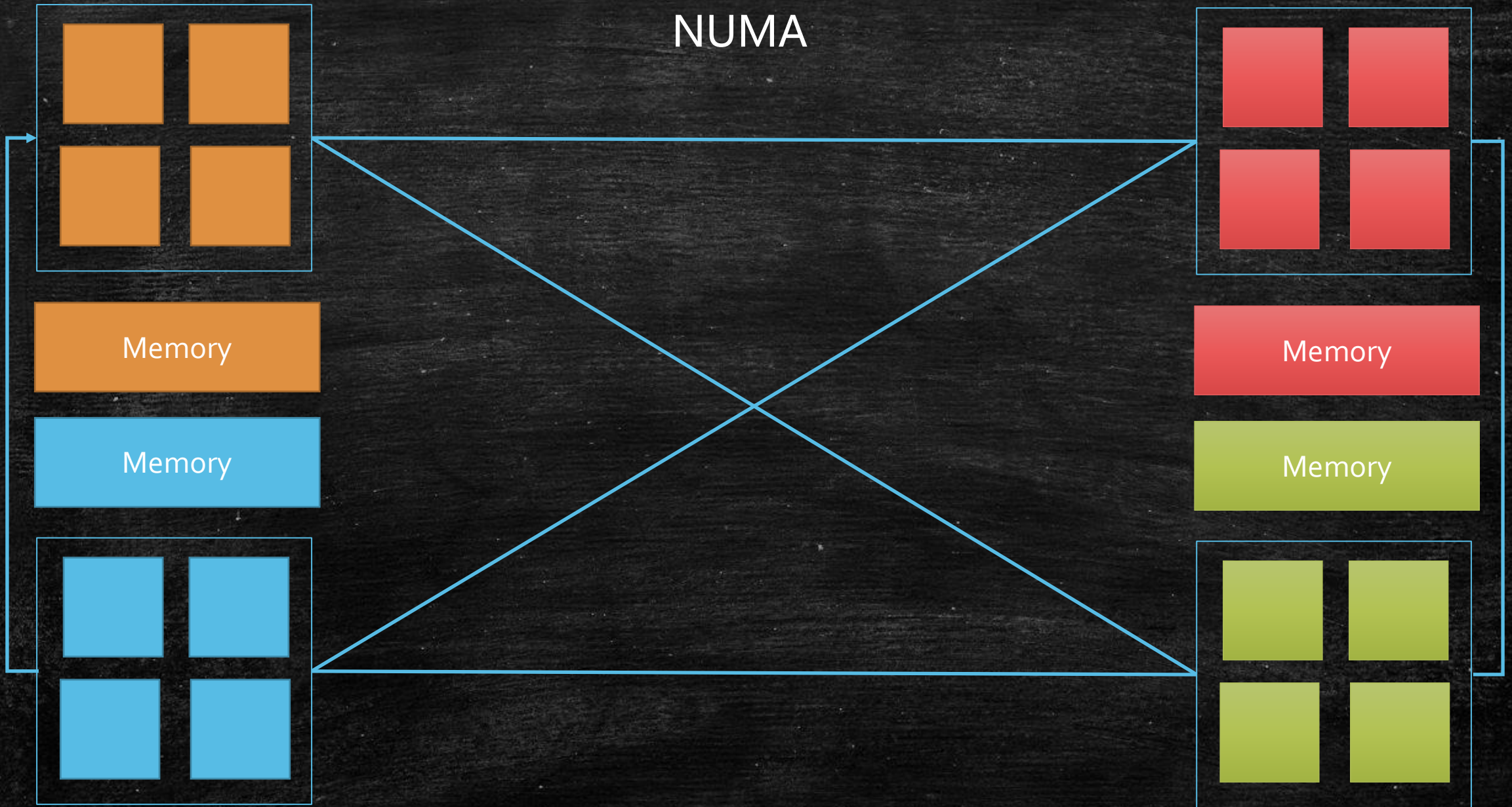
Non Uniform Memory Access

UMA



不易扩展
CPU数量增多后引起内存访问冲突加剧
CPU的很多资源花在争抢内存地址上面
4颗左右比较合适

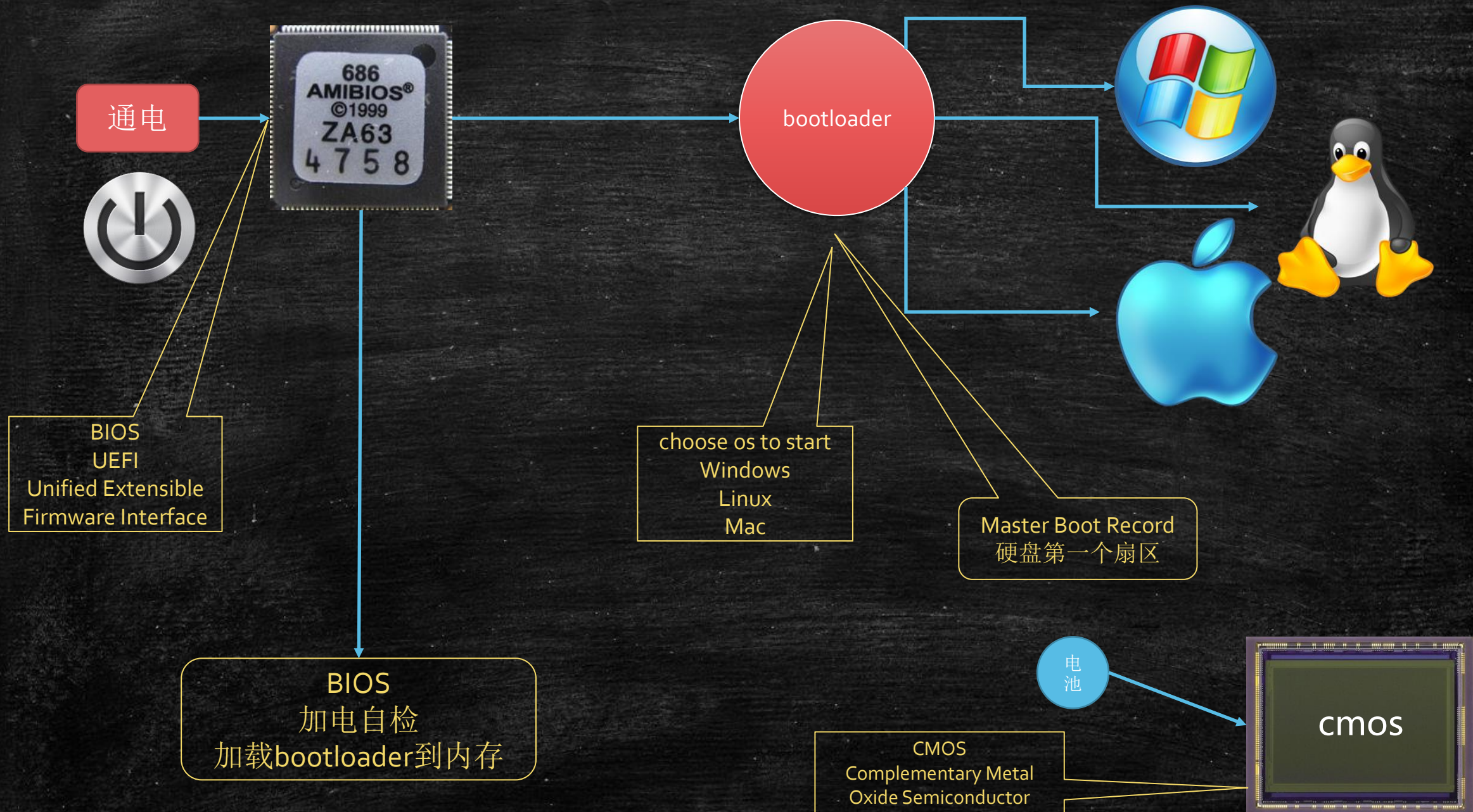
NUMA



ZGC NUMA-Aware

在分配内存的时候优先在当前线程所在CPU组的内存进行分配
可以显著提高效率

启动



OS基础

鸿蒙

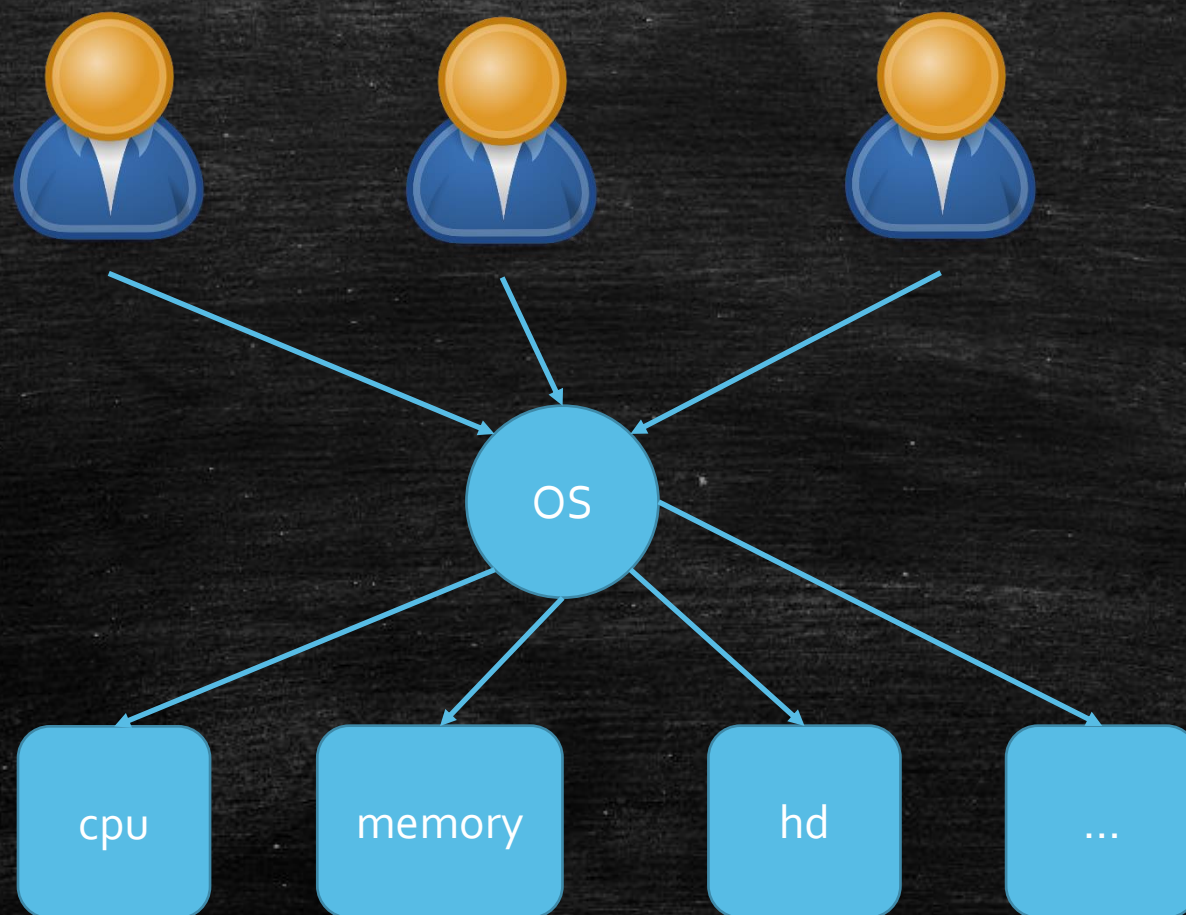


harmony

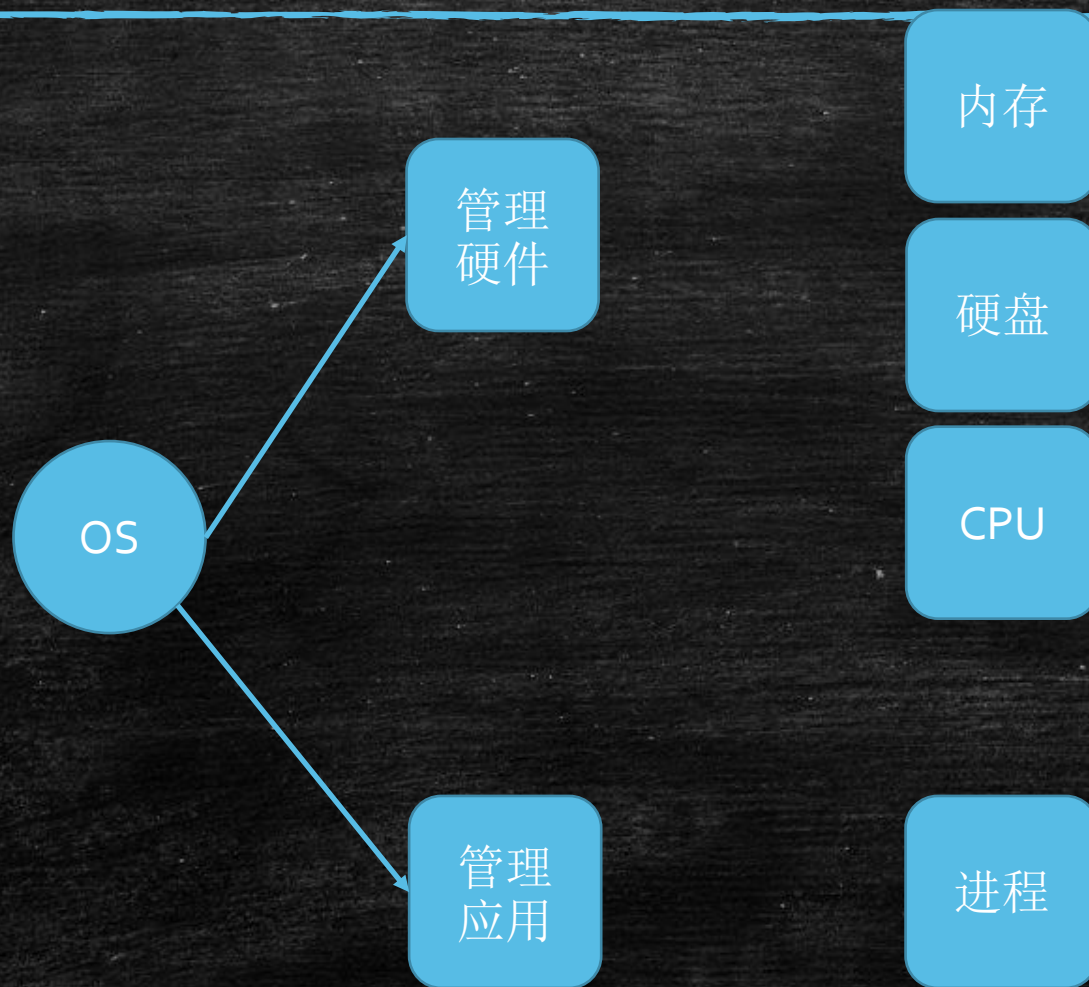
- 微内核
 - 针对5G + IoT
 - 全场景
 - 手机 PC 平板 车辆 智能穿戴 家居设备...
 - 弹性部署
 - 开源
 - 方舟编译器
 - 混合内核 Linux + LiteOS + ...

简介

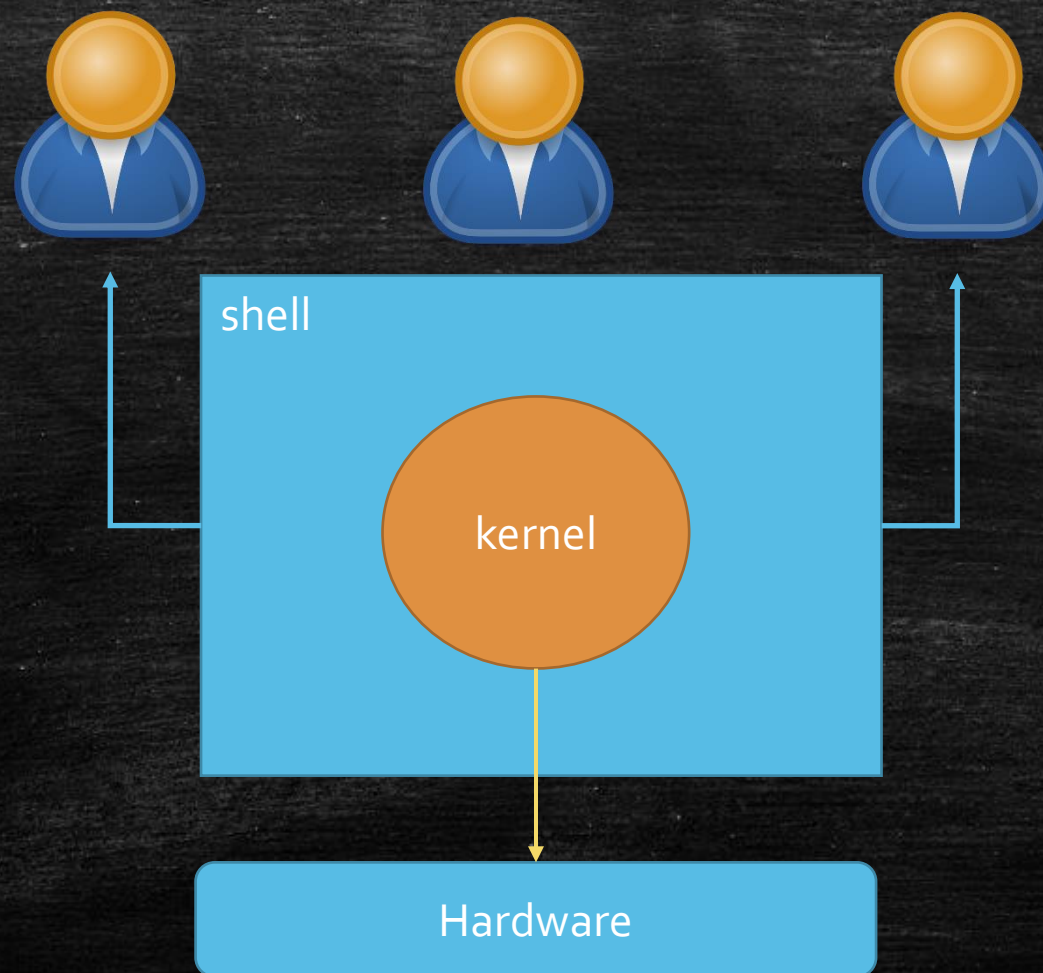
什么是操作系统



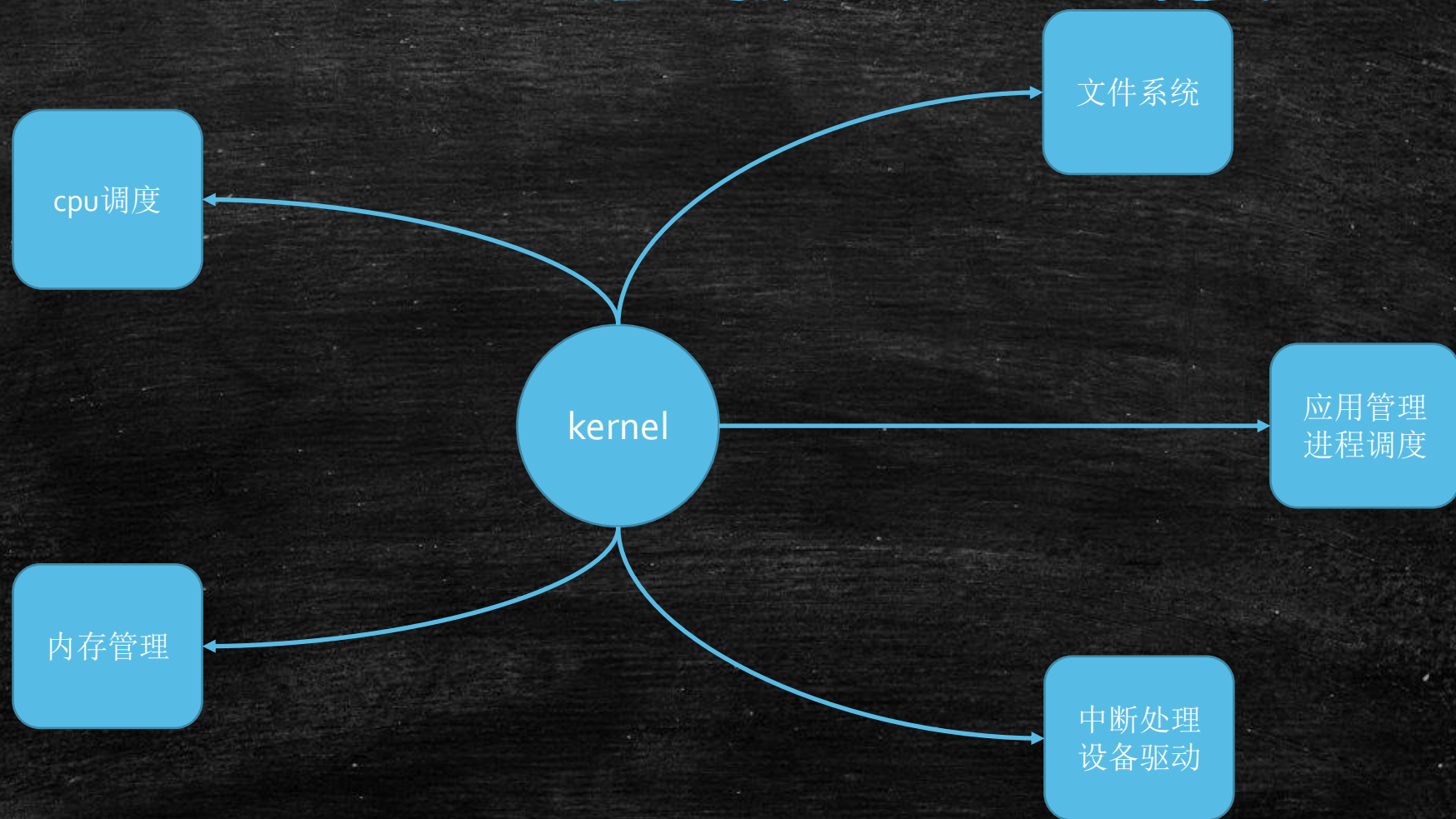
操作系统主要做什么



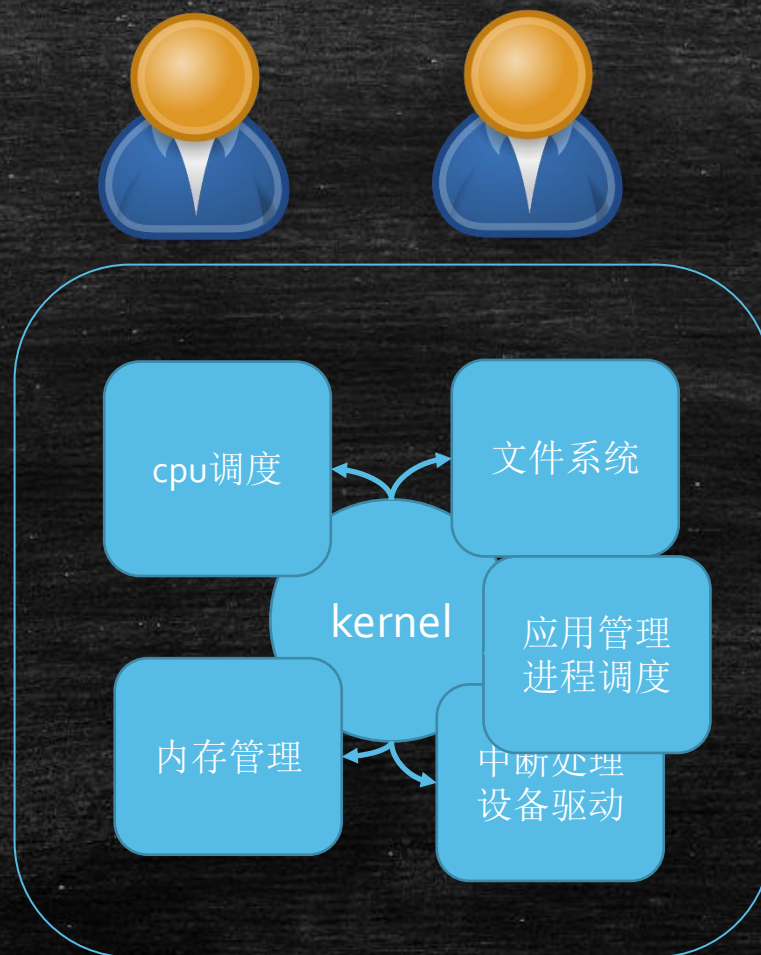
简要结构



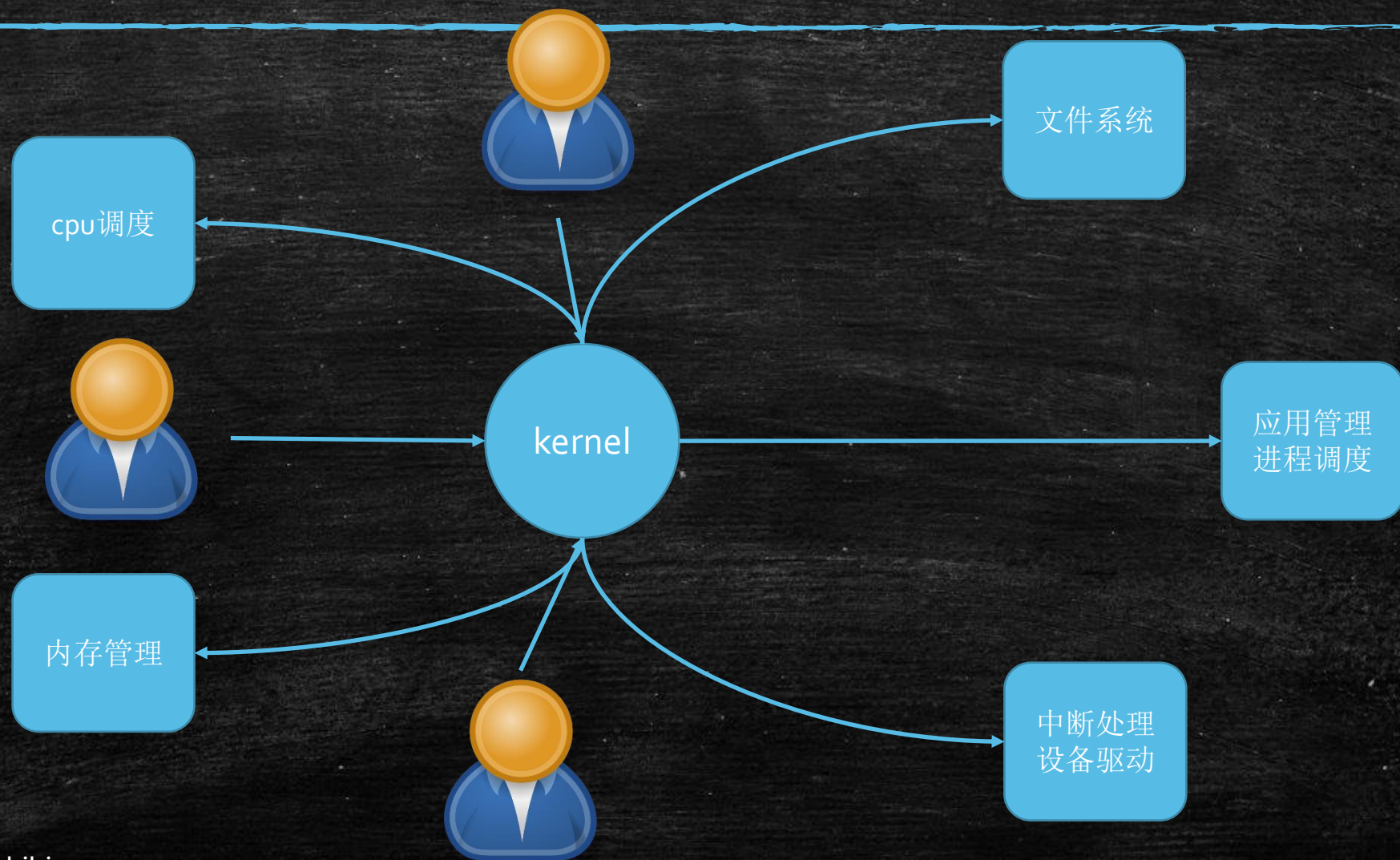
kernel



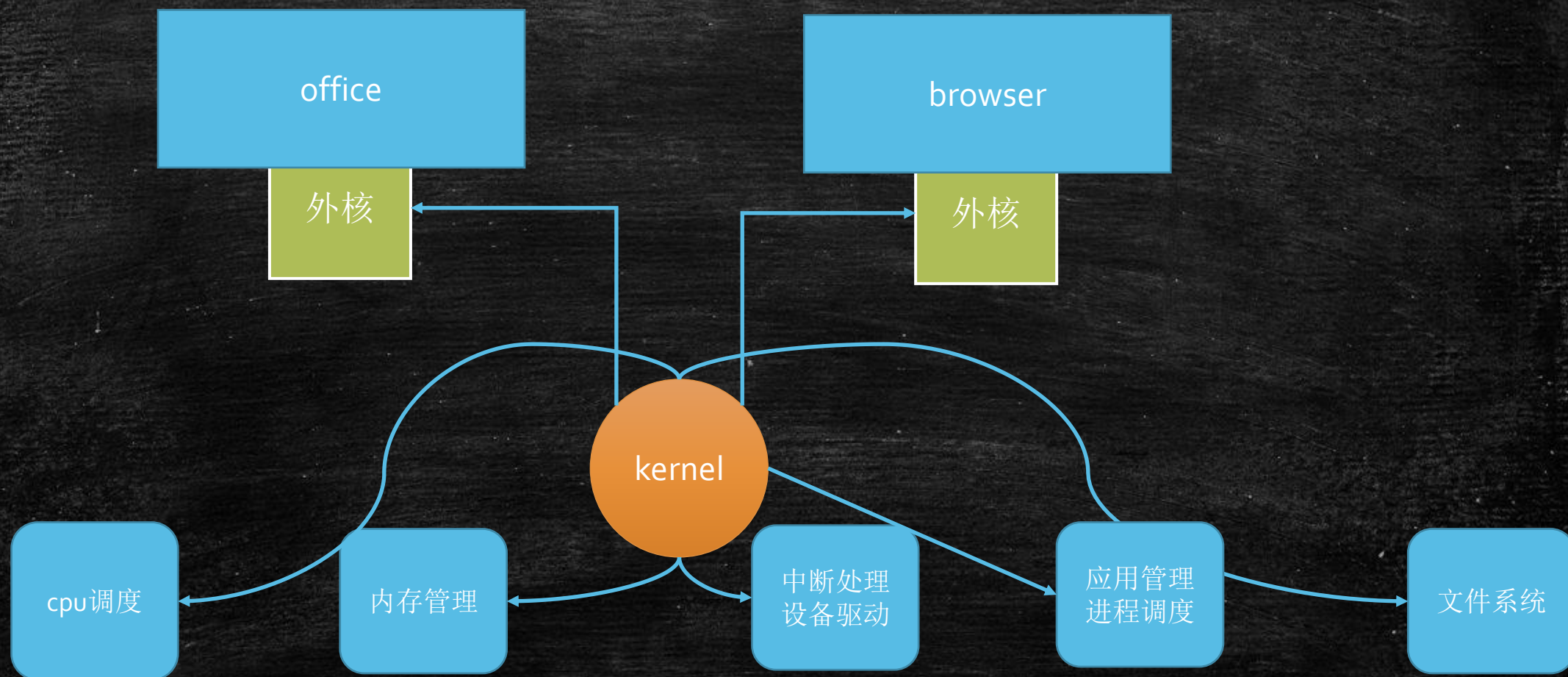
宏内核



微内核

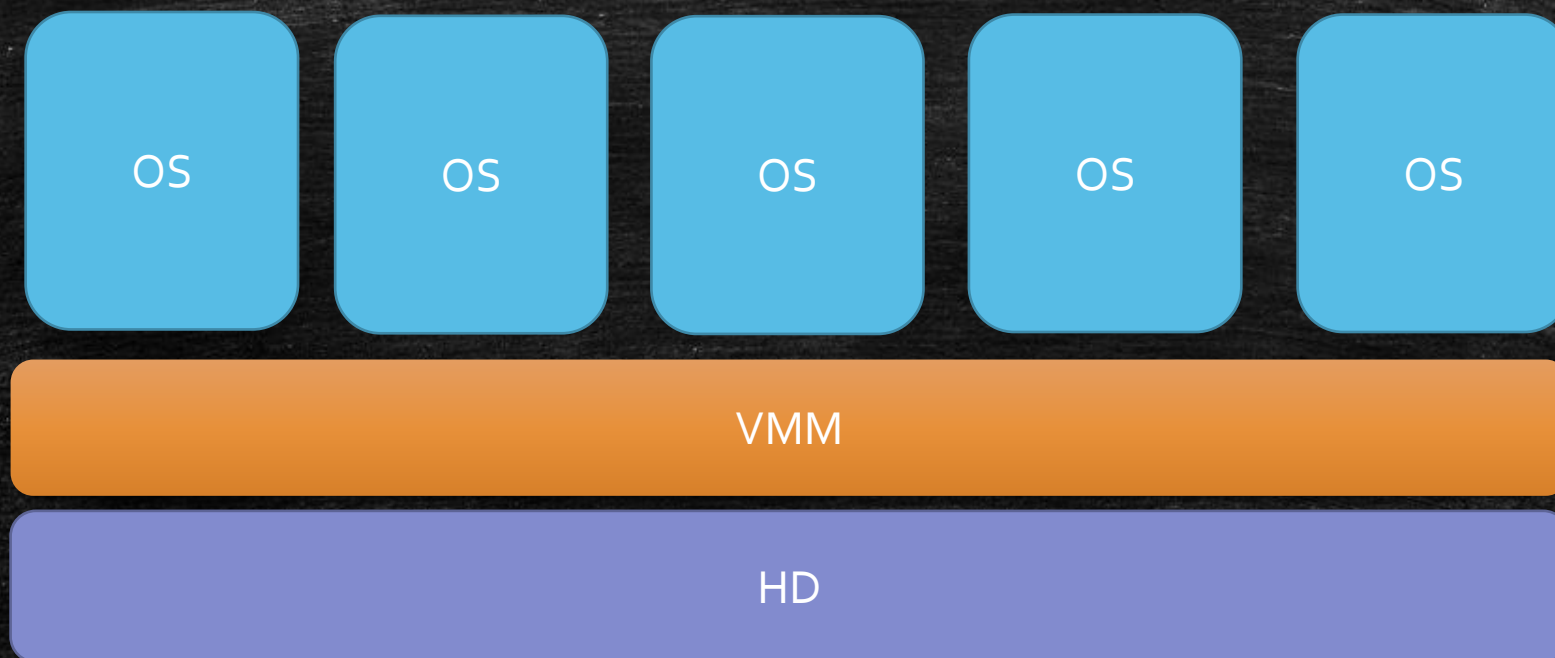


外核



VMM

硬件资源过剩

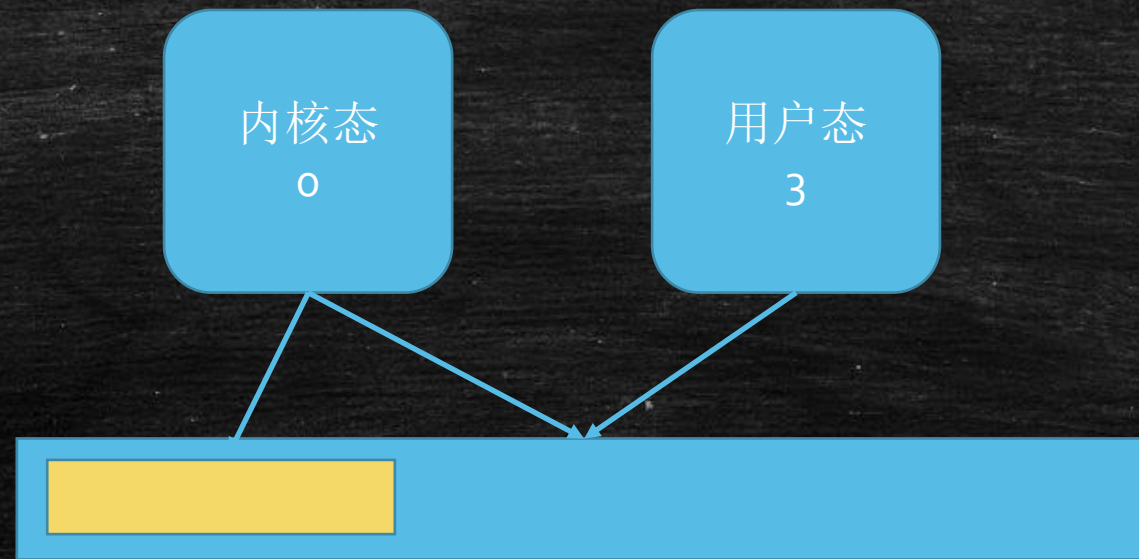


流行的操作系统



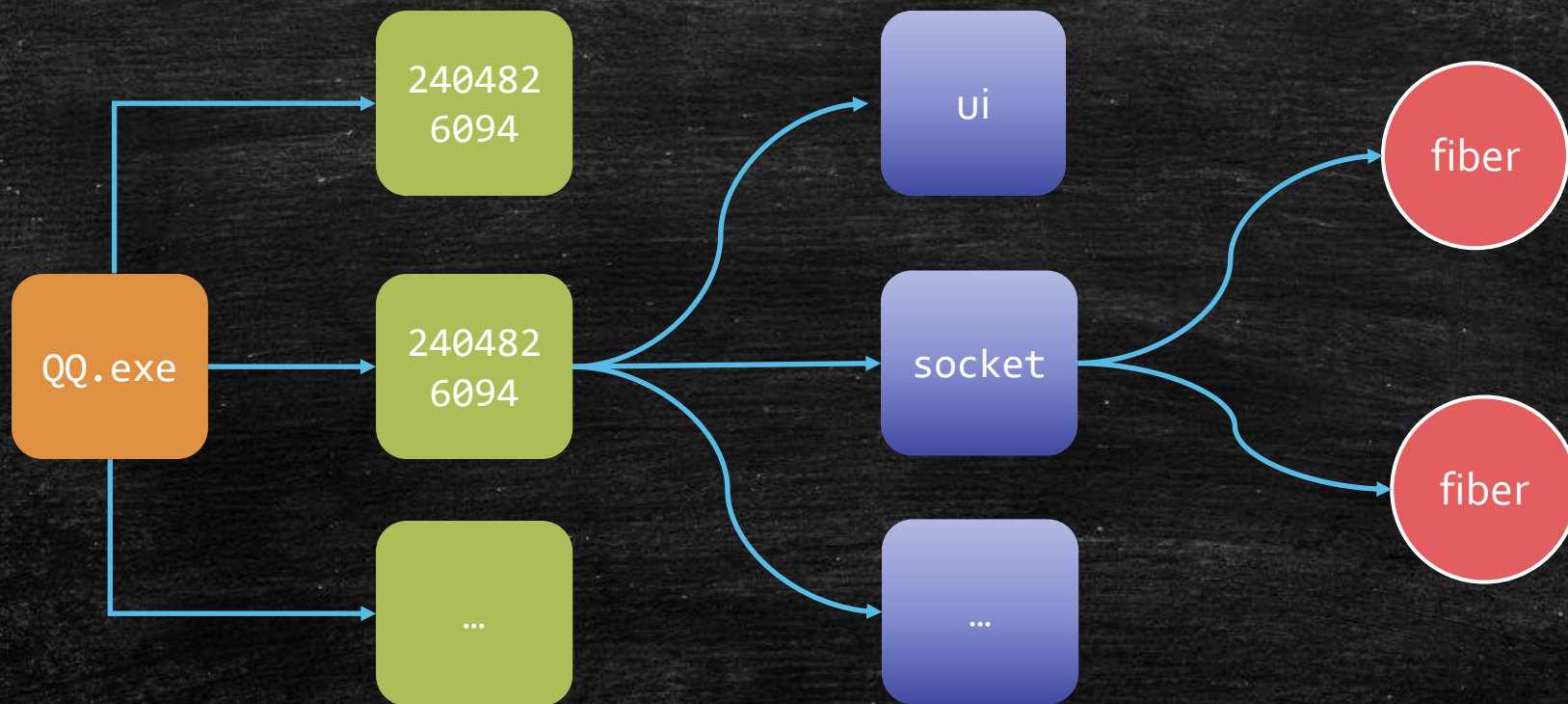
基础概念

CPU – CPL -> 0 1 2 3



进程 线程 纤程

程序 进程 线程 纤程



进程

linux中也称为task，是系统分配资源的基本单位

资源：独立的地址空间 内核数据结构（进程描述符...） 全局变量 数据段...

进程描述符：PCB（Process Control Block）



QQ 1

QQ 2

QQ 3

线程

线程在linux中的实现:

《linux内核设计与实现》第三版28页

就是一个普通进程，只不过和其他进程共享资源（内存空间 全局数据等..）

其他系统都有各自的所谓LWP的实现 Light Weight Process

高层面理解：一个进程中不同的执行路线

内核线程

内核启动之后经常需要做一些后台操作，这些由Kernel Thread来完成
只在内核空间运行

进程创建和启动

系统函数fork() exec()
从A中fork B的话, A称之为B的父进程



僵尸进程 孤儿进程

什么是僵尸线程

`ps -ef | grep defuct`

父进程产生子进程后，会维护子进程的一个PCB结构，子进程退出，由父进程释放
如果父进程没有释放，那么子进程成为一个僵尸进程

`zombie.c`

什么是孤儿线程

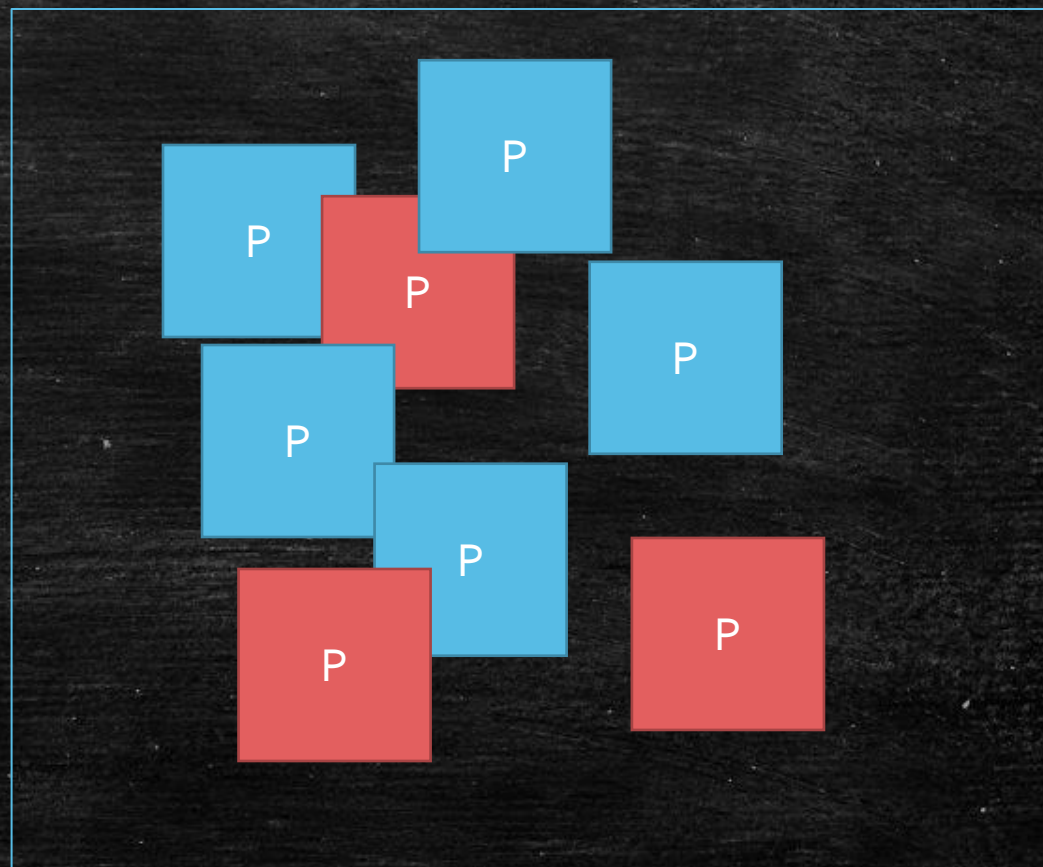
子进程结束之前，父进程已经退出

孤儿进程会成为init进程的孩子，由1号进程维护

`orphan.c`

进程（任务）调度

内核进程调度器决定：
该哪一个进程运行？
何时开始？
运行多长时间？



从单任务（独占）到多任务（分时）

DOS MacOS早期版本 Windows3.1 -> ...
timeslice

原则：最大限度的压榨CPU资源

多任务 (multitasking)

非抢占式 (cooperative multitasking)

除非进程主动让出cpu (yielding) , 否则将一直运行

抢占式 (preemptive multitasking)

由进程调度器强制开始或暂停 (抢占) 某一进程的执行

进程调度（选修）

linux2.5 经典Unix $O(1)$ 调度策略，偏向服务器，但对交互不友好

linux2.6.23 采用CFS 完全公平调度算法Completely Fair Scheduler

进程调度基本概念

- 进程类型：
 - IO密集型 大部分时间用于等待IO
 - CPU密集型 大部分时间用于闷头计算
- 进程优先级
 - 实时进程 > 普通进程 (0 - 99)
 - 普通进程nice值 (-20 - 19)
- 时间分配
 - linux采用按优先级的CPU时间比
 - 其他系统多采用按优先级的时间片
- eg. 两个app同时运行
 - 一个文本处理程序
 - 一个影视后期程序

linux默认的调度策略

对于实时进程：使用SCHED_FIFO 和 SCHED_RR两种
对于普通进程：使用CFS

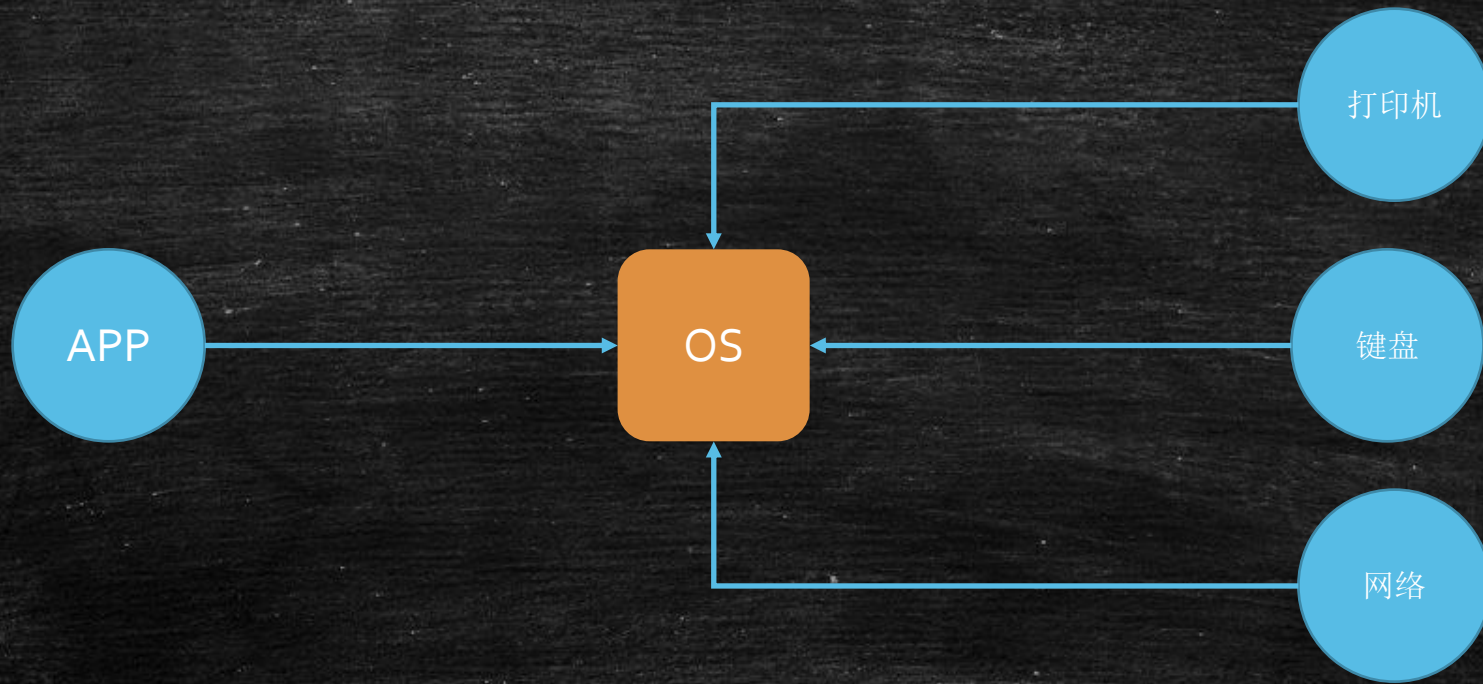
其中等级最高的是FIFO，这种进程除非自己让出CPU否则Linux会一直执行它
除非更高等级的FIFO和RR抢占它
RR只是这种线程中是同级别FIFO中的平均分配
只有实时进程主动让出，或者执行完毕后，普通进程才有机会运行

举例：

医院的医生和病人：急诊（实时） 和 门诊（普通）
病危（只要检验结果到了就得反馈） 和 疼痛（不断喊人，需要实时反馈）

中断

中断



系统调用: int 0x80 或者 sysenter原语
通过ax寄存器填入调用号

参数通过bx cx dx si di传入内核
返回值通过ax返回

java读网络 -> jvm read() -> c库read() ->
内核空间 -> system_call() (系统调用处理程序)
-> sys_read()

中断处理机制的实现细节



中断向量表

1 - 键盘 - 处理程序

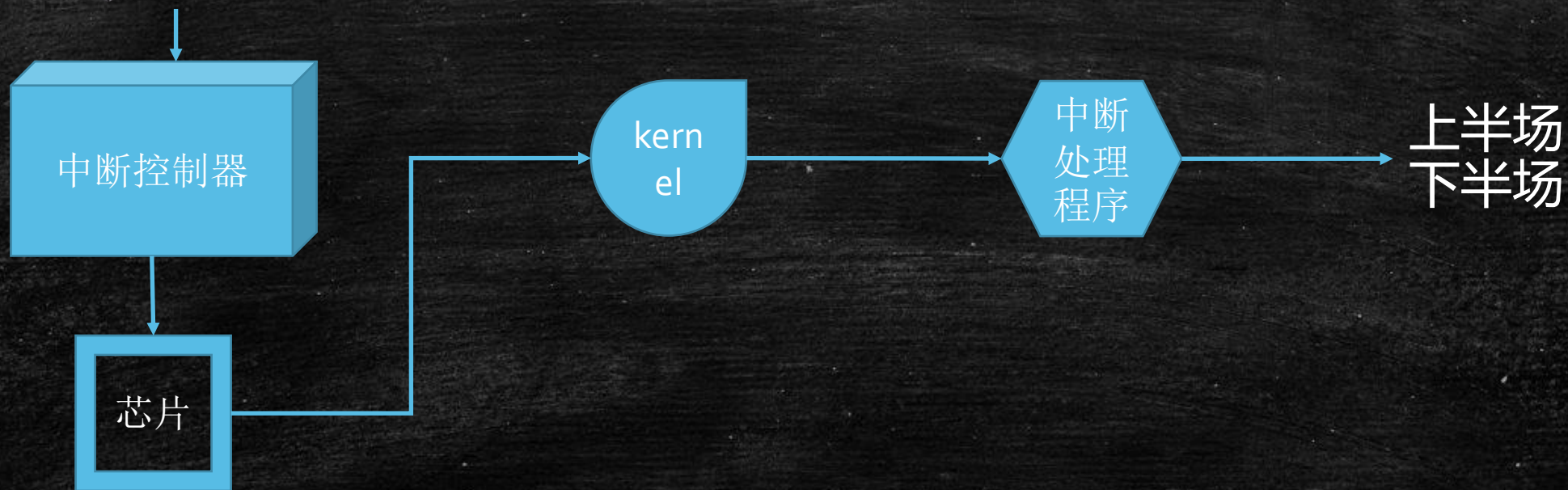
2 - 鼠标 - 处理程序

0x80 - 软中断 - 处理程序

eax <- 系统调用号

write <- 4号

exit <- 1号





1. app发出0x80中断 或 调用sysenter原语
2. os进入内核态
3. 在中断向量表中查找处理例程
4. 保存硬件现场 CS IP等寄存器值
5. 保存app现场 堆栈与寄存器的值
6. 执行中断例程system_call
 1. 根据参数与编号寻找对应例程
 2. 执行并返回
7. 回复现场
8. 返回用户态
9. app继续执行

用汇编语言展现0x80调用过程

```
; hello.asm
section .data ; 数据段声明
msg db "Hello, world!", 0xA ; 要输出的字符串 + 换行
len equ $ - msg ; 字符串长度
section .text ; 代码段声明
global _start ; 指定入口函数
_start: ; 在屏幕上显示一个字符串
mov edx, len ; 参数三: 字符串长度
mov ecx, msg ; 参数二: 要显示的字符串
mov ebx, 1 ; 参数一: 文件描述符(stdout)
mov eax, 4 ; 系统调用号(sys_write)
int 0x80 ; 调用内核功能
; 退出程序
mov ebx, 0 ; 参数一: 退出代码
mov eax, 1 ; 系统调用号(sys_exit)
int 0x80 ; 调用内核功能
```

编译:

```
nasm -f elf -g -F stabs hello.asm -o hello.o
```

链接:

```
ld -m elf_i386 -o hello hello.o
```

一个程序在某个时间点或处于用户态或处于内核态，通过系统调用，实现用户态到内核态的切换

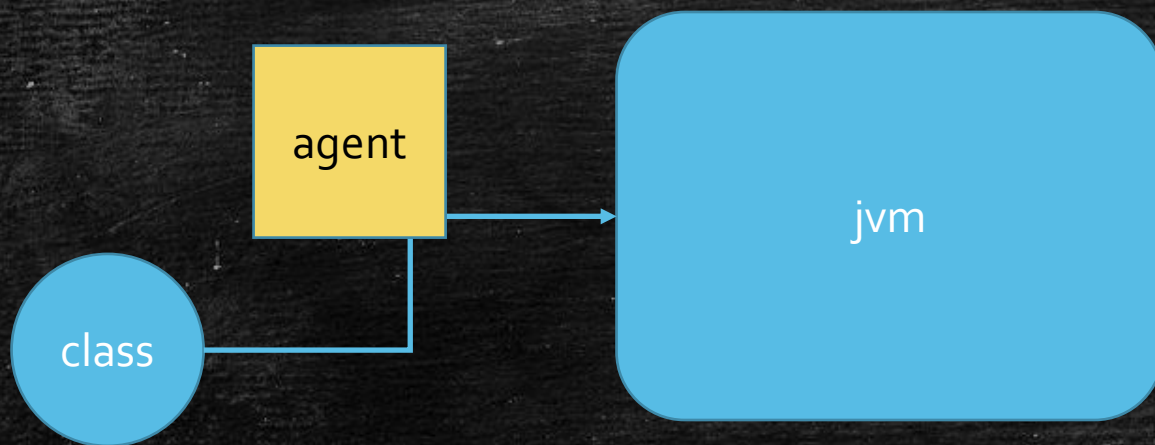
纤程与中断

腾讯面试

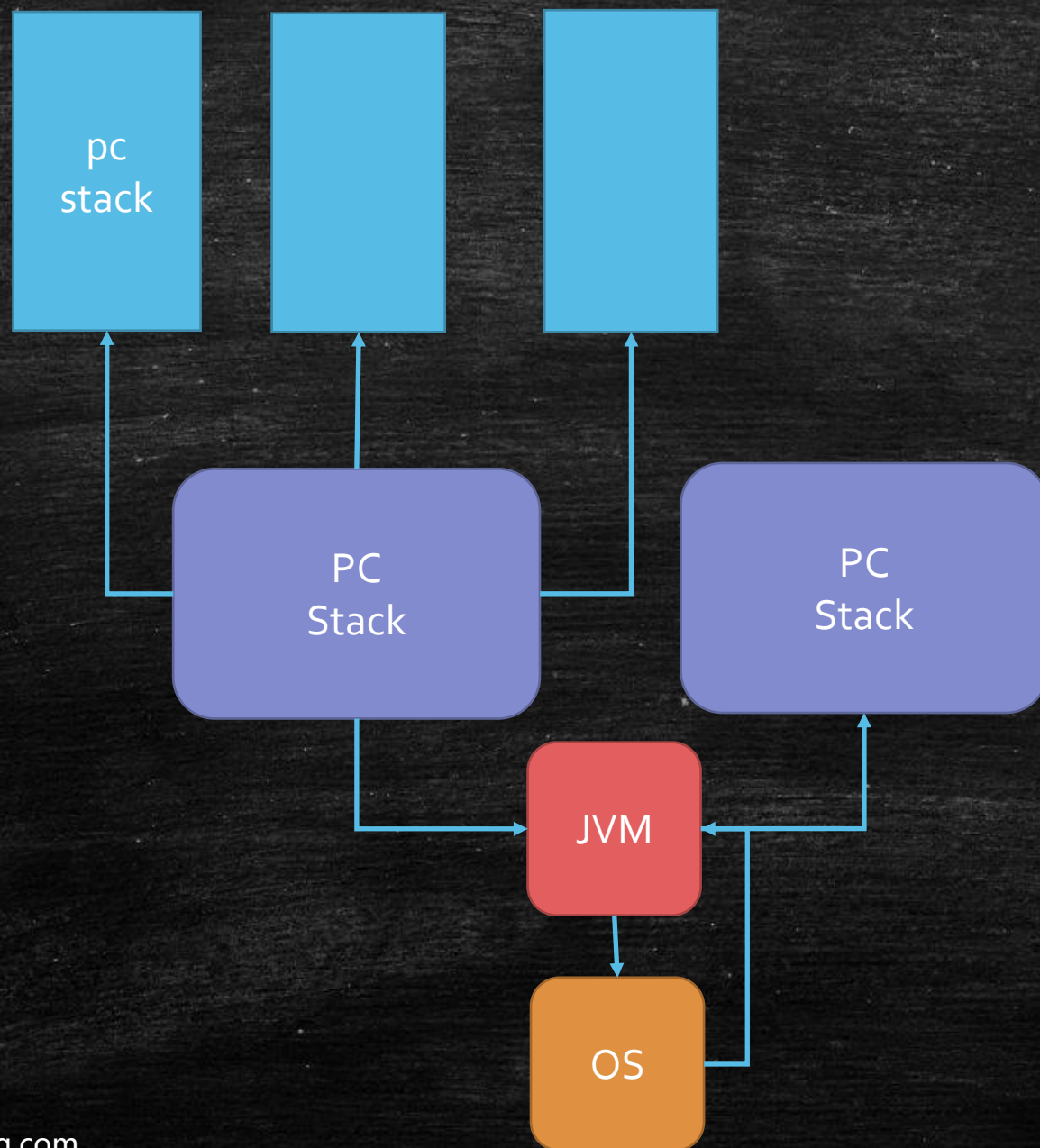
腾讯的面试：

1. 第一轮：JVM 算法
2. 第二轮：Redis + MySQL
3. 第三轮：疯狂追问式
 1. 线程
 2. 纤程 golang lua 协程
 3. Fiber vs Thread
 4. 为啥和内核态打交道效率低呢

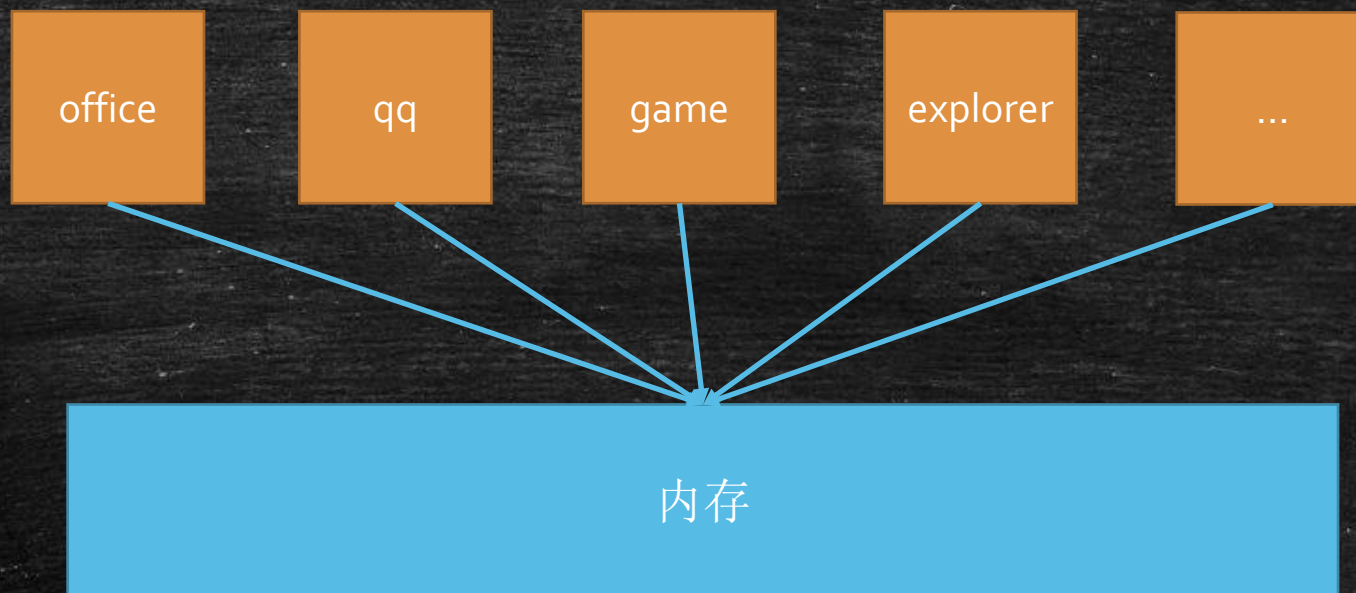
java instrumentation



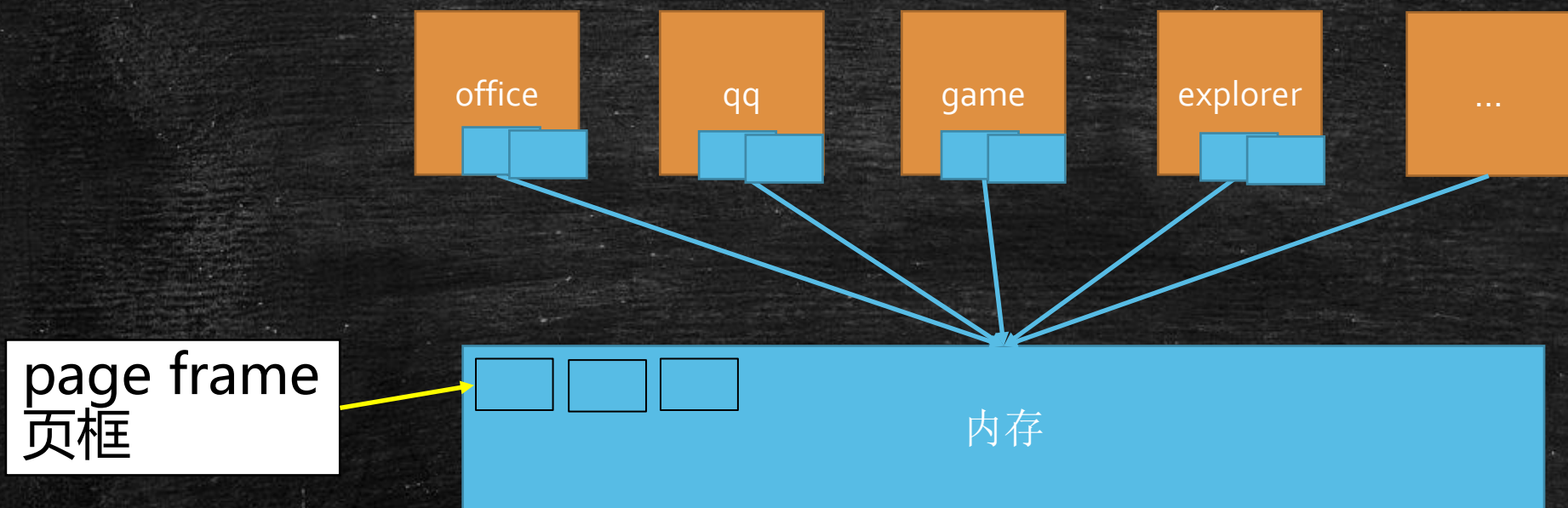
Fiber



内存管理



早期系统：多个进程全部装入内存
内存撑爆
互相干扰 不小心访问到别人的空间



解决内存撑爆问题：分块儿装入页框中（内存页 4K标准页）
（局部性原理：时间局部性-指令旁边的指令很快执行
空间局部性 - 数据旁边的数据很快用到）

内存满了，进行交换分区（LRU算法）

解决相互干扰问题：虚拟地址空间

虚拟内存

每一个进程都虚拟的独占
整个CPU

进程内部分段
段内部分页
需要该页的时候加载到页框

内核

用户代码不可见

用户栈

共享库

read printf

运行时堆

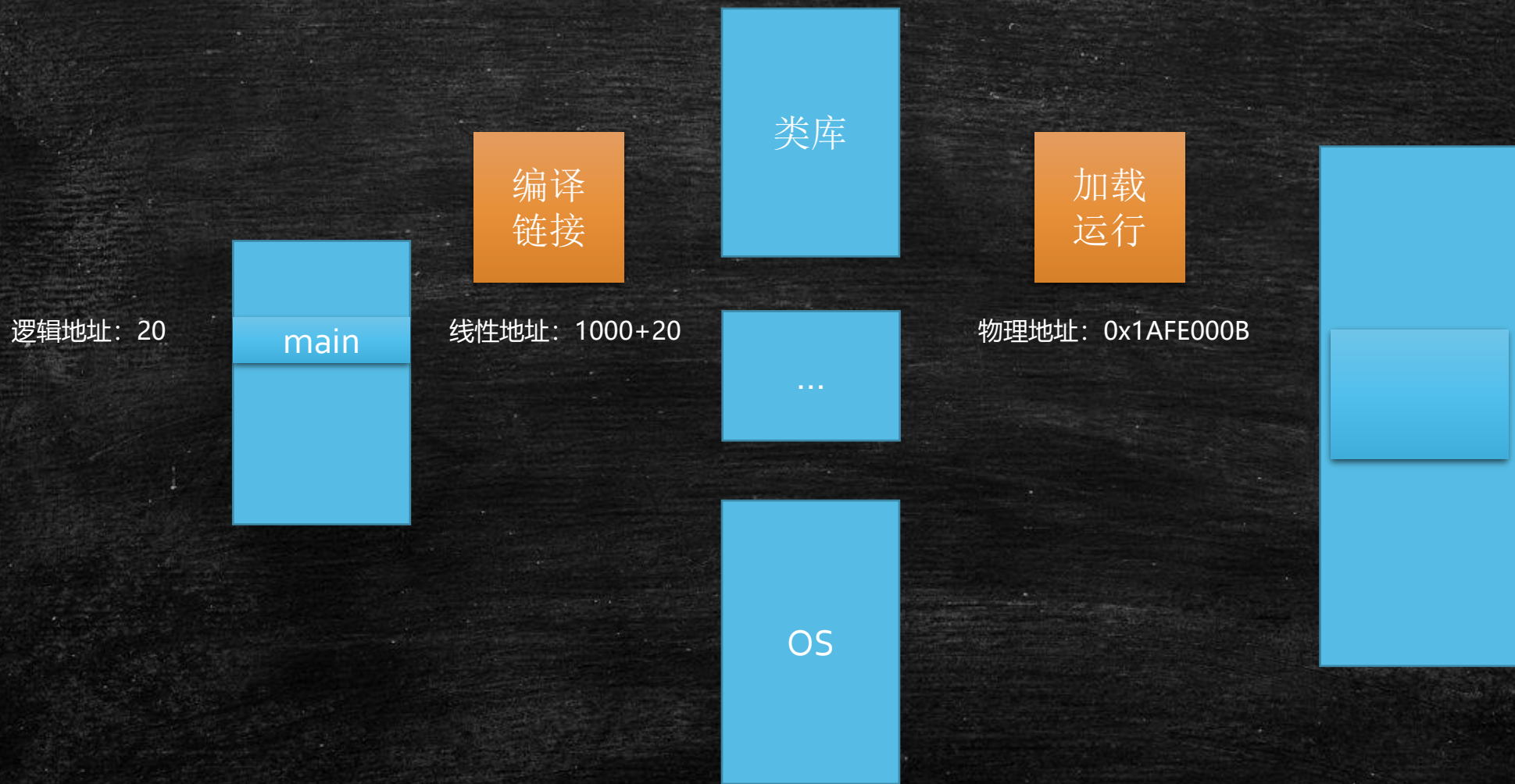
可读写的数据

只读的代码和数据

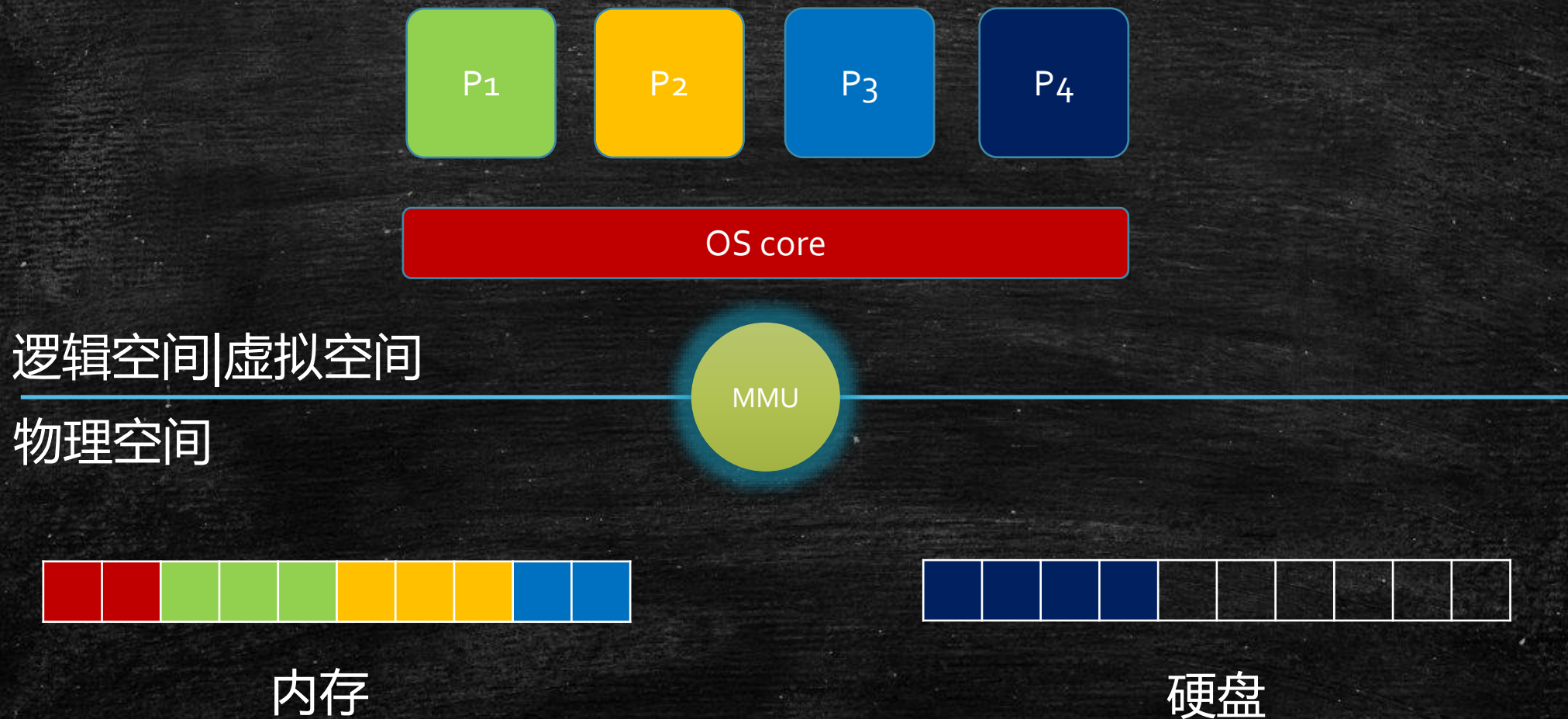
} 从hello加载

为什么使用虚拟内存？

- 隔离应用程序
 - 每个程序都认为自己有连续可用的内存
 - 突破物理内存限制
 - 应用程序不需要考虑物理内存是否够用，是否能够分配等底层问题
- 安全
 - 保护物理内存，不被恶意程序访问



内存地址映射



内核同步方法

介绍几个关于同步的理论上的概念 避免书中读到不知道什么意思

- 临界区 (critical area) : 访问或操作共享数据的代码段
简单理解: synchronized大括号中部分 (原子性)
- 竞争条件 (race conditions) 两个线程同时拥有临界区的执行权
- 数据不一致: data inconsistency 由竞争条件引起的数据破坏
- 同步 (synchronization) 避免race conditions
- 锁: 完成同步的手段 (门锁, 门后是临界区, 只允许一个线程存在)
上锁解锁必须具备原子性
- 原子性 (象原子一样不可分割的操作)
- 有序性 (禁止指令重排)
- 可见性 (一个线程内的修改, 另一个线程可见)

原子性 有序性 可见性 (Atomicity Ordering Visibility)

1. 原子操作 – 内核中类似于AtomicXXX, 位于<linux/types.h>
2. 自旋锁 – 内核中通过汇编支持的cas, 位于<asm/spinlock.h>
3. 读-写自旋 – 类似于ReadWriteLock, 可同时读, 只能一个写
读的时候是共享锁, 写的时候是排他锁
4. 信号量 – 类似于Semaphore(PV操作 down up操作 占有和释放)
重量级锁, 线程会进入wait, 适合长时间持有的锁情况
5. 读-写信号量 – downread upread downwrite upwrite
(多个写, 可以分段写, 比较少用) (分段锁)
6. 互斥体(mutex) – 特殊的信号量 (二值信号量)
7. 完成变量 – 特殊的信号量 (A发出信号给B, B等待在完成变量上)
vfork() 在子进程结束时通过完成变量叫醒父进程 类似于(Latch)
8. BKL: 大内核锁 (早期, 现在已经不用)
9. 顺序锁 (2.6) : – 线程可以挂起的读写自旋锁
序列计数器 (从0开始, 写时增加(+1), 写完释放(+1), 读前发现单数,
说明有写线程, 等待, 读前读后序列一样, 说明没有写线程打断)
10. 禁止抢占 – preempt_disable()
11. 内存屏障 – 见volatile

汇编

for javaer

段寄存器的由来：16位的寄存器想进行20位寻址
开机自启动程序