

# 2025年度操作系统复习

# 题型

- 选择题 2分\*15
  - 基本概念的直接考察，知识的综合运用，脑筋急转弯
- 计算题 10分\*3
  - 利用某个概念或者原理，进行计算、分析、设计等
- 系统分析题 20分\*2
  - 运用各种所学知识（包括理论和实验）进行综合设计、分析、优化
- 有10%-20%的卷面试题与实验相关

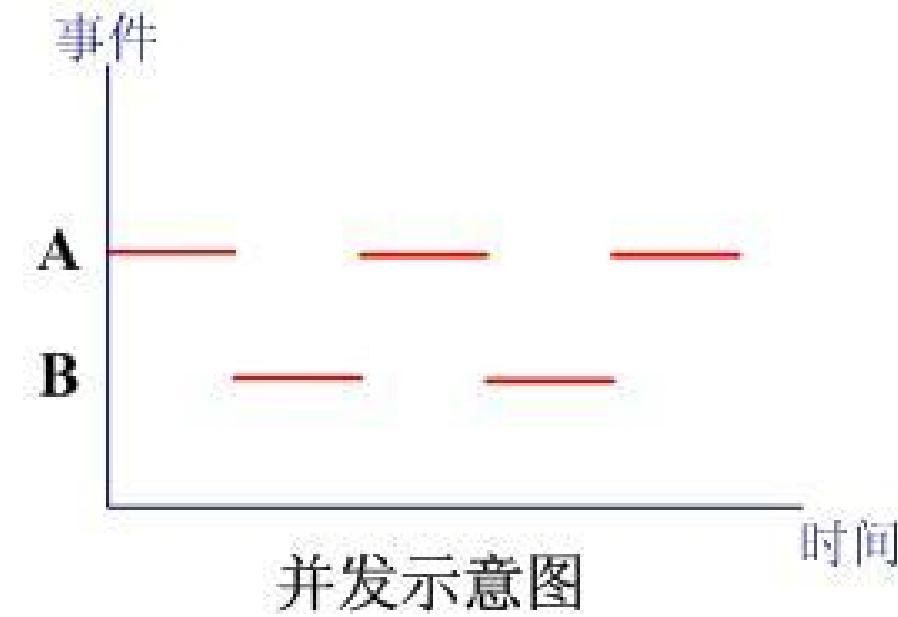
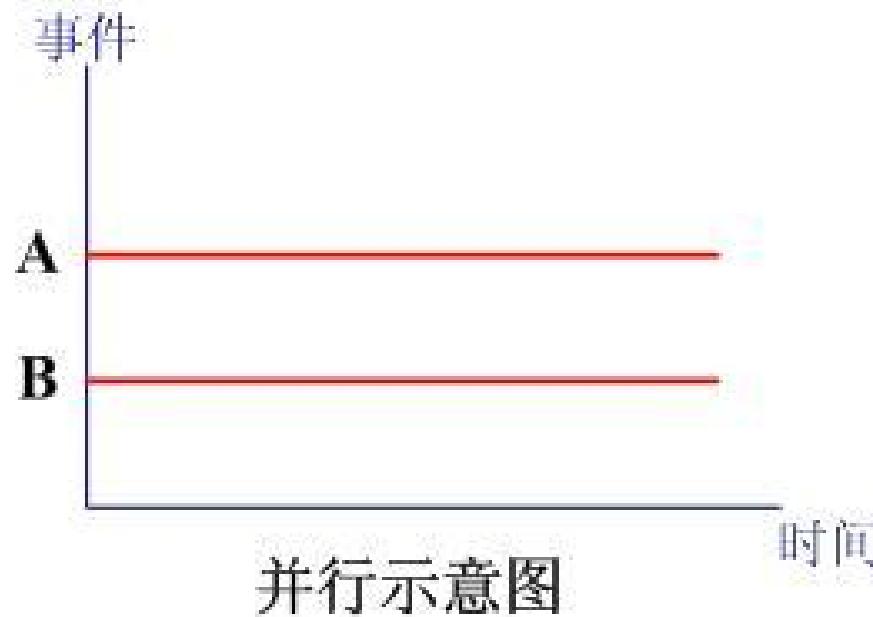
# 操作系统的试卷组成（不严谨不准确的统计）

- 进程管理 30%-45%
- 内存管理 30%-45%
- 文件系统 10%-15%
- 设备管理 10%-15%
- 死锁 5%-10%
- 安全

# 进程模型

- 进程的概念
- 并发与并行
- 进程引入带来的四个特性
  - 虚拟：每个进程都有自己虚拟的地址空间、虚拟的设备，以为自己掌控了这台计算机
  - 并发：多个进程是在并发交替的执行以提升效率
  - 共享：资源（设备、内存）在进程之间共享，分时复用
  - 不确定：进程无法预知调度在什么时刻发生

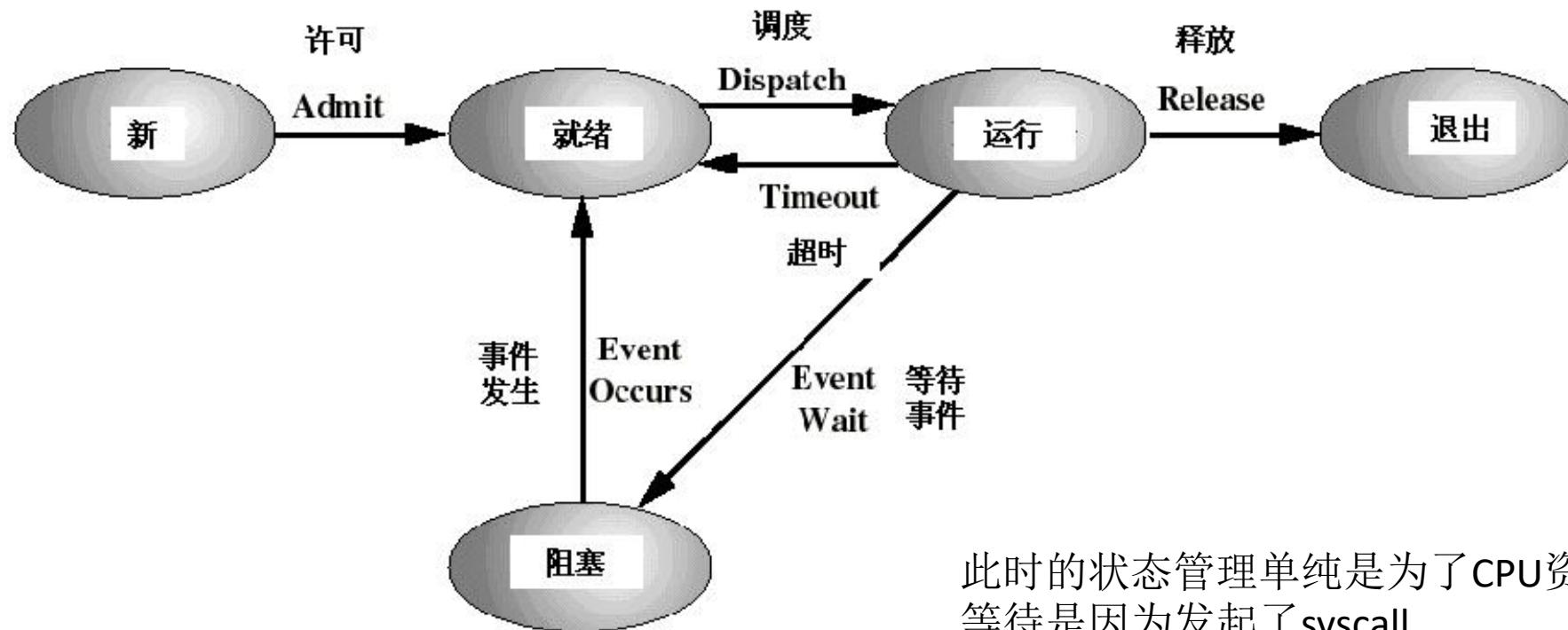
# 并发还是并行？这是个问题



# 进程实现

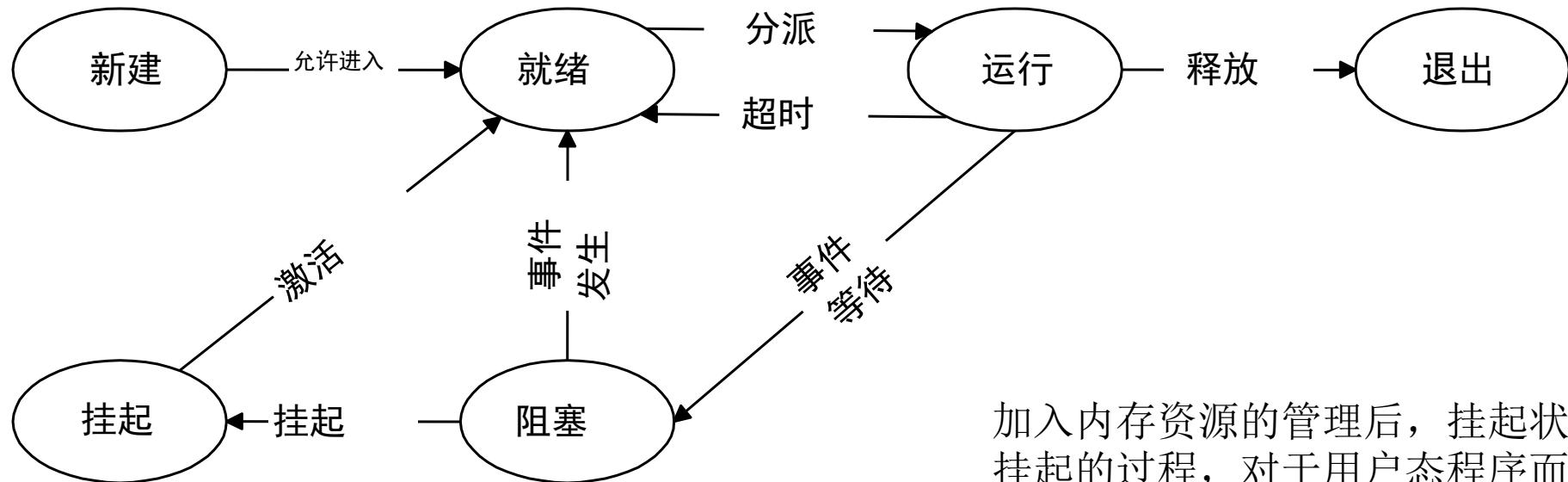
- 进程的状态迁移模型-----解决谁停下来的问题
- 调度算法----就绪队列里选择一个进程运行， 解决谁去运行的问题
- 进程的实现方法-----解决怎么停下来、怎么运行的问题
- 线程-----与内存管理理解耦

# Complex Process States (1)



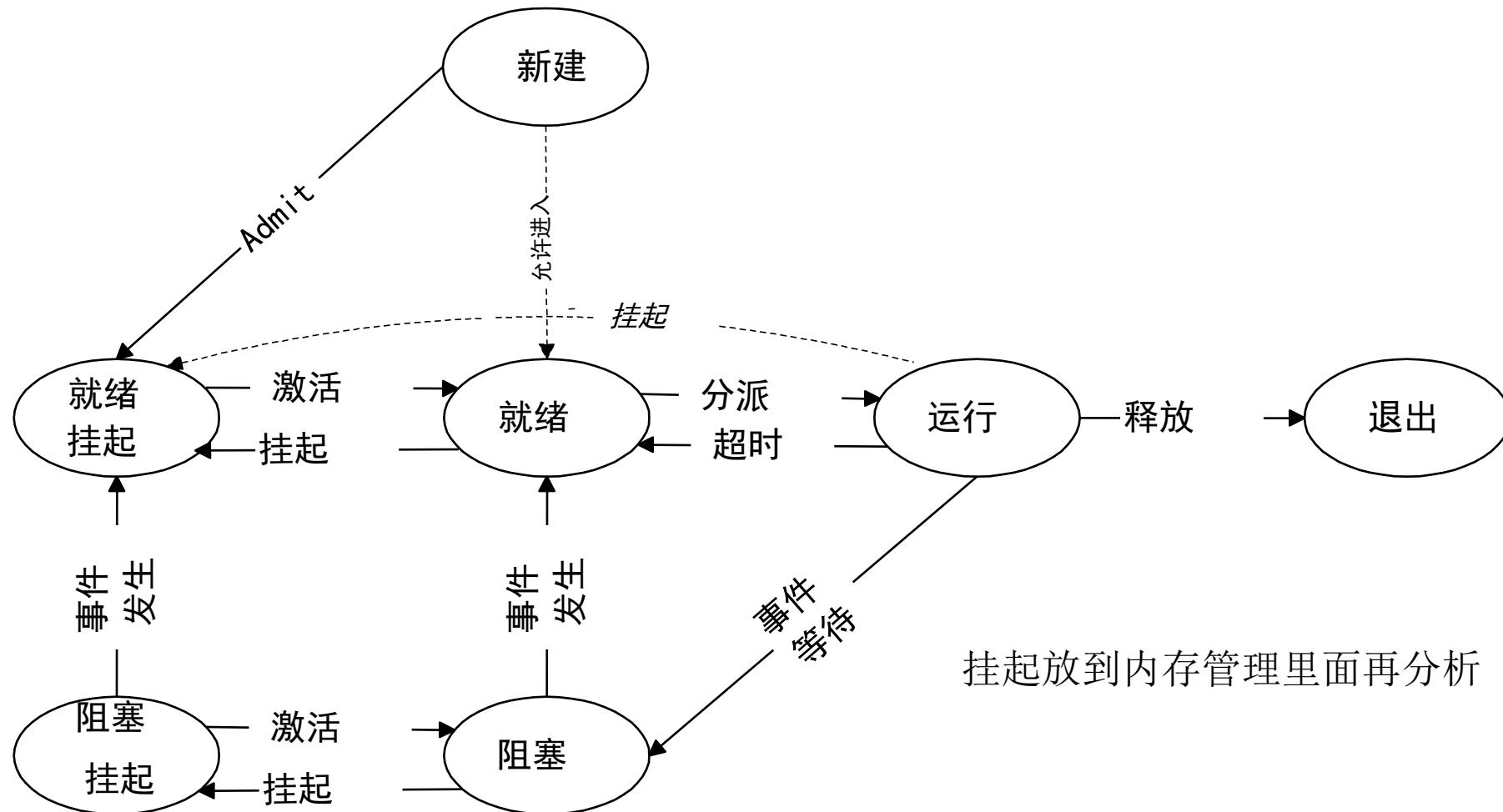
此时的状态管理单纯是为了CPU资源的分配  
等待是因为发起了syscall  
中断导致从阻塞回到了就绪  
就绪得到运行，会从syscall函数返回

# 单挂起进程模型



加入内存资源的管理后，挂起状态出现了挂起的过程，对于用户态程序而言不可见

# 双挂起进程模型



# 进程调度算法

- 先来先服务
- 短任务优先， 剩余时间短任务优先
- 时间片轮转
- 优先级队列
- 调度算法的设计原则：最优解， 可以获取的参数， 获取和推理的代价
- 调度的驱动力：进程主动调度（进程主动让出CPU， 或者进程发起`syscall`而OS认为有必要调度）， 或者中断抢占调度

# 比较调度算法的几个**非要重要的**指标体系

- CPU使用率
  - ▶ CPU处于忙状态的时间百分比
- 吞吐量
  - ▶ 单位时间内完成的进程数量
- 周转时间
  - ▶ 进程从初始化到结束(包括等待)的总时间
- 等待时间
  - ▶ 进程在就绪队列中的总时间
- 响应时间
  - ▶ 从提交请求到产生响应所花费的总时间

# 进程切换

## ■ 进程切换(上下文切换)

- ▶ 暂停当前运行进程，从运行状态变成其他状态
- ▶ 调度另一个进程从就绪状态变成运行状态

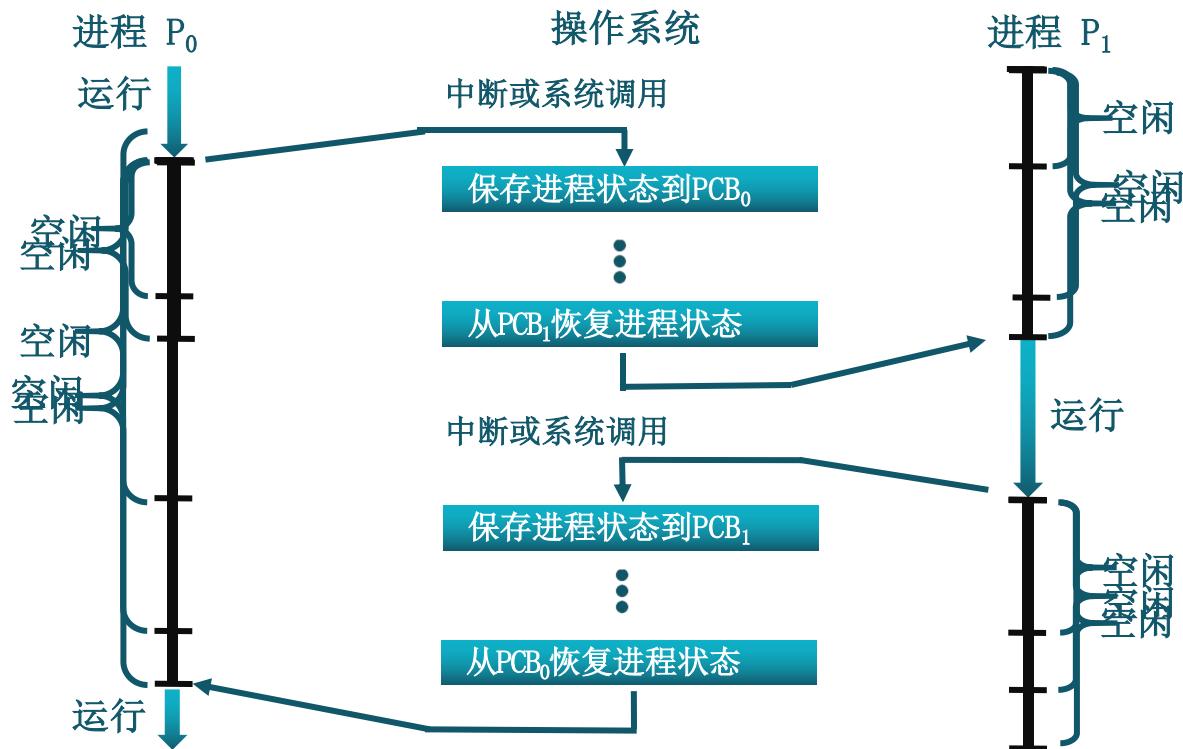
## ■ 进程切换的要求

- ▶ 切换前，保存进程上下文
- ▶ 切换后，恢复进程上下文
- ▶ 快速切换

## ■ 进程生命周期的信息

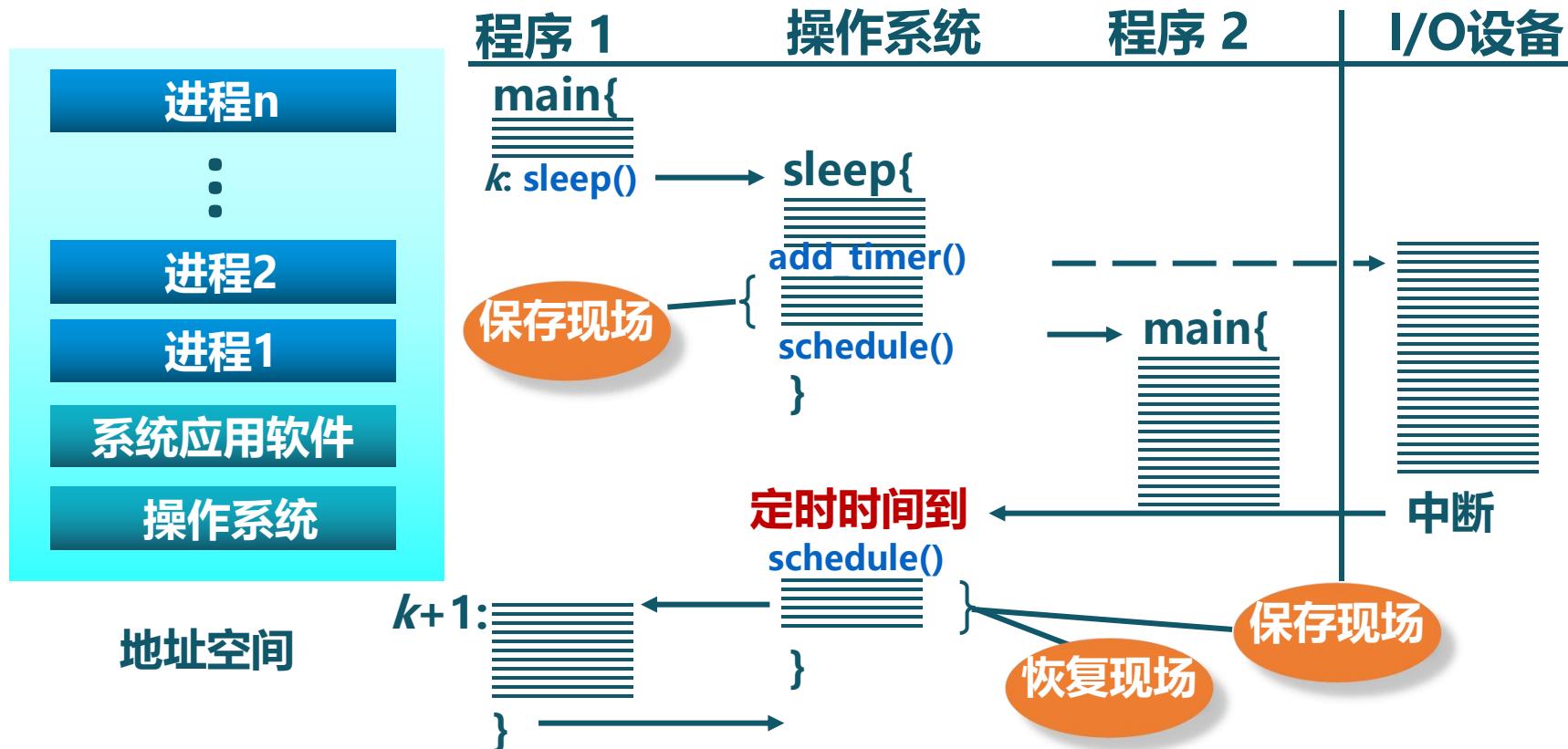
- ▶ 寄存器 (PC, SP, ...)
- ▶ CPU状态
- ▶ 内存地址空间

# 上下文切换图示



同步的还有地址空间的切换

## 进程切换



# switch to的实现

kern-ucore/arch/i386/process/switch.S

```
.text
.globl switch_to
switch_to:          # switch_to(from, to)

# save from's registers
    movl 4(%esp), %eax      # eax points to from
    popl 0(%eax)           # save eip !popl
    movl %esp, 4(%eax)
    movl %ebx, 8(%eax)
    movl %ecx, 12(%eax)
    movl %edx, 16(%eax)
    movl %esi, 20(%eax)
    movl %edi, 24(%eax)
    movl %ebp, 28(%eax)

# restore to's registers
    movl 4(%esp), %eax
    # not 8(%esp): popped return address already
    # eax now points to to

    movl 28(%eax), %ebp
    movl 24(%eax), %edi
    movl 20(%eax), %esi
    movl 16(%eax), %edx
    movl 12(%eax), %ecx
    movl 8(%eax), %ebx
    movl 4(%eax), %esp

    pushl 0(%eax)          # push eip
    ret
```

进程上下文的衍生问题：  
地址空间的切换是如何实现的？  
进程切换的代价有多大？  
线程呢？

文件(F) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)

```
    return last_pid;
}
```

```
// proc_run - make process "proc" running on cpu
```

```
// NOTE: before call switch_to, should load base addr of "proc"'s new PDT
```

```
void
```

```
proc_run(struct proc_struct *proc) {
```

```
    if (proc != current) {
```

```
        bool intr_flag;
```

```
        struct proc_struct *prev = current, *next = proc;
```

```
        local_intr_save(intr_flag);
```

```
{
```

```
    current = proc;
```

```
    load_esp0(next->kstack + KSTACKSIZE);
```

```
    lcr3(next->cr3);
```

```
    switch_to(&(prev->context), &(next->context));
```

```
}
```

```
    local_intr_restore(intr_flag);
```

```
}
```

```
// forkret -- the first kernel entry point of a new thread/process
```

```
// NOTE: the addr of forkret is setted in copy_thread function
```

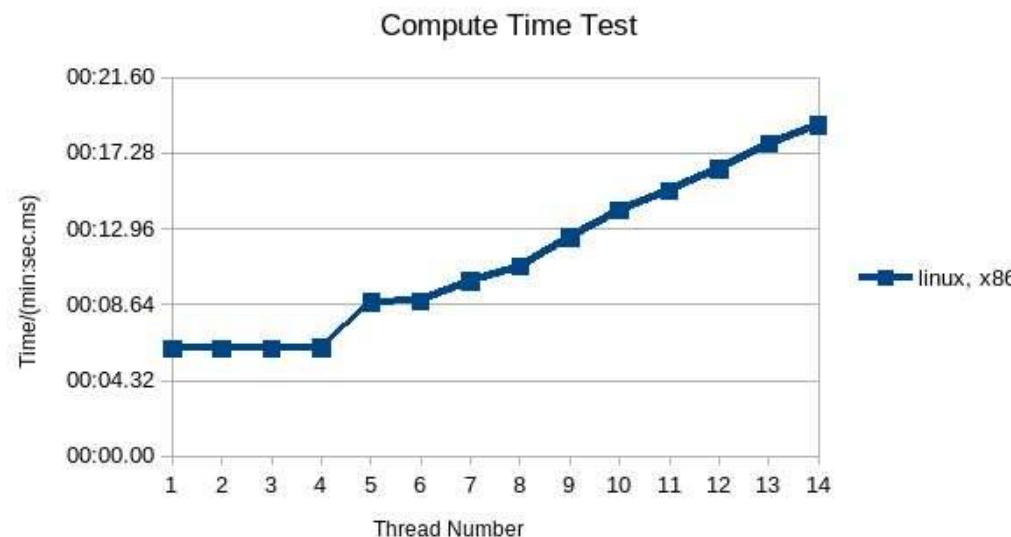
"kern/process/proc.c" 1115 行 --19--

1. 在哪里换的虚拟地址空间?  
cr3
2. 换了空间为啥不影响  
switch\_to的执行? 地址空  
间重叠
3. TLB中存的是什么? 页表  
的部分信息
4. 为什么换页表不用管TLB?  
写入cr3时自动失效
5. 这样有什么缺陷? 重复的  
页表项也没了。
6. 有cache吗? Cache怎么办?  
丢弃, 或者用实地址  
cache
7. 为什么换了地址空间不需要  
重新定位变量? 因为多个  
进程之间内核空间是共  
享和统一的

# 调度的代价

- i7四核处理
- 四个完全独立的程序(用泰勒级数计算pai)
- 最后一个线程的完成时截止
- 单线程执行约5分钟
- 四线程与单线程一致
- 5线程时，需要8分40秒（理论值）

实验表明：华为鲲鹏916服务器的线程调度时间为1900ns，可供1000余条机器指令执行



# 进程间通信

- 信号
- 管道
- 消息队列
- 共享内存

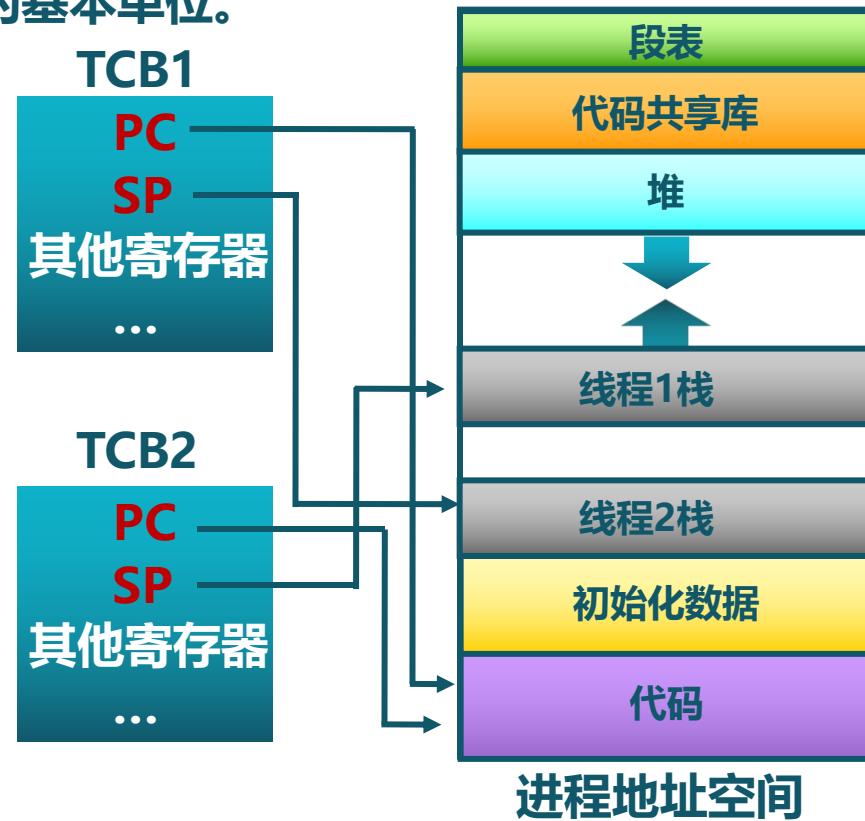
# 线程

- 几种线程的实现方案对比
  - 用户空间线程
  - 内核线程
  - 组合式

# 线程的概念

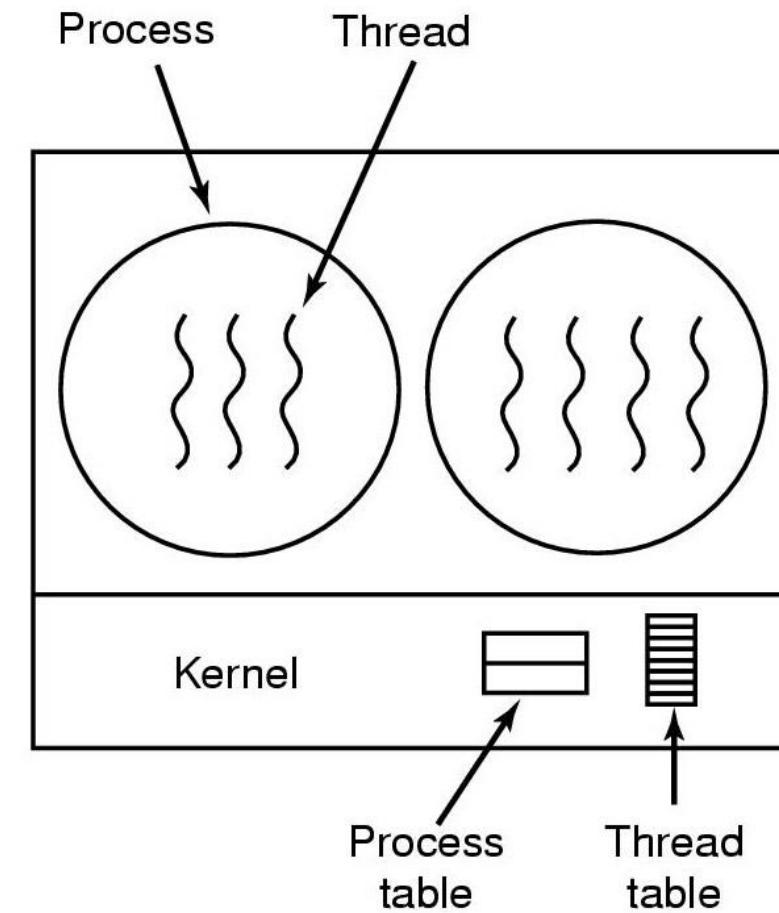
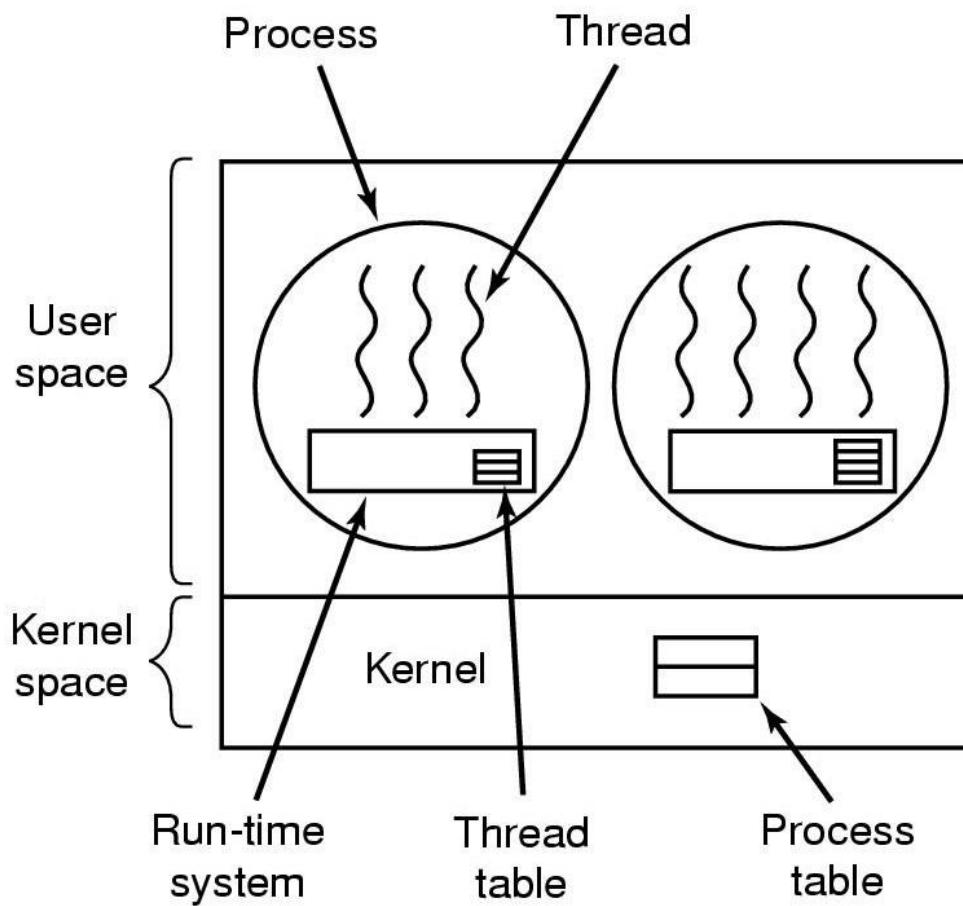
线程是进程的一部分，描述指令流执行状态。它是进程中的指令执行流的最小单元，是CPU调度的基本单位。

- ▶ 进程的资源分配角色：  
进程由一组相关资源构成，包括地址空间（代码段、数据段）、打开的文件等各种资源
- ▶ 线程的处理机调度角色：  
线程描述在进程资源环境中的指令流执行状态



# 线程应该在用户态还是内核态？

- 用户态线程 vs 内核态线程



## ULT管理优势:

- 线程切换不需要内核模式特权.
- 线程调用可以是应用程序级的,根据需要可改变调度算法,但不会影响底层的操作系统调度程序.
- ULT管理模式可以在任何操作系统中运行,不需要修改系统内核,线程库是提供应用的实用程序。

## ULT的劣势:

- 系统调用会引起进程阻塞
- 这种线程不利于使用多处理器并行

核心级线程（KLT）：线程由OS内核进行管理，内核给应用程序级提供系统调用，实现对线程的使用。

### 特点:

- 线程在内核中有保存的信息,系统调度是基于线程完成的.
- 可克服ULT的两个缺点,且内核程序本身也可以是多线程结构的.

### 劣势:

- 线程间的控制转换需要转换到内核模式.（为什么）

# 进程管理中的新概念

- 纤程 Fiber, ucontext
- 协程 coroutine
- 发挥ULT快速切换的优势
- 避免无序的争抢，切换都是程序员有意识的发生的
- 在编程时提出对程序员的限制，要求他们妥善的设计代码
- ucontext是如何实现的？这仍然是个新兴的研究课题
- 面向对象的运行时包装原来的user space thread
- 使用体系结构支持用户空间的context switch等等

论文数据展示：华为鲲鹏916服务器的线程调度时间从2500ns左右被用户级context缩减致900ns左右，性能提升近60%

# 并发与同步（进程间通信vs线程间并发）

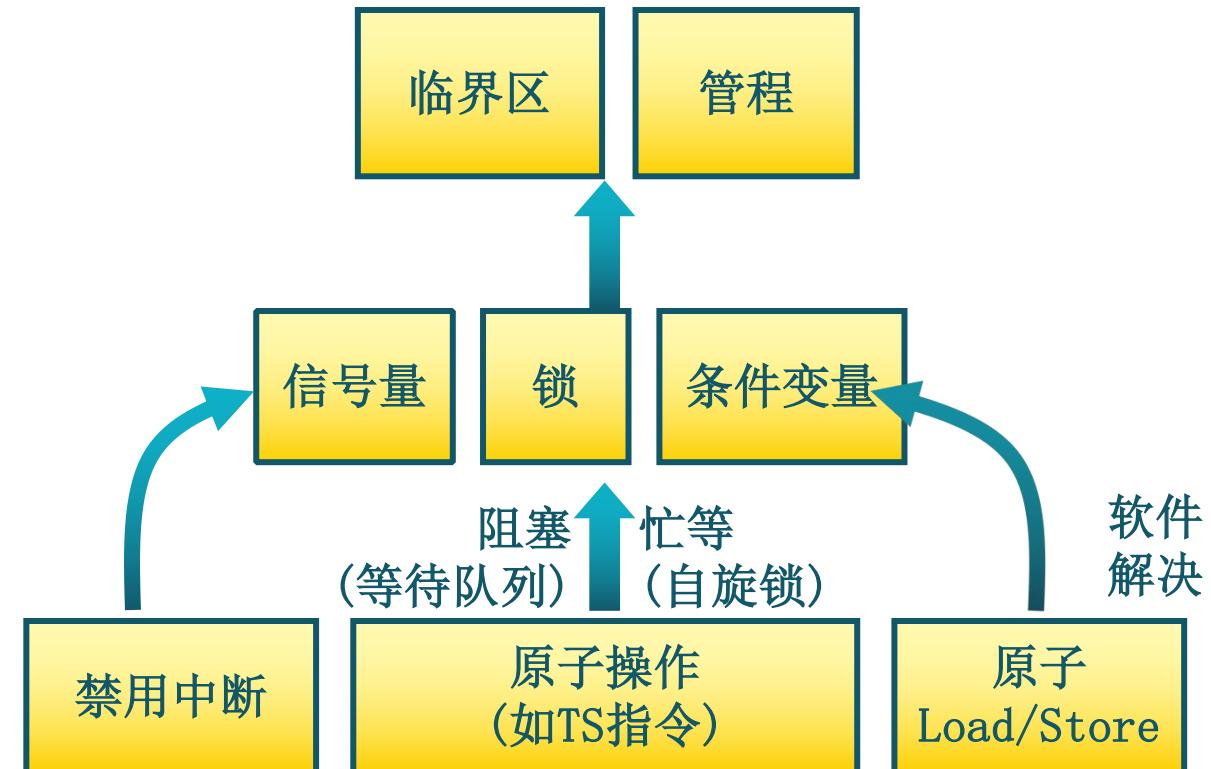
- 竞争、临界区
  - 几种常见的错误
  - 几种基于高级语言的“取巧”的设计
- 互斥、同步---空闲则入，忙则等待，有限等待
- 关中断---理论上能够解决单核心的问题，但是不安全因此不能用（为什么）
- 原语、信号量、**管程**
  - 借助体系结构（TSL指令）
  - 借助操作系统的封装，使关中断和配套的操作暴露给用户（系统调用得到的信号量）
  - **借助编程语言再次升级易用性**
- 经典问题
  - 生产者消费者
  - 沉睡的理发师
  - 饥饿的哲学家
  - 读者写者
- **具有方案设计或既有方案bug分析的能力**
  - 记住设计的套路

# 基本同步方法

并发编程

高层抽象

硬件支持



# 用信号量实现临界区的另一种伪码形式

每个临界区设置一个信号量，其初值为1

```
mutex = new Semaphore(1);  
  
mutex->P();  
Critical Section;  
mutex->V();
```

- 必须成对使用P()操作和V()操作
  - ▶ P()操作保证互斥访问临界资源
  - ▶ V()操作在使用后释放临界资源
  - ▶ PV操作不能次序错误、重复或遗漏

# 用信号量实现条件同步

每个条件同步设置一个信号量，其初值为0

```
condition = new Semaphore(0);
```

线程A

```
... M ...  
condition->P();  
... N ...
```

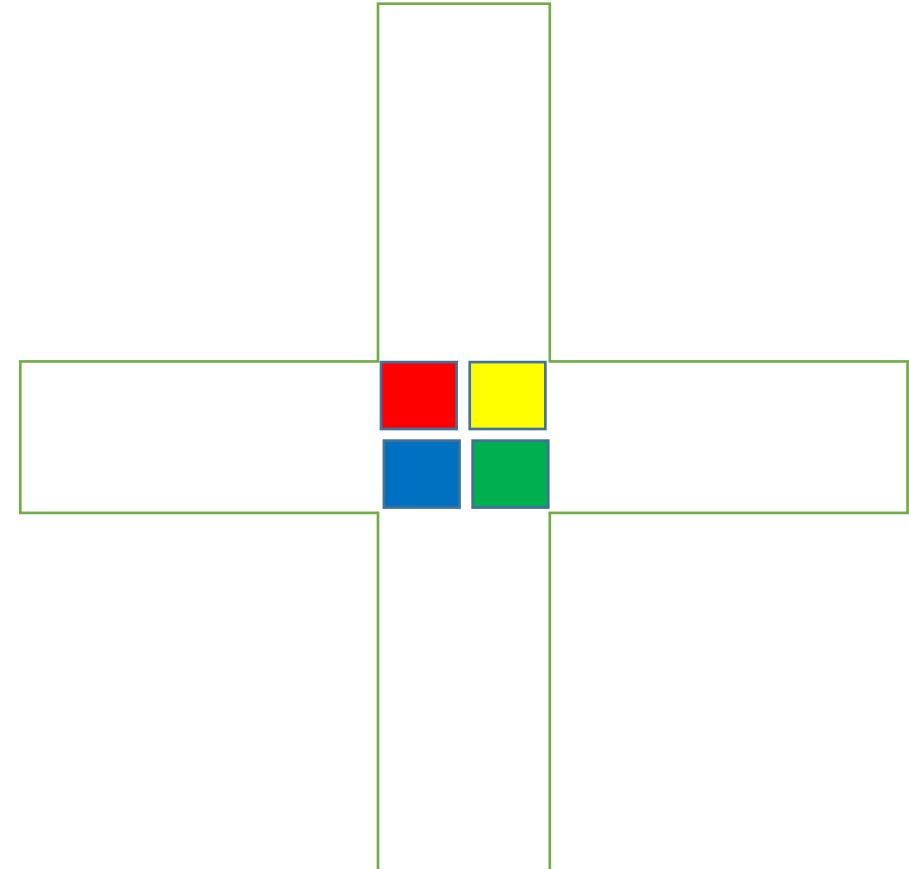
线程B

```
... X ...  
condition->V();  
... Y ...
```



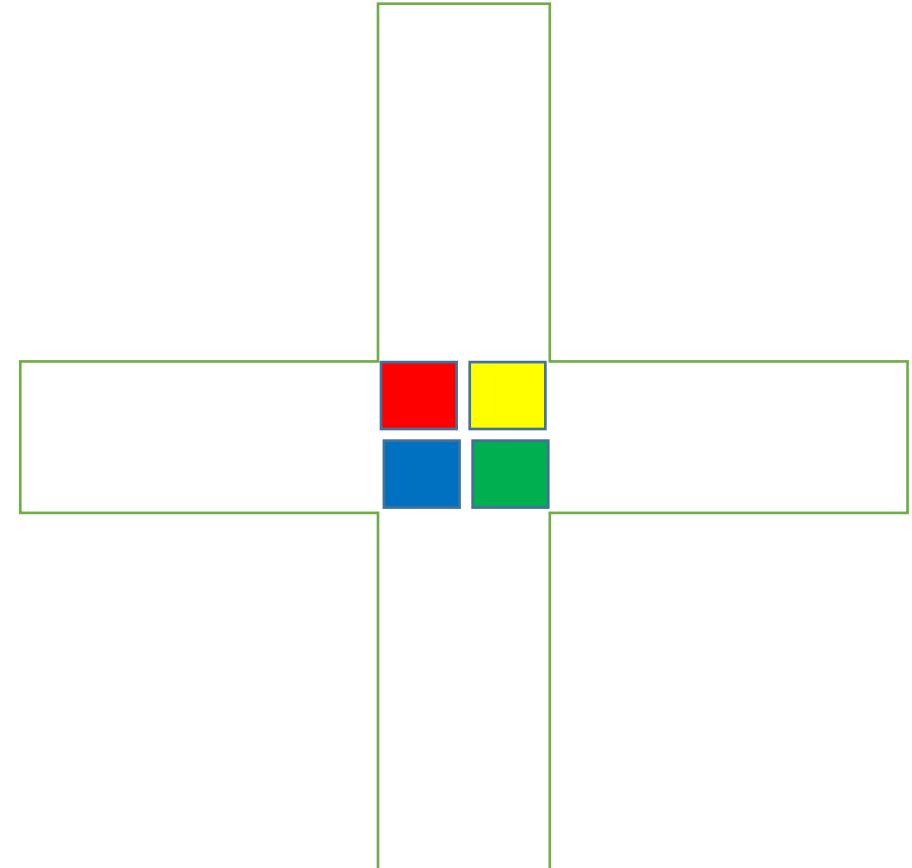
# 学会使用信号量设计问题的解决方案

- 如何设计信号量，可以使得车辆在通行时不会发生事故？
- 如何设计，可以使得不出现一方堵塞时间过长的情况？
- 如何设计可以避免死锁？



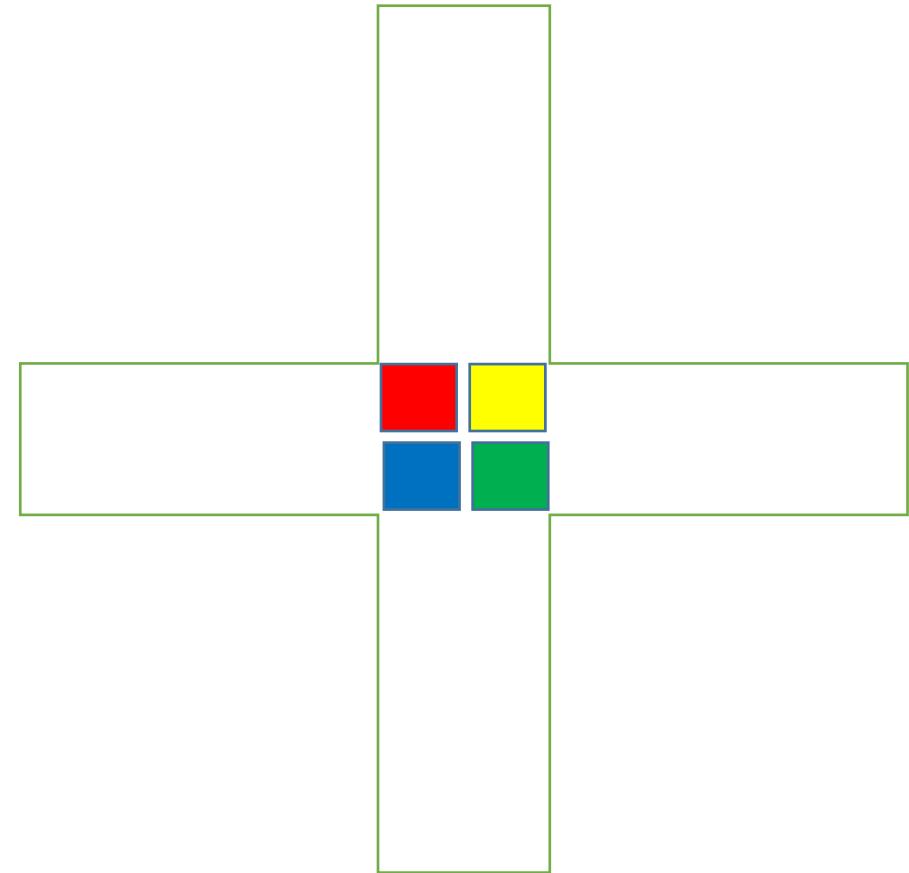
# 解析

- 申请四个信号量 $R, Y, B, G$ 分别对应四块路面
- 以从左到右通行的车辆为例
  - $P(B), P(G)$ 成功后，就可以通行了，通行离开后，释放资源即可
  - 问题？



# 解析

- 申请四个信号量 $R, Y, B, G$ 分别对应四块路面
- 以从左到右通行的车辆为例
  - $P(B), P(G)$ 成功后，就可以通行了，通行离开后，释放资源即可
- 通过观察不难发现，最极端的情况，也只能有3辆车同时通过，所以可以再设置一个进入路口用的信号量，不要让4辆车同时到达路口
- 如何设计，可以使得不出现一方堵塞时间过长的情况？
- 死锁？？



# 另一个解题思路

- 参考哲学家就餐问题，这两块地面资源，其实就是两支筷子，需要同时拿到，才可以通行
- 因此，为车辆加入状态，如果左向或右向车正在通行，则上向或下向车等待，反之亦然
- 请参考哲学家就餐，自行实现

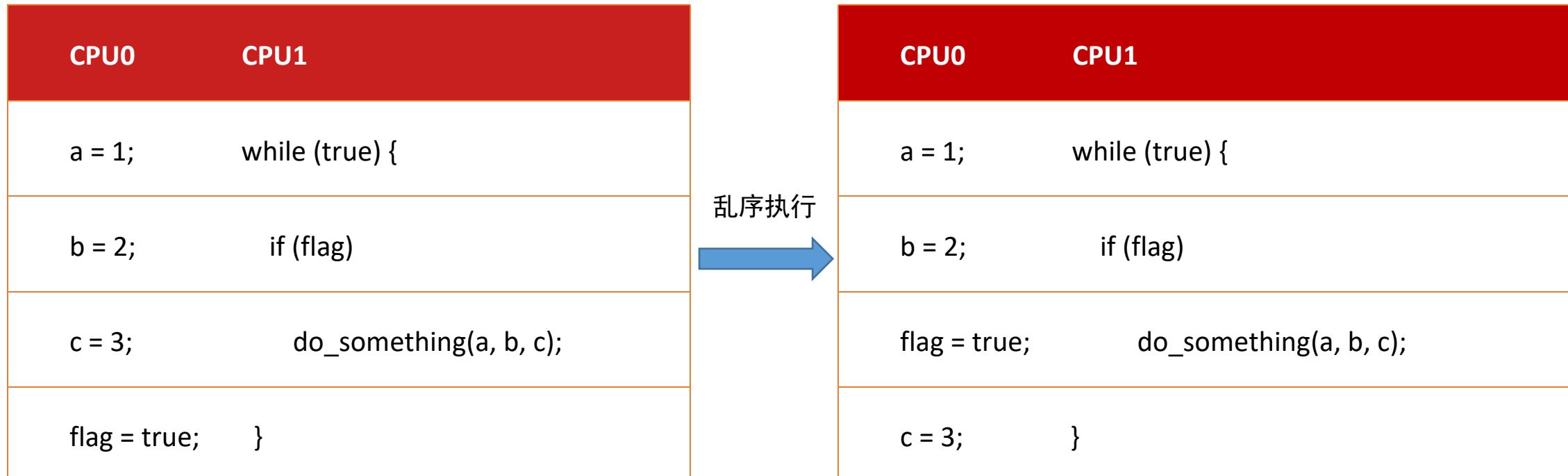
# 再一个例题

11. 假设有一个数据类型 complex 定义如下：
- (1) 什么是互斥？这段代码里哪些操作需要互斥？（2 分）
  - (2) 下图给出了 3 个线程对应的代码，假设线程的临时变量都放在栈区，请定义相应的信号量，用 PV 操作保证代码的正确性，并尽可能减少线程间不必要的等待。（8 分）

Complex 类型的代码示意	thead1	thread2	thread3
<pre>class Complex{ protected:     double real;     double img; public:     friend Complex add(Complex p, Complex q){         Complex s;         s.real=p.real+q.real;         s.img=p.img+q.img;         return s} .....//其他代码省略 }; Complex x, y, z; //共享的全局变量</pre>	<pre>Threadfunc1{     Comlpex w;     .....//无关代码     w=add(x,y);     .....//无关代码 }</pre>	<pre>Threadfunc2{     Comlpex n;     .....//无关代码     n=add(y,z);     .....//无关代码 }</pre>	<pre>Threadfunc3{     Comlpex w;     .....//无关代码     z=add(z,w);     y=add(y,w);     w=add(x,z);     .....//无关代码 }</pre>

# 多核SMP结构下指令可以乱序执行引发的新问题

考虑如下代码：



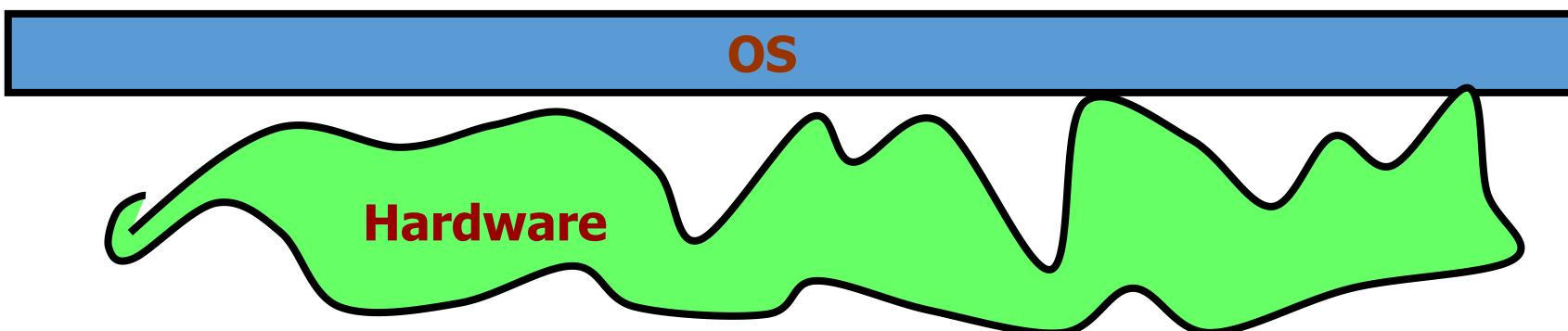
如果CPU0顺序执行，*a*、*b*、*c*初始化完毕后  
CPU1再*do something*，符合预期。

当CPU0乱序执行，*c*还未初始化CPU1就*do something*，出错。

Peterson算法和Deckers算法能够在理论上完成单核正确性的证明，但是在多核运行时，由于指令执行顺序交换而导致原来的依赖性问题无法保障，因此需要内存屏障指令才能保证其正确性

# 系统调用和保护机制

- 如何实现内存的保护？设置不同的权限
- 系统调用：切换系统的权限状态
  - 系统调用的本质是一个函数，一个由OS提供的函数
  - 提升权限的过程是一个中断，一个用户程序可以主动发起的中断
  - 系统调用的意义是什么？
  - 系统调用的代价是什么？
  - 系统调用（权限转换）有没有高性能的实现方式？（另一个有趣的科研话题，比如Intel的Sysenter & sysexit）



- 哪一个指令提升了权限？**int**
- 为什么提升了权限不会破坏安全性？  
因为权限提升之后只会执行固定的代码，而这段代码一定会回收权限
- 怎么知道要干什么？  
编号和参数是用户和OS的约定
- 为什么**syscall**的代价大？  
因为要有复杂的参数检查、安全性检查，导致运行了一些无用的代码，以及硬件上的清理工作

## 系统调用库接口示例

sfs\_filetest1.c: ret=read(fd,data,len);

```
.....
8029a1: 8b 45 10          mov  0x10(%ebp),%eax
8029a4: 89 44 24 08        mov  %eax,0x8(%esp)
8029a8: 8b 45 0c          mov  0xc(%ebp),%eax
8029ab: 89 44 24 04        mov  %eax,0x4(%esp)
8029af: 8b 45 08          mov  0x8(%ebp),%eax
8029b2: 89 04 24          mov  %eax,(%esp) ;以上代码在填充栈
8029b5: e8 33 d8 ff ff    call  8001ed <read> ; 这里的read是宏==6
syscall(int num, ...) {
...
asm volatile (
    "int %1; //这一句会个产生一个中断
    : "=a" (ret)
    : "i" (T_SYSCALL), //这句指定中断号
    "a" (num), //这一句的作用是把6放进了eax
    "d" (a[0]),
    "c" (a[1]),
    "b" (a[2]),
    "D" (a[3]),
    "S" (a[4])
    : "cc", "memory");
return ret;
```

```
// System call numbers
#define SYS_fork 1
#define SYS_exit 2
#define SYS_wait 3
#define SYS_pipe 4
#define SYS_write 5
#define SYS_read 6
#define SYS_close 7
#define SYS_kill 8
#define SYS_exec 9
#define SYS_open 10
#define SYS_mknod 11
#define SYS_unlink 12
#define SYS_fstat 13
#define SYS_link 14
#define SYS_mkdir 15
#define SYS_chdir 16
#define SYS_dup 17
#define SYS_getpid 18
#define SYS_sbrk 19
#define SYS_sleep 20
#define SYS_procmem 21
```

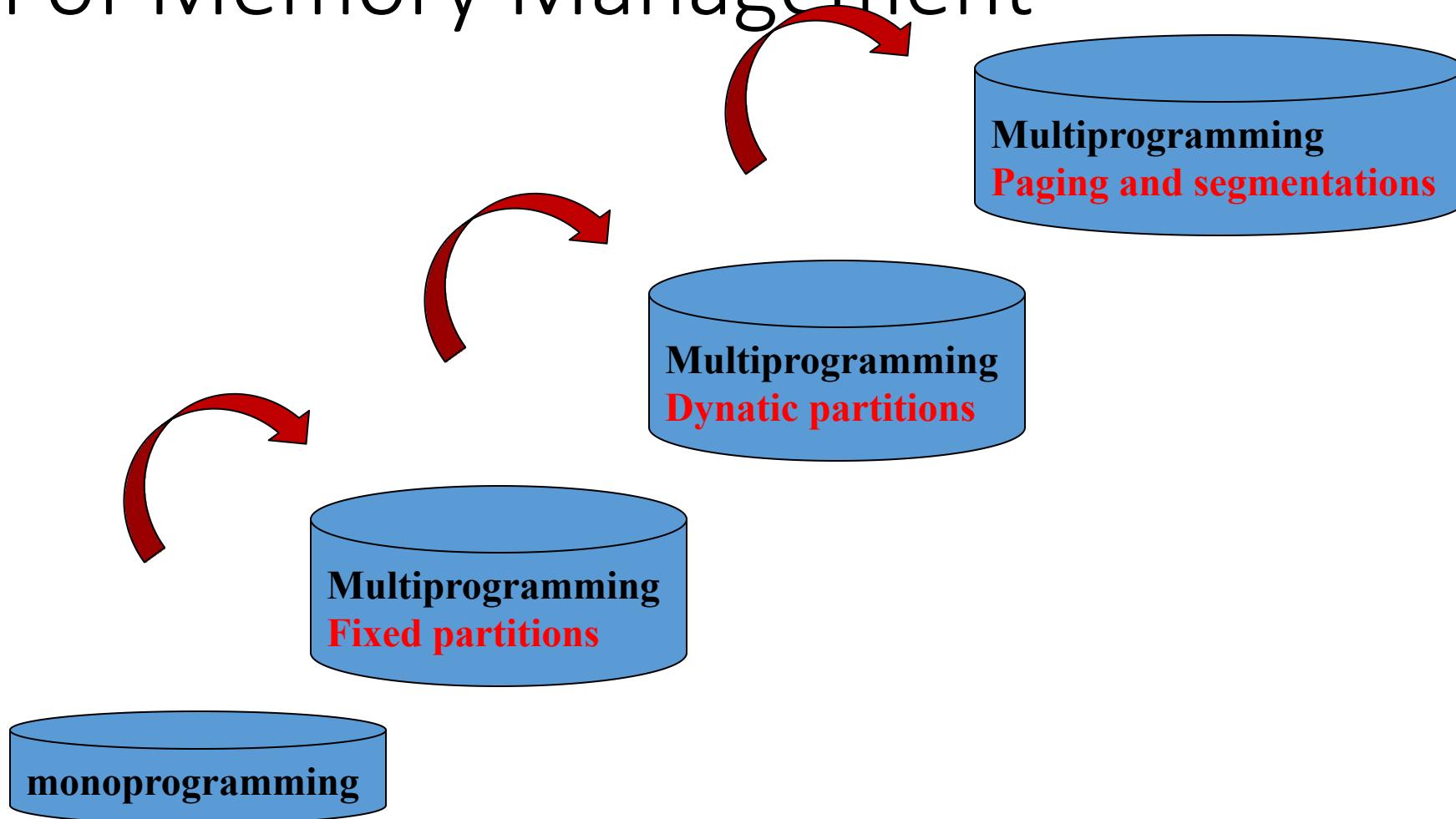
# 系统调用read(fd, buffer, length)的实现

1. kern/trap/trapentry.S: alltraps()
2. kern/trap/trap.c: trap()  
`tf->trapno == T_SYSCALL`
3. kern/syscall/syscall.c: syscall()  
`tf->tf_regs.reg_eax == SYS_read`
4. kern/syscall/syscall.c: sys\_read()  
从 `tf->sp` 获取 `fd, buf, length`
5. kern/fs/sysfile.c: sysfile\_read()  
读取文件
6. kern/trap/trapentry.S: trapret()

# 内存管理

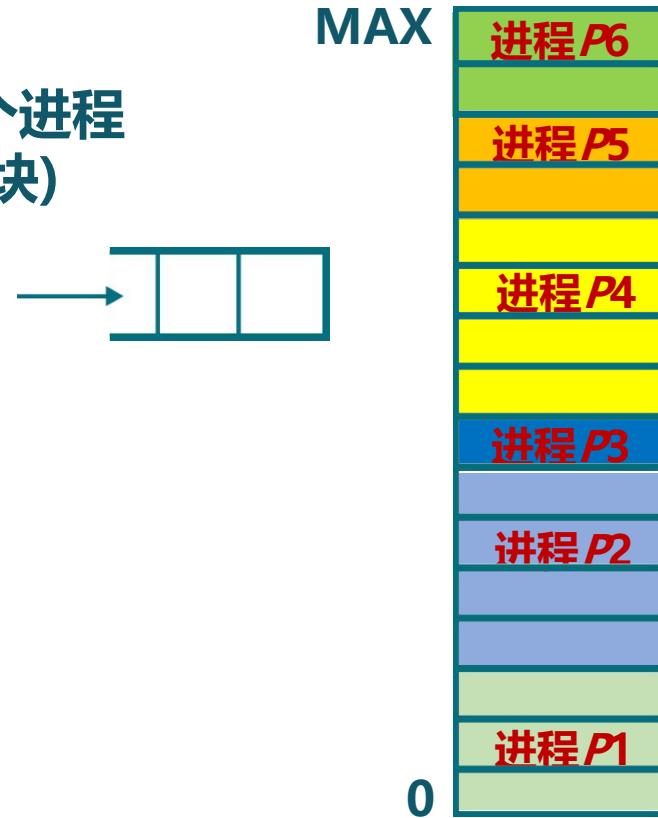
- 几个地址之间的关系
  - 逻辑地址、虚拟地址、线性地址、物理地址
  - 它们之间的转换是如何实现的
- 连续内存管理
  - 覆盖技术，紧缩技术
  - 内存空洞管理方法
- 页式内存管理
  - 页、页框、页表、地址转换、TLB
  - 页面置换算法
  - 虚拟地址管理
- 段式内存管理

# Model of Memory Management



# 连续内存分配：动态分区分配

- 动态分区分配
  - ▶ 当程序被加载执行时，分配一个进程指定大小可变的分区(块、内存块)
  - ▶ 分区的地址是连续的
- 操作系统需要维护的数据结构
  - ▶ 所有进程的已分配分区
  - ▶ 空闲分区(Empty-blocks)



# 碎片整理：紧凑(compaction)

## ■ 碎片整理

- ▶ 通过调整进程占用的分区位置来减少或避免分区碎片

## ■ 碎片紧凑(紧缩)

- ▶ 通过移动分配给进程的内存分区，以合并外部碎片

### ▶ 碎片紧凑的条件

- 所有的应用程序可动态重定位

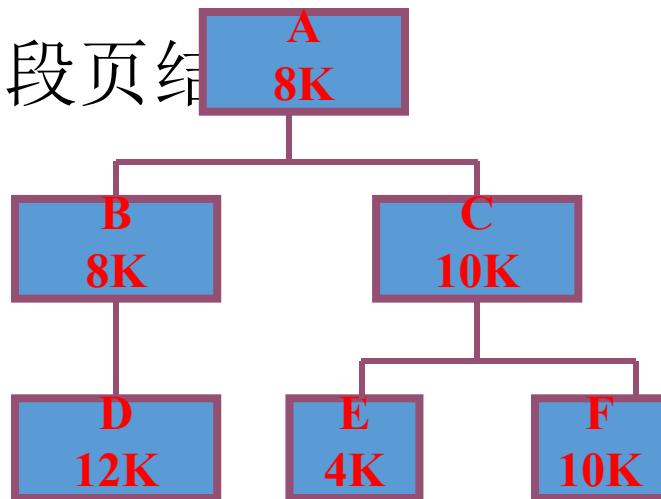
### ▶ 需要解决的问题

- 什么时候移动？
- 开销

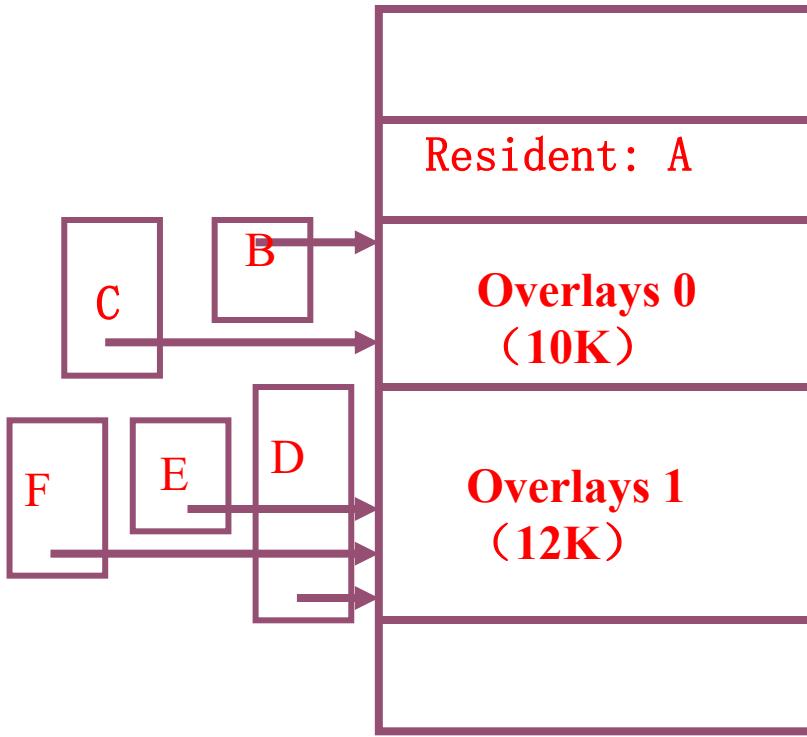


# 虚拟内存方案

- 覆盖式
- 分页式
- 分段式
- 段页结合

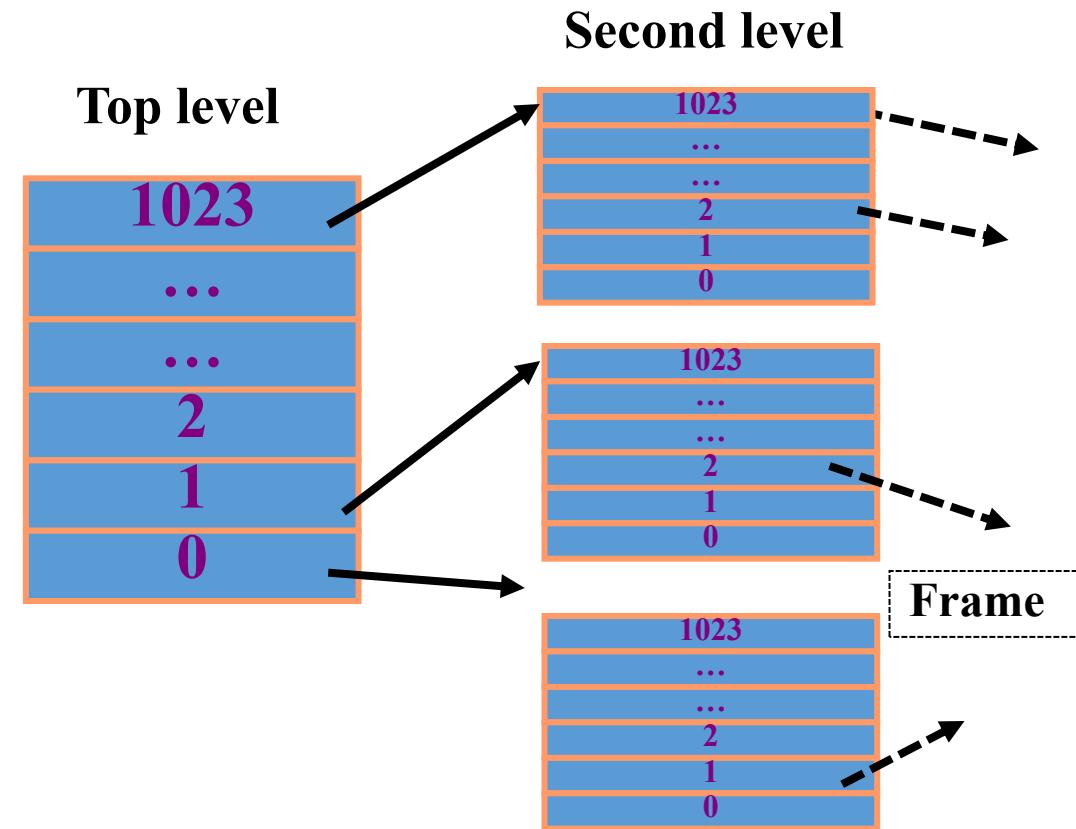
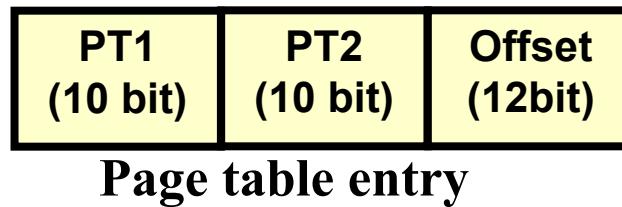


Structure of Job X



# 页表与多级页表

- 要有页表的设计能力



还记得对这个页表的解释吗？

# 4K产生了什么深远意义的影响？

- 页对齐发生在系统的哪些地方?
  - Elf文件中的数据格式?
  - 缓存向IO更新的数据块大小?
  - 网络数据传输?
  - .....

# X64的页表是什么样的

- 如果给你

Virtual Address

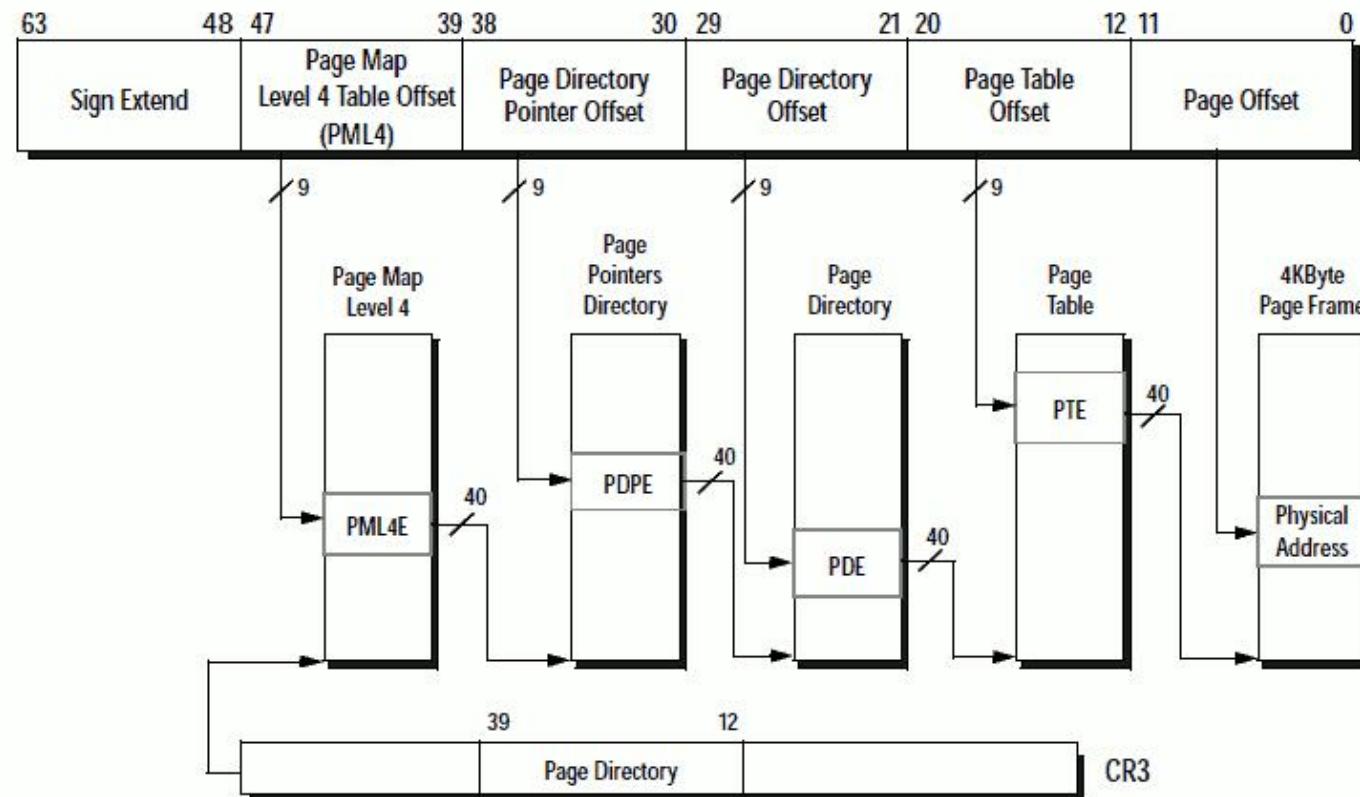
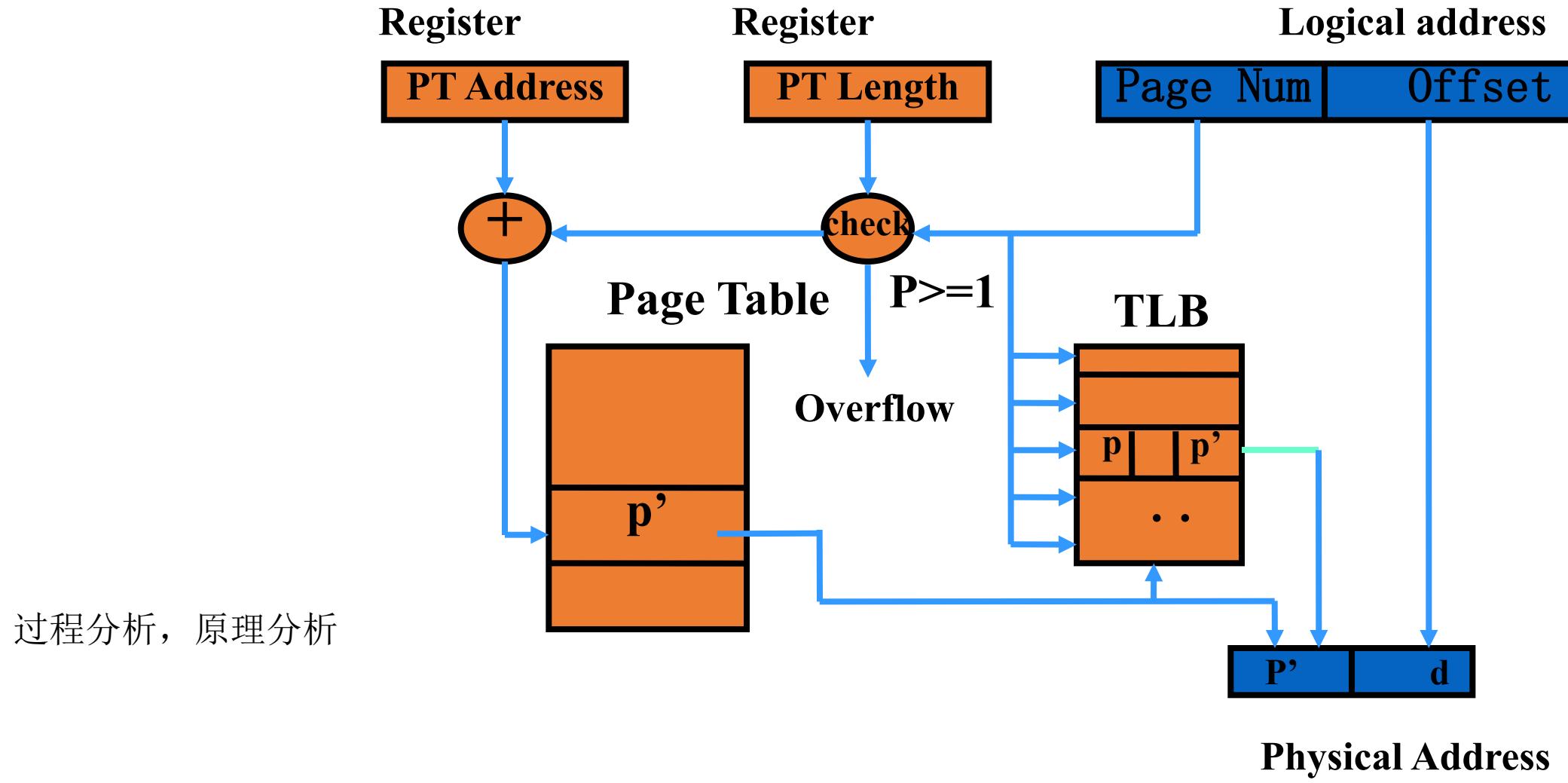


Figure 17. 4KB-Page Translation

# X64的页表是什么样的

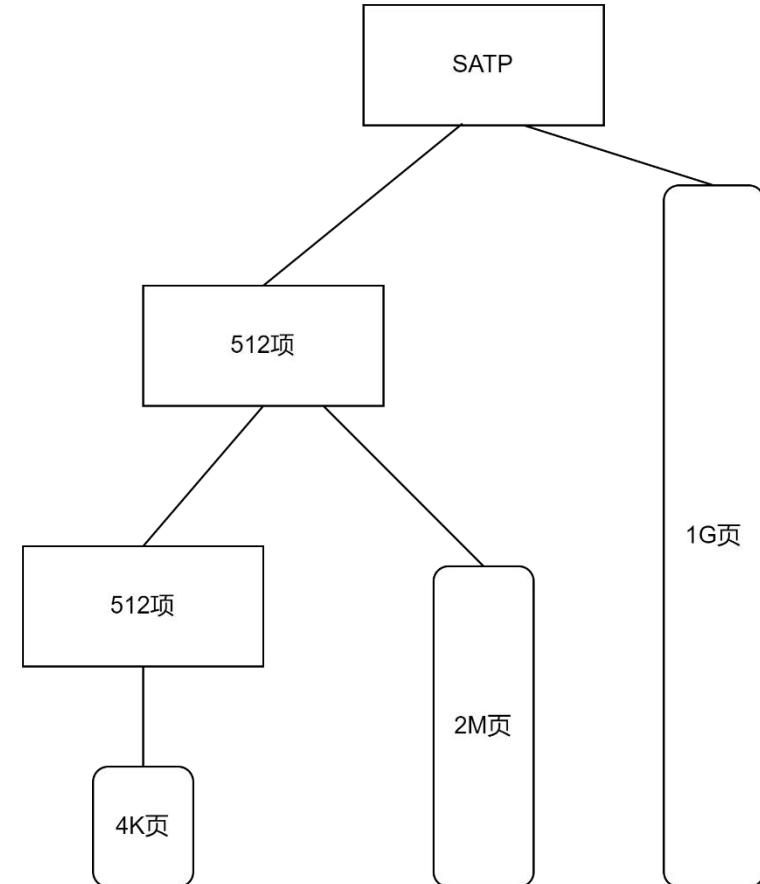
- 每个页表里面有512项
  - $4\text{k}/8\text{字节}(64\text{bit}) = 512$
  - 因此用于页表项的寻址部分仅有9位
- 类似的，页目录项也只有512项
  - 页目录项也只有9位
- 页表扩展为4级，每级都是9位地址
  - 寻址空间最大为256T
  - 最高的16位目前没有使用

# Mechanism of TLB

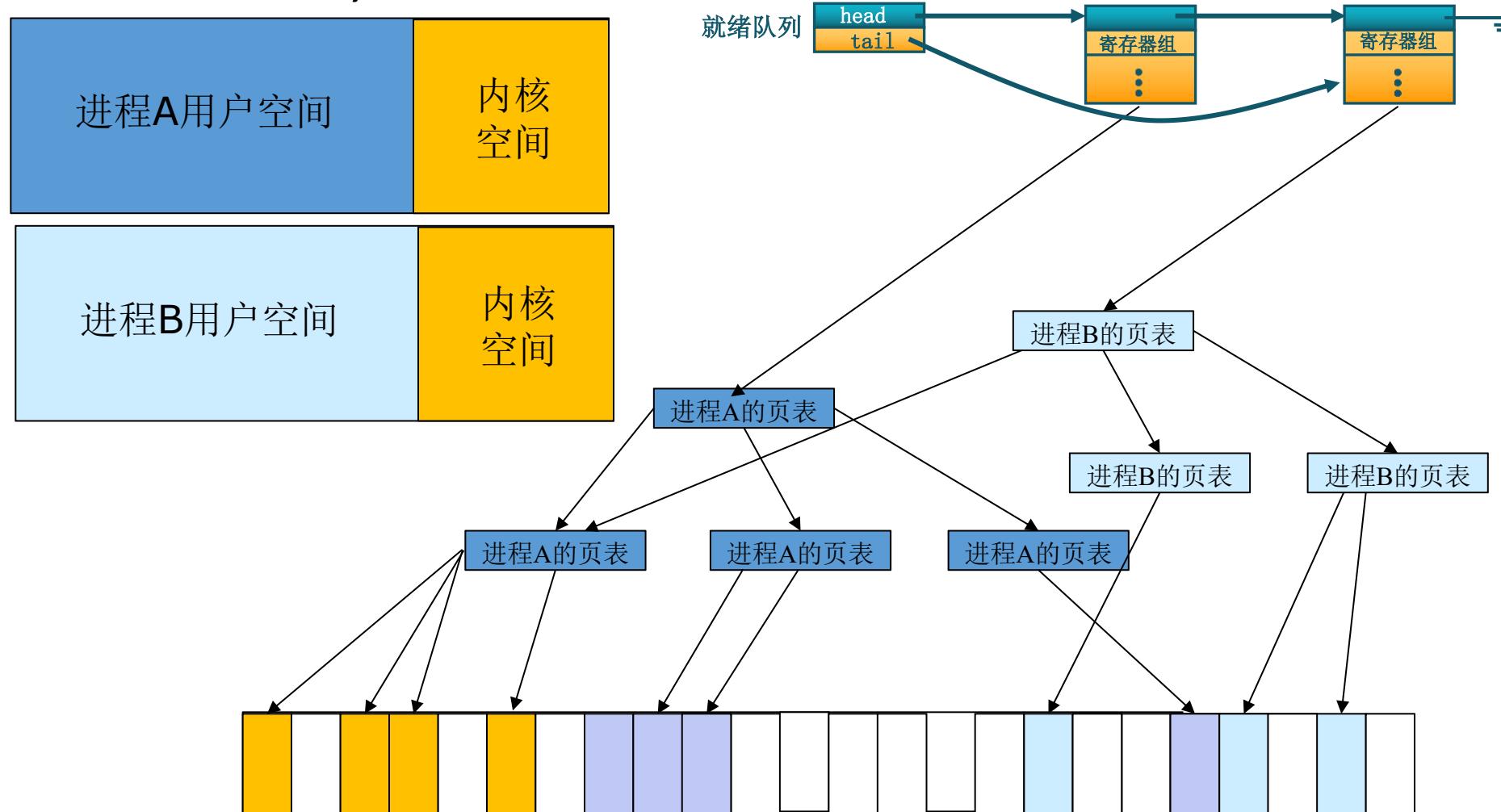


# 4K的页真的是不可动摇的吗：回忆ucore

- 4K的页
- 2M的页
- 1G的页
- 大页有什么优缺点？
  - 省页表的查找时间
  - 省TLB项
  - 浪费内存（是吗？）
  - 破坏兼容性

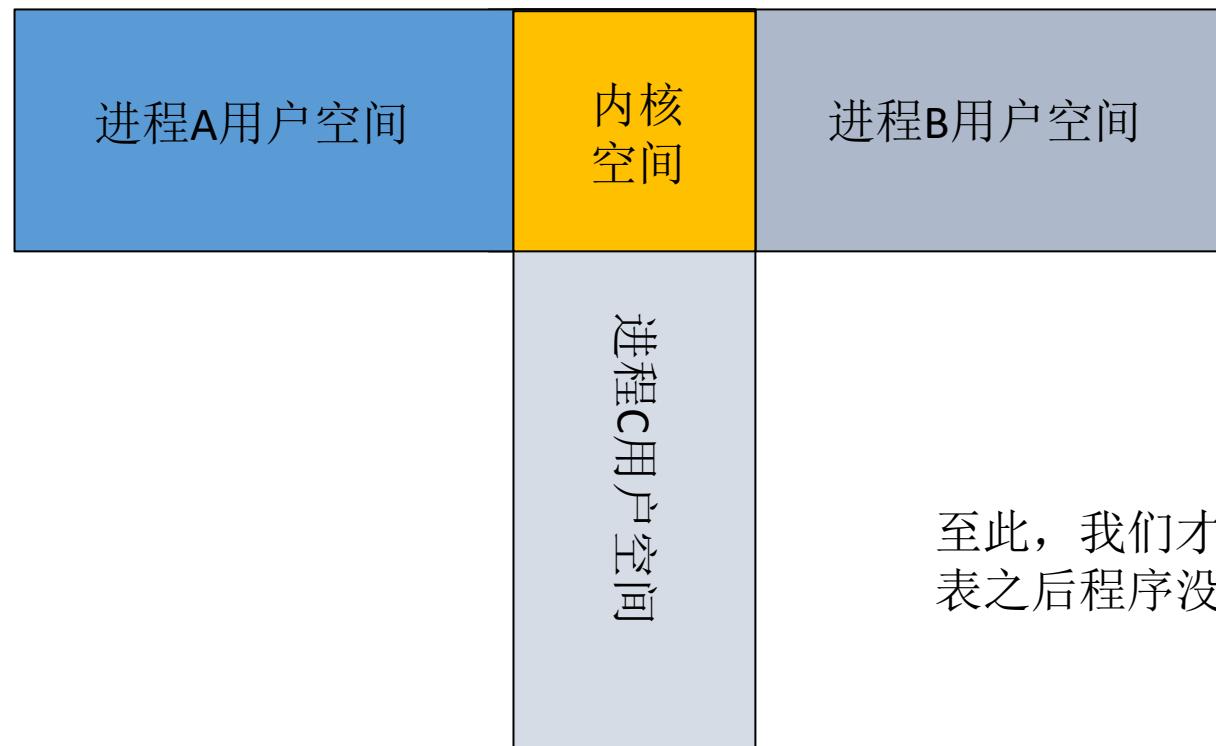


# fork调用后,exec调用后



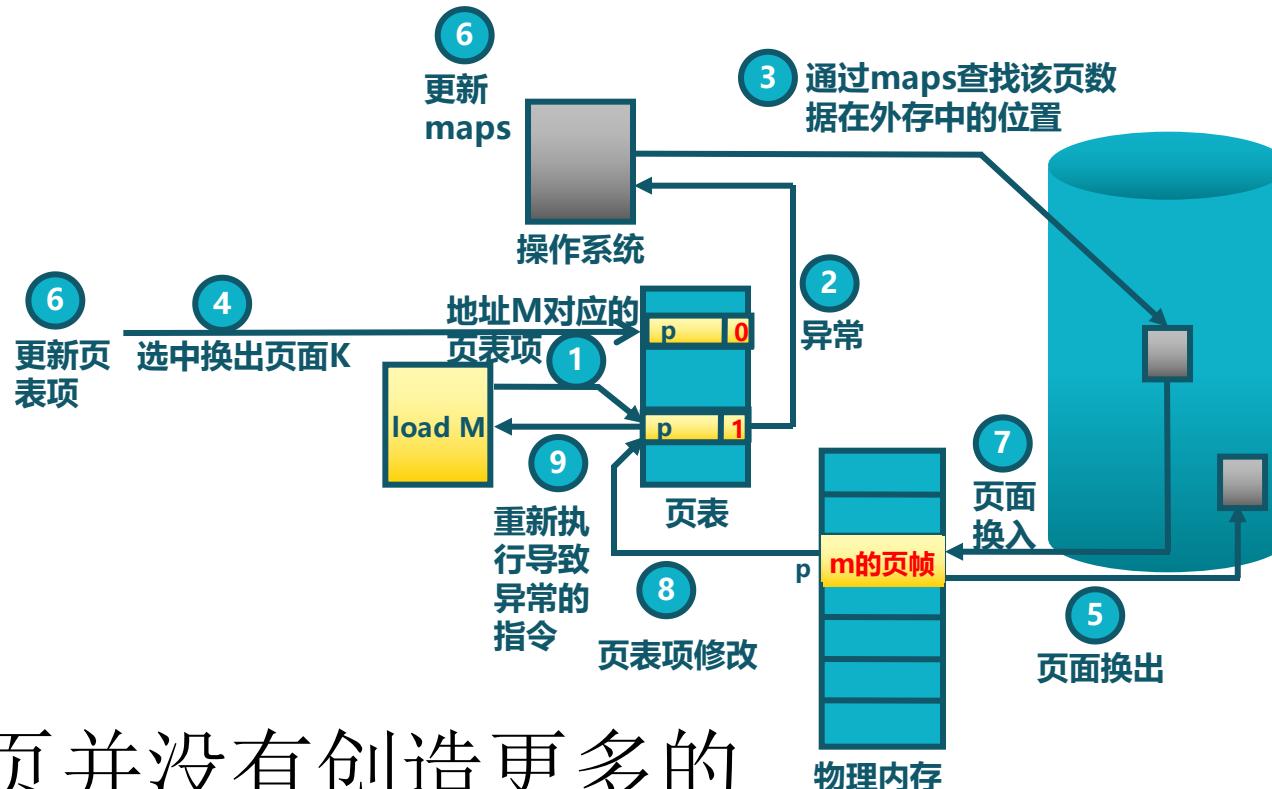
# 虚拟地址空间真正的布局

- 这样有什么风险？



至此，我们才回答了为什么切换页表之后程序没有受到影响

## 缺页异常（缺页中断）的处理流程



分页并没有创造更多的内存，分页使得内存可以被延迟加载/按需加载

1. CPU发出访存指令到MMU, MMU根据地址高位部分得到页号, 以页号查页表;
2. 查表发现该页表项未填入有效值, 因此产生缺页异常
3. 缺页异常由OS响应, OS需要分配对应的页帧, 并找到该页的内容在磁盘中的位置;
4. OS发现没有可用的空间页帧, 于是启动页面换出过程, 选出准备换出的页面;
5. 找到待换出页面对应的页帧, 将其数据写回到磁盘上以避免丢失;
6. 将换出页的页表项失效, 并修改maps记录该页内存的存储位置, 同时将页帧标记为空闲;
7. 找到系统中的空闲页帧, 从磁盘中读取相应的数据填入页帧中;
8. 将页帧号填入页表中, 并修改页表的有效位;
9. 异常返回, 重新执行产生缺页的指令

# Summary: Page replacement algorithm

理解算法的原理  
推演算法的过程

	Principle	Performance
最优算法	按照轨迹置换最远的未来使用的页面	性能最好，但是需要预知未来
NRU	“RM” classification	<b>Not bad, time-consuming</b>
FIFO	Time sorting	<b>Not reasonable</b>
LRU & NFU	Locally optimal	<b>Approach “perfect”</b>
工作集	找到最适合工作集的内存域	<b>Stable and efficiency</b>
缺页率算法	根据缺页发生的	<b>Stable, efficiency and practical</b>

# 示例：工作集置换算法过程推演

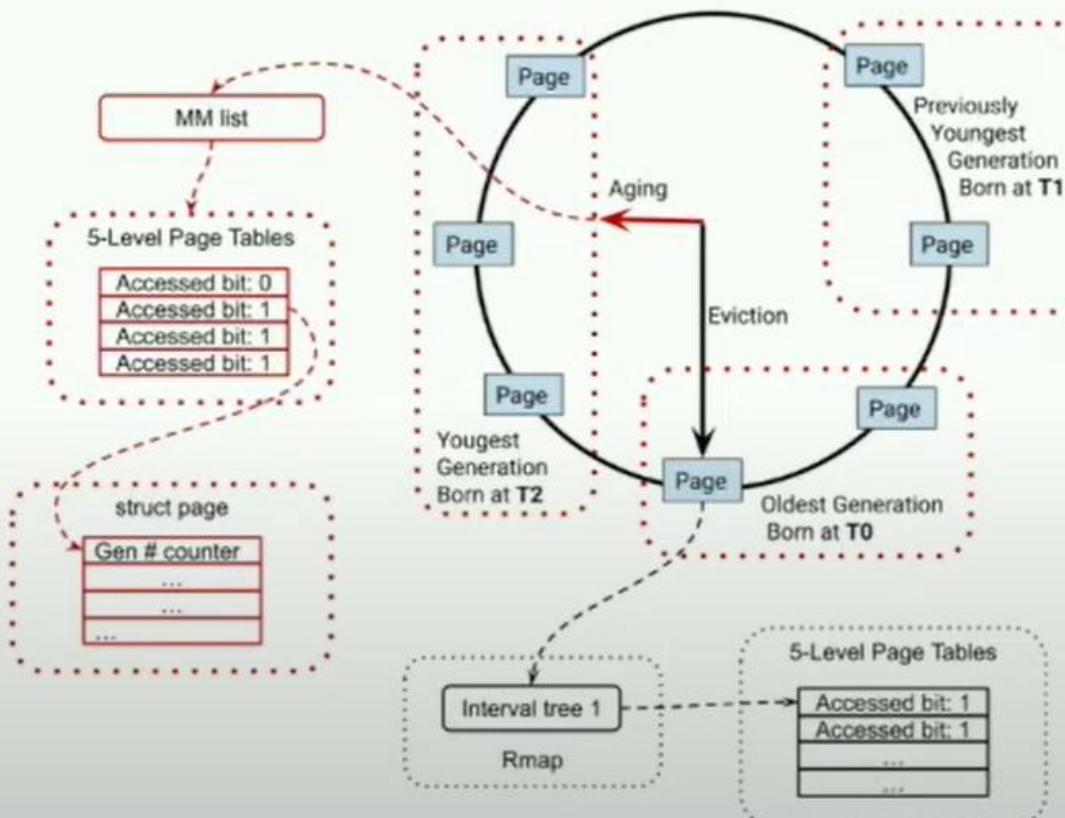
$\tau = 4$

时间	0	1	2	3	4	5	6	7	8	9	10
访问页面	c	c	d	b	c	e	c	e	a	d	
逻辑页面状态	页面a t=0	●	●	●					●		
逻辑页面状态	页面b t=-1		●	●	●	●	●	●	●	●	
逻辑页面状态	页面c t=-2	●		●	●	●	●	●	●	●	
逻辑页面状态	页面d t=-3		●	●	●	●	●	●	●	●	
逻辑页面状态	页面e t=-4	●				●	●	●	●	●	
缺页状态		○		○		○		○		○	

## ● MGLRU (Linux内核中决策换出哪一页的新机制)

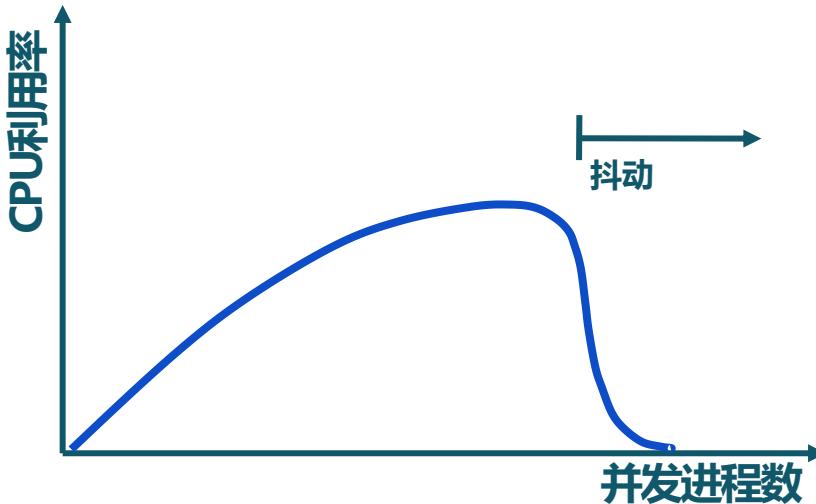
### MGLRU internals

- Generations
- Page table walks
- Feedback loops



图片来源: <https://www.youtube.com/watch?v=9HvJfN21H9Y>

# CPU利用率与并发进程数的关系



- CPU利用率与并发进程数存在相互促进和制约的关系
  - ▶ 进程数少时，提高并发进程数，可提高CPU利用率
  - ▶ 并发进程导致内存访问增加
  - ▶ 并发进程的内存访问会降低了访存的局部性特征
  - ▶ 局部性特征的下降会导致缺页率上升和CPU利用率下降

# 页面置换算法的问题

- Belady现象是什么？**增加页面不能降低缺页。**它的本质是什么？
  - 是什么导致了增加物理页不能降低缺页
  - 违背了局部性原理是不是就是违背程序员世界的真理？
- LRU看起来很不错，为什么不在现在的OS中使用？
  - 如何探测工作集，仍然是一个开放问题
  - 借助硬件、编译器或者程序员的帮助，会更加有效的找到工作集
  - LRU剩下的东西，是不是工作集？
- 抖动问题的根本原因是什么？有没有“两全齐美”的解决方案？

通过QPI点对点互联

通过QPI点对点互联

QPI

NUMA节点

CPU

本地内存

通过QPI点对点互联

通过QPI点对点互联

NUMA节点

CPU

本地内存

NUMA节点

CPU

本地内存

知乎 @沉默的猫

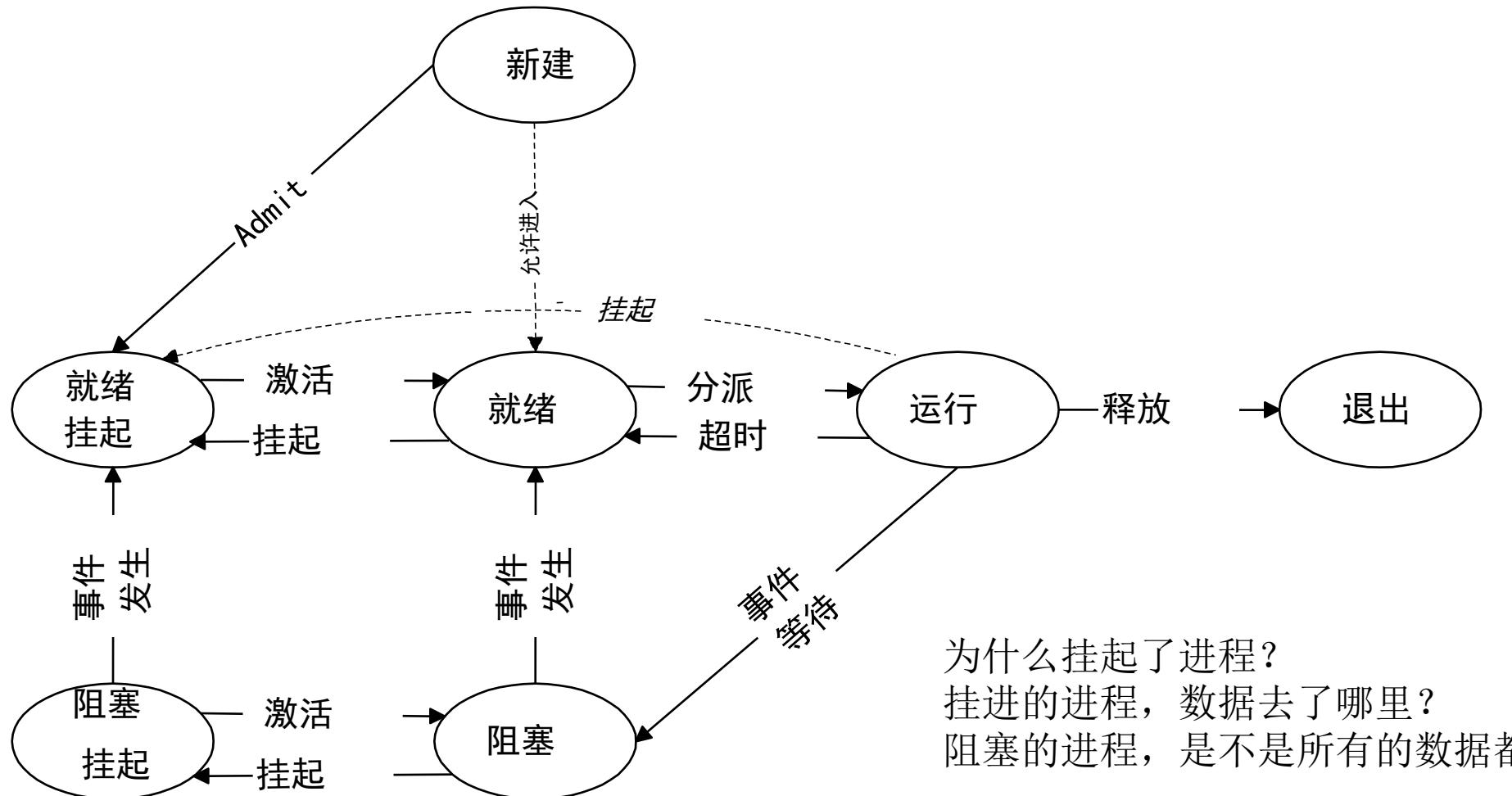
# 页面置换算法在NUMA问题的应用

- 为什么LRU、LFU看起来很好，不在NUMA里面使用？
  - 难以有效的追踪每一句的访存
  - 难以高效存储整个访存记录
  - 即使有了这些记录，利用过去预测未来仍然充满挑战
  - LRU剩下的东西，是不是工作集？

# 页面置换算法的问题

- 我们为了缓解内存不足都做了什么？
- 一个进程不常用的页面，可以提前换出去
  - 什么状态的进程？
- 一个不怎么运行的进程，可以整体换出去
  - 什么状态的进程？

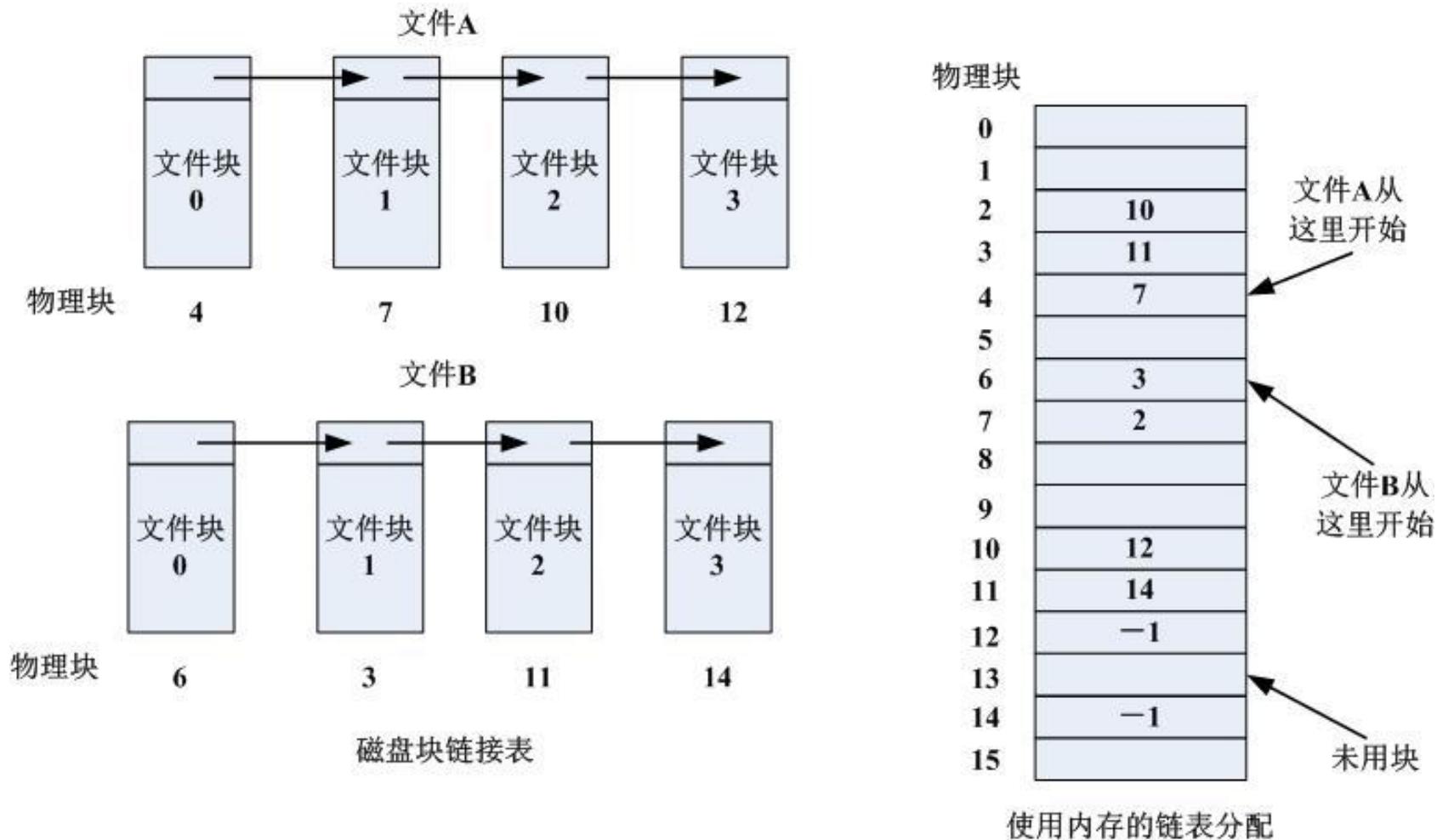
# 挂起进程模型



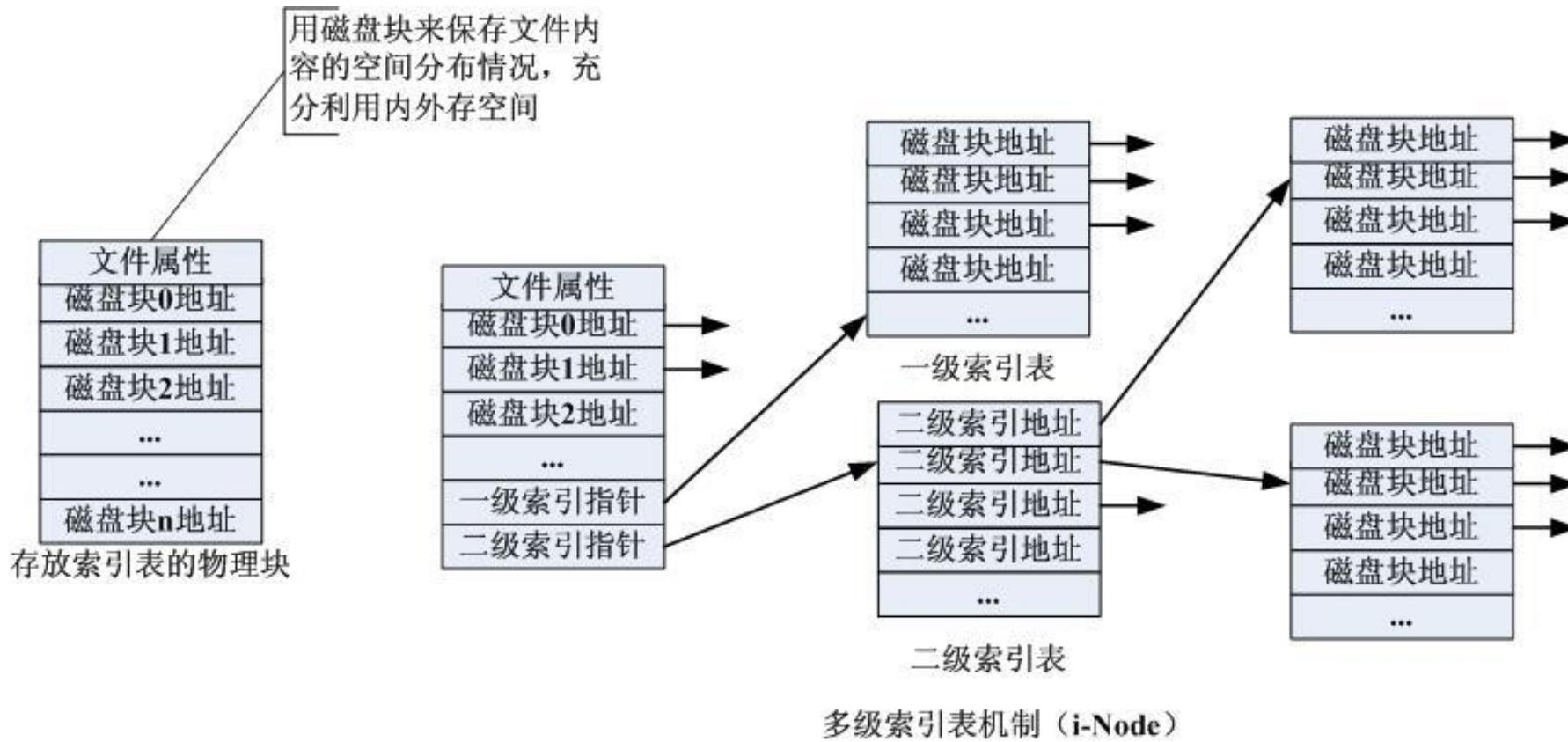
# 文件系统

- 文件的管理
  - 链表型、树型
- 文件夹的管理
- 文件数据一致性
- VFS
- 用户权限

# Physical structure of file: link table



# Physical structure of file: index table



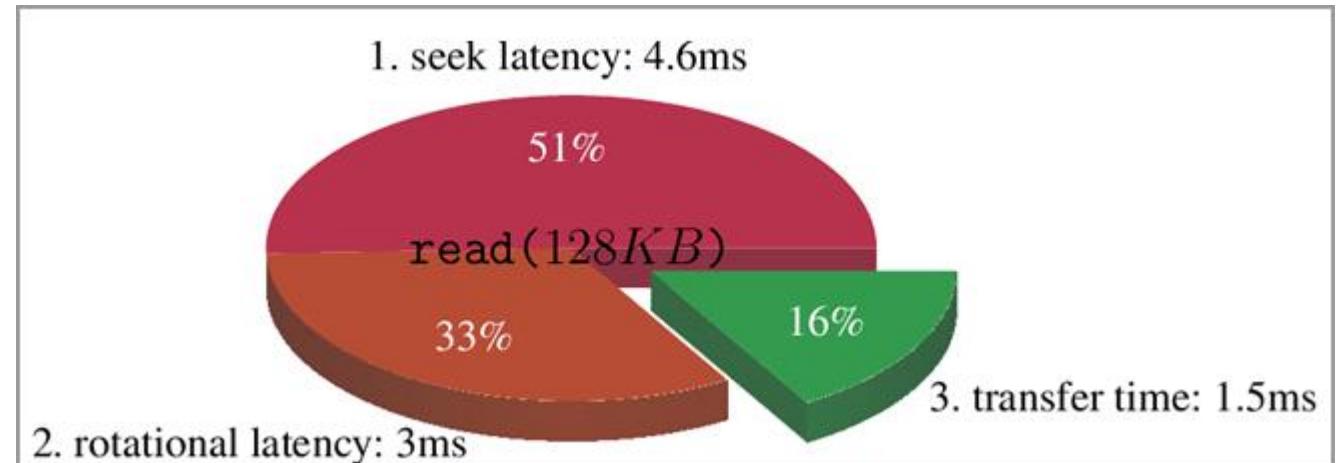
# Summary of file physical structure

理解相应数据结构的原理  
推演文件系统访问磁盘的过程

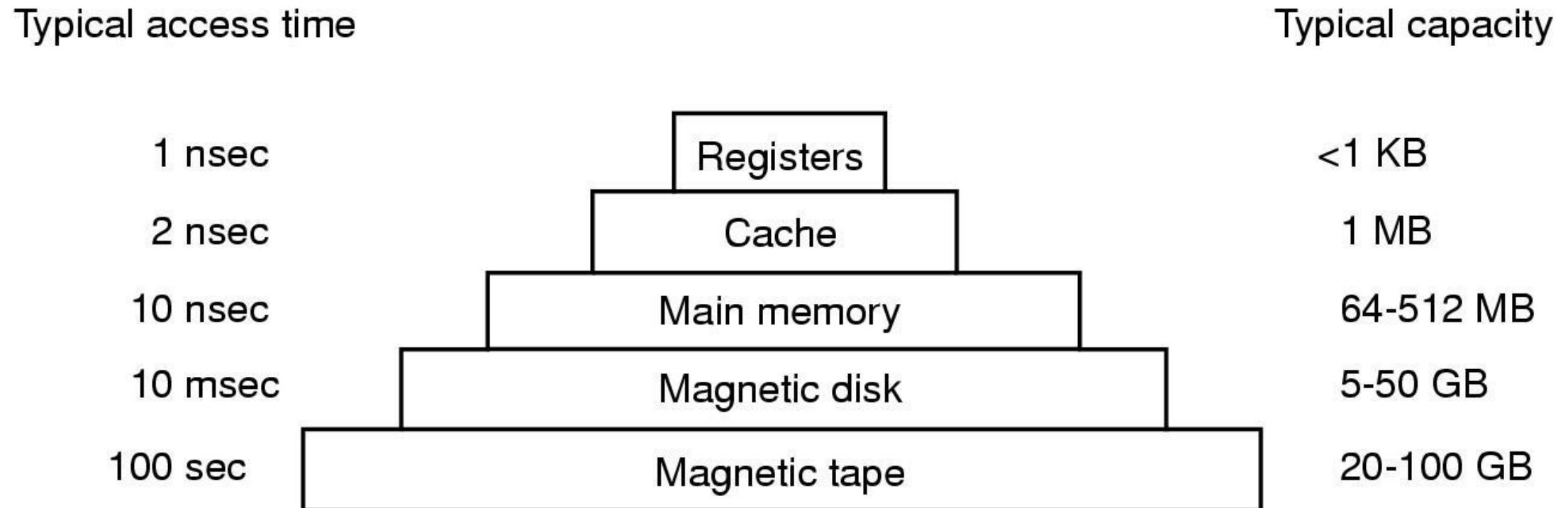
		Continuous	Link table	Index table
media	tape	Supported	Unsupported	unsupported
	disk	Supported	supported	supported
Access mode		Sequential & random	sequential	Sequential & random
Efficiency		Low	Middle	High
Application		To simple to be used	Not popular	Widely used

# 文件系统与硬盘管理的结合

- 文件在逻辑上是树形的
- 文件在物理上是线性的
- 文件系统解决的是数据块在扇区和磁道上的分布
- 硬盘的调度算法是控制磁臂移动的
- 优化的目标：
  - 减少磁盘读写次数
  - 减少磁臂的移动



# 存储体系结构的使用



# 计算机世界的时间

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

## 7. 磁盘访问调度策略

来自不同进程的磁盘I/O请求会构成一个随机分布的请求队列。对磁盘访问包含了三个因素：

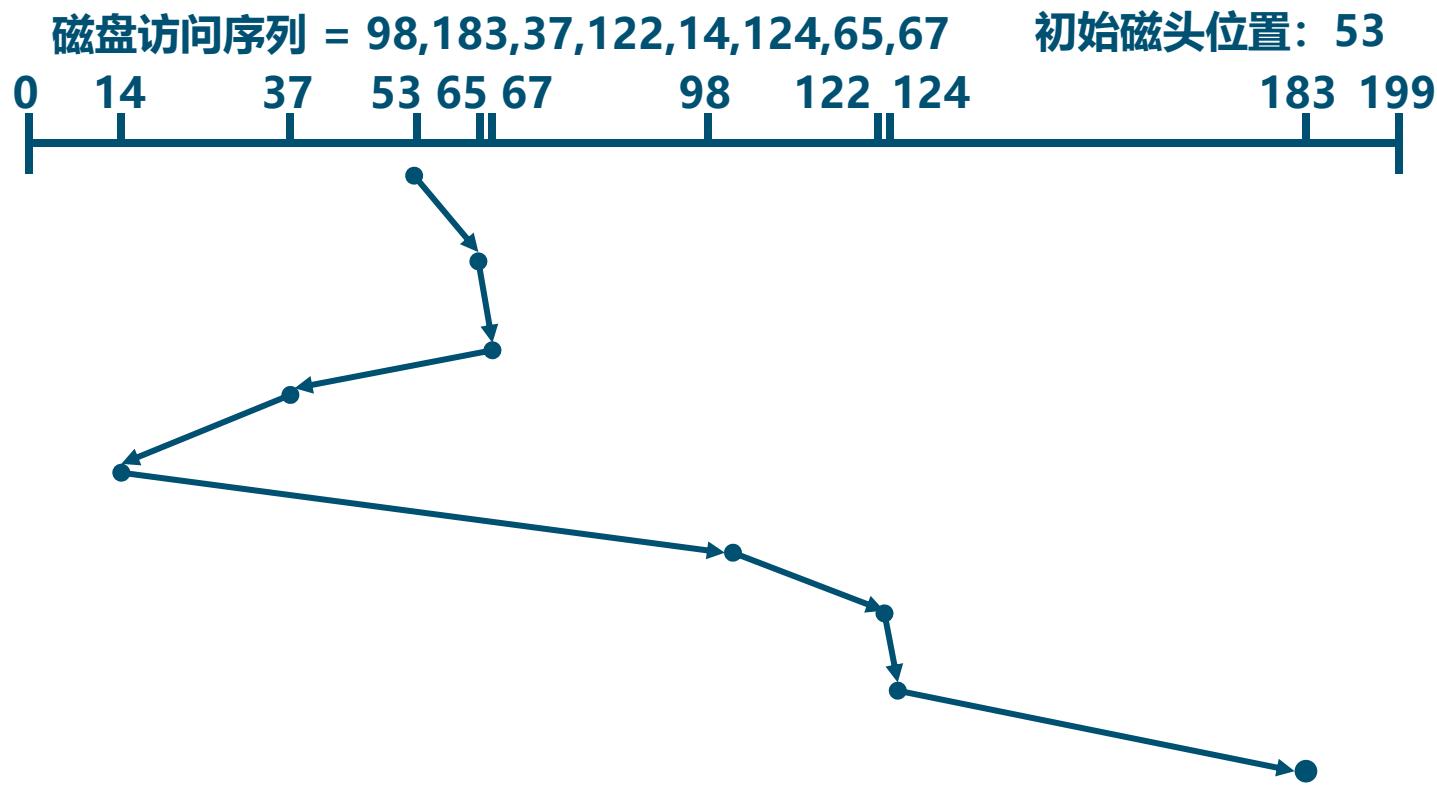
- 1 ) 磁头臂移到柱面的时间（寻道）
- 2 ) 等待扇区到位时间（旋转延时）
- 3 ) 数据传输时间

磁盘调度的主要目标就是减少请求队列对应的平均柱面定位时间。

## 算法描述：

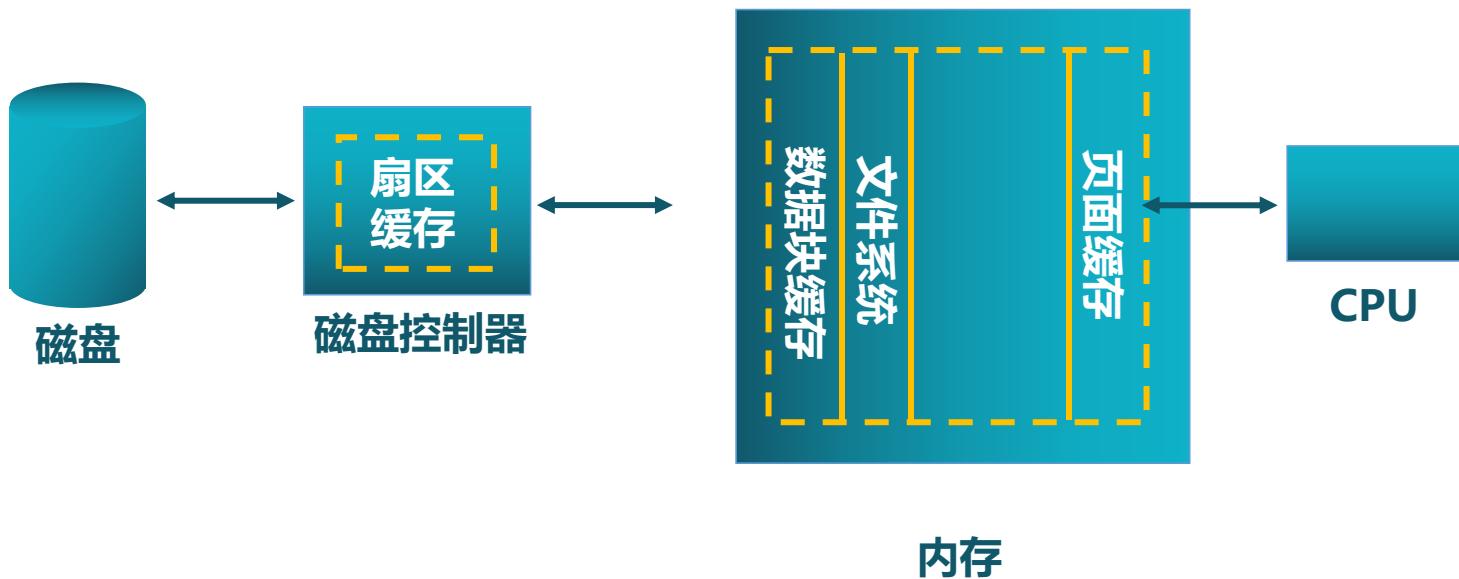
- 先进先出算法（FIFO）：按进入队列的先后分配磁盘。体现公平。
- 优先级算法（PRI）：按请求进程的优先级分配磁盘。满足进程性能需要。
- 后进先出算法（LIFO）：后进入请求队列的进程先分配。符合局部性原理，可使资源利用率较高。

# SSTF算法示例



合计磁头移动距离=  $12+2+30+23+84+24+2+59=236$

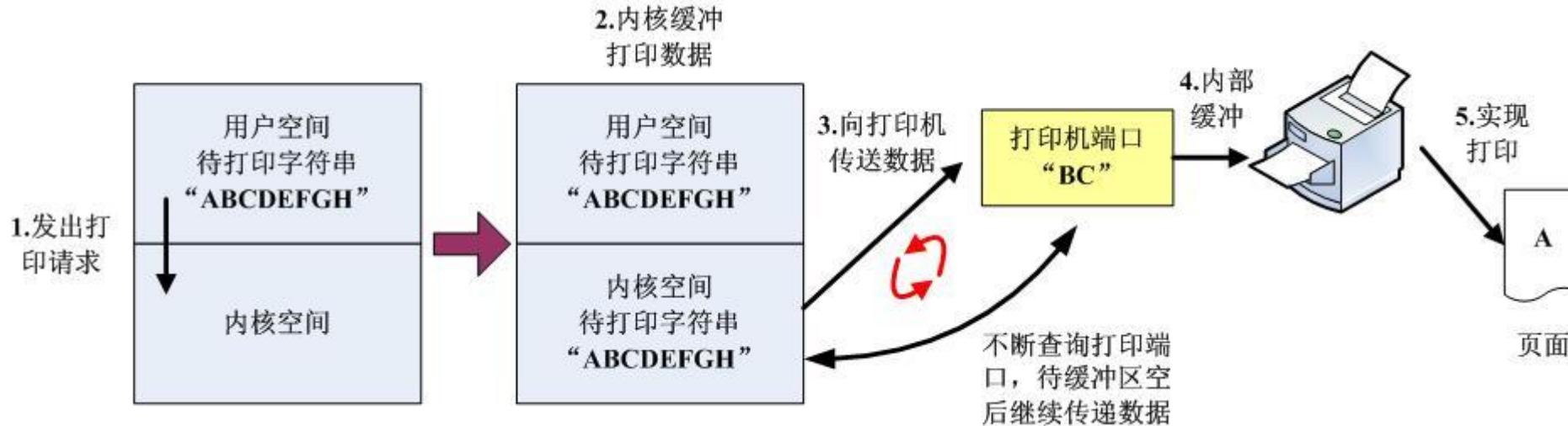
# 多种磁盘缓存位置



# 设备管理

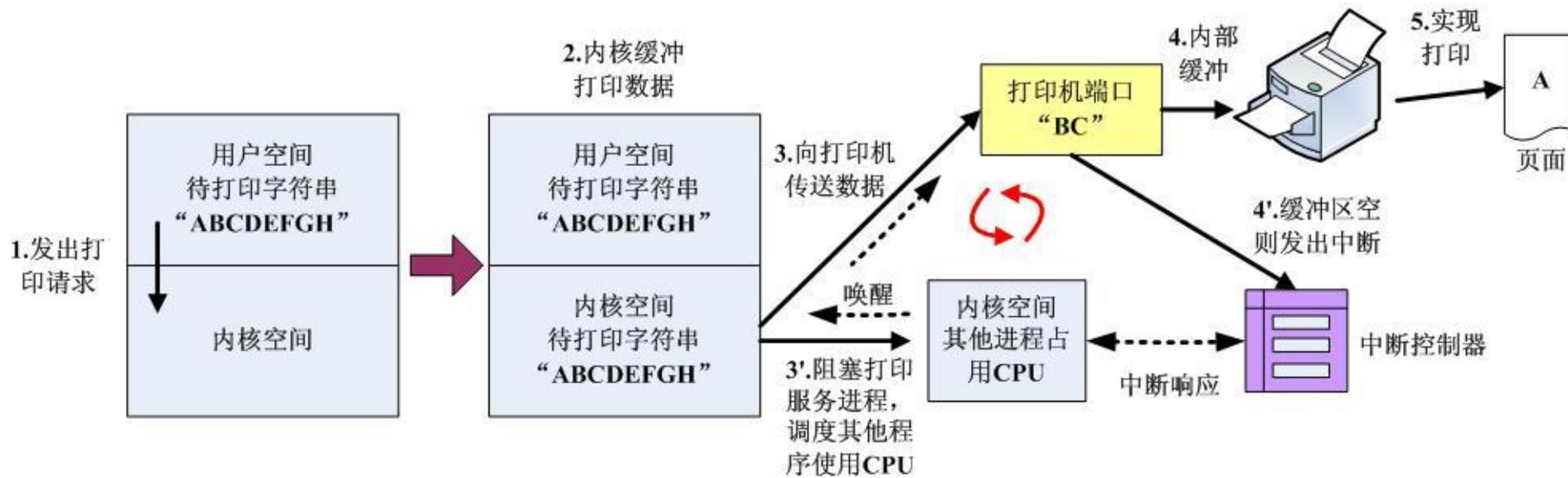
- IO资源的管理
  - 独立编址、共享编址、混合式
- 三种设备的通信模式
  - 忙等待、中断、DMA
- 驱动程序设计中的典型问题
  - 异步、缓冲、spooling
- 硬盘管理中的若干问题
  - RAID
  - 磁臂管理

# Working mode of devices: busy waiting



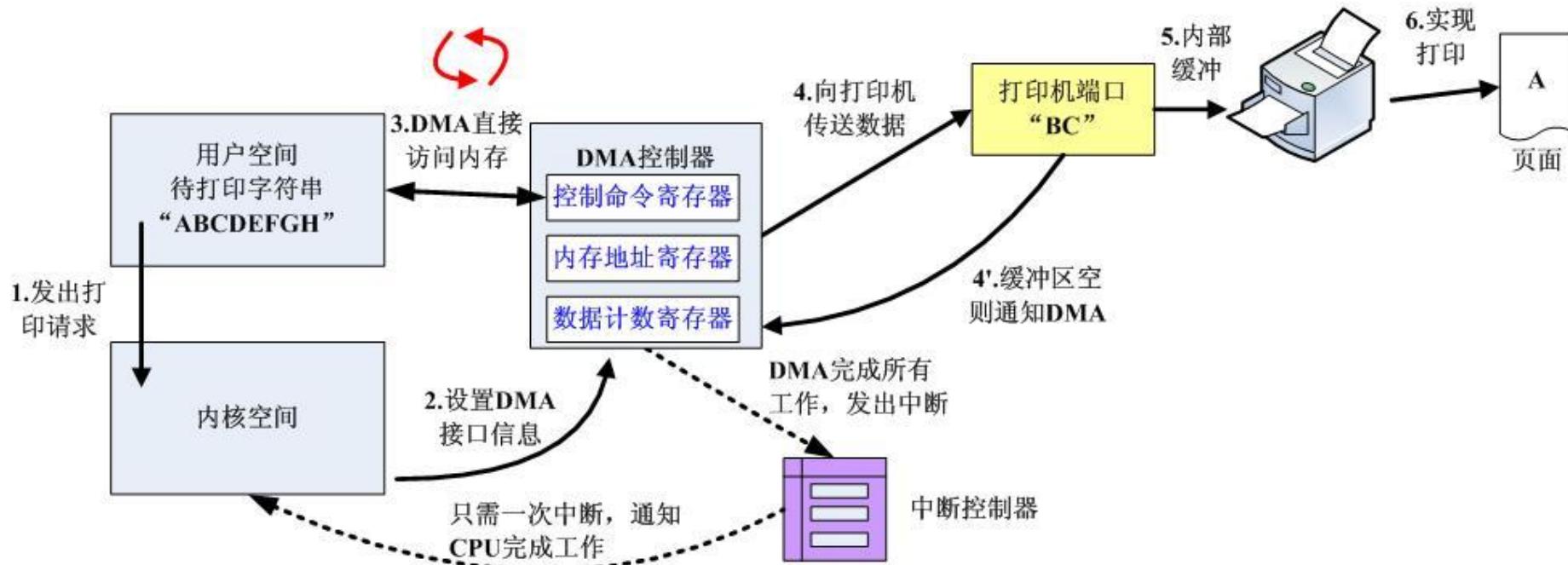
- Special kernel process sends the data to device port;
- The process checks the port repeatedly until the port is available and sends rest data;
- The user process continues run after the kernel process is finished;
- Disadvantage: CPU is wasted too much

# Working mode of devices: interrupt

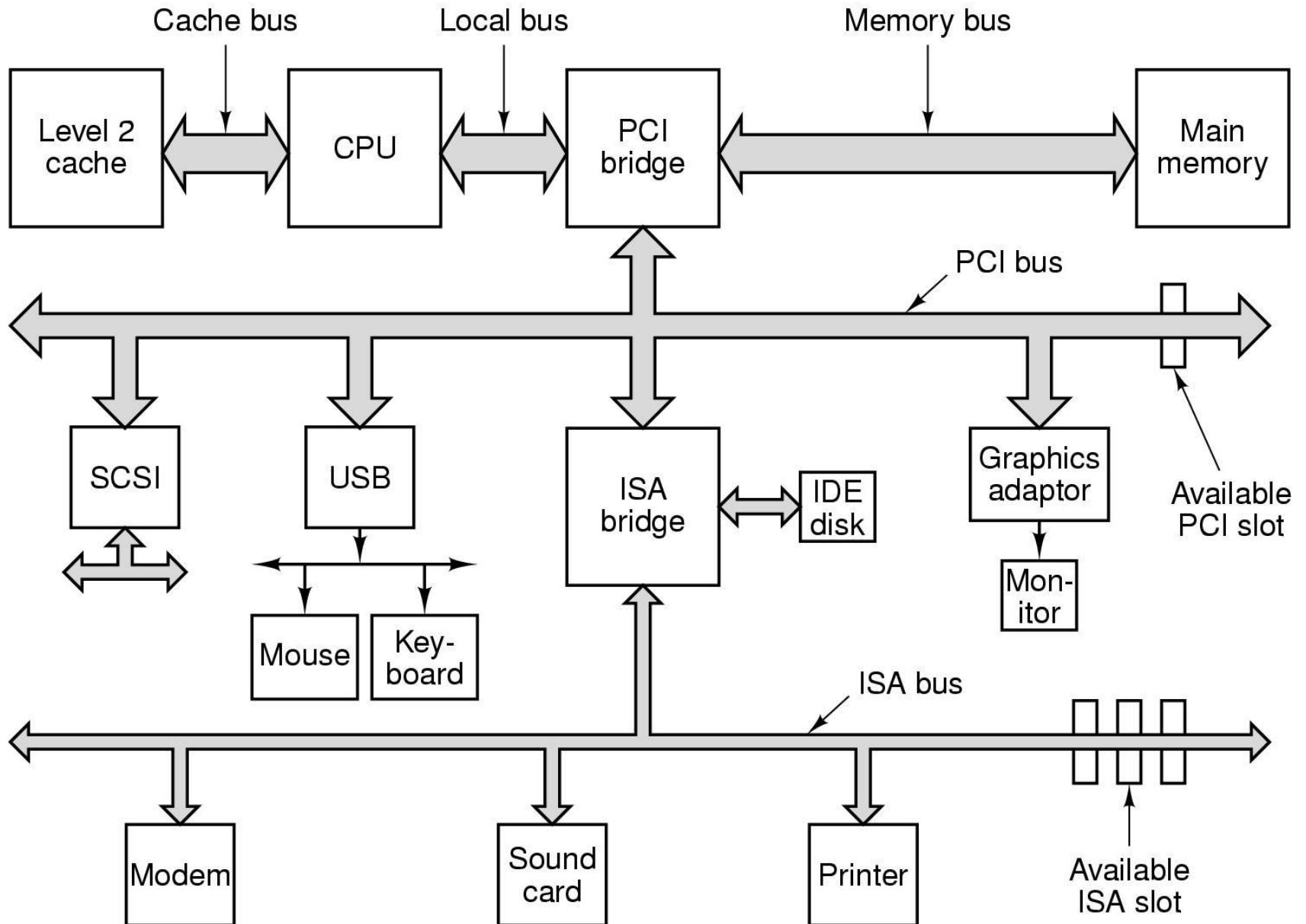


- Special kernel process sends the data to device port;
- The process goes to sleep and CPU will run other processes;
- The device send interrupt to CPU after the data buffer is empty;
- The kernel process is waked up and send the rest data
- Disadvantage: frequent interrupts are time-consuming

# Working mode of devices: DMA



- User process causes a CPU trap, the special kernel process sets the registers in device and exits;
- Device read data from memory directly;
- The device send interrupt to CPU after the job is done;
- The user process is waked up and continues run



# 死锁

- 四种方案
- 银行家算法

方法	资源分配策略	各种可能模式	主要优点	主要缺点
预防 Prevention	保守的；宁可资源闲置(从机制上使死锁条件不成立)	一次请求所有资源<条件 1>	适用于作突发式处理的进程；不必剥夺	效率低；进程初始化时间延长 剥夺次数过多；多次对资源重新起动 不便灵活申请新资源
		资源剥夺<条件 3>	适用于状态可以保存和恢复的资源	
		资源按序申请<条件 4>	可以在编译时（而不必在运行时）就进行检查	
避免 Avoidance	是“预防”和“检测”的折衷（在运行时判断是否可能死锁）	寻找可能的安全的运行顺序	不必进行剥夺	使用条件：必须知道将来的资源需求；进程可能会长时间阻塞
检测 Detection	宽松的；只要允许，就分配资源	定期检查死锁是否已经发生	不延长进程初始化时间；允许对死锁进行现场处理	通过剥夺解除死锁，造成损失
忽略 ignore	不理睬死锁问题，认为它不是主要问题（鸵鸟法）			

# 练习

1. 处理外部中断时，应该由操作系统保存的是( )。  
A. 程序计数器(PC)的内容                           B. 通用寄存器的内容  
C. 块表(TLB)中的内容                           D. Cache 中的内容
  
2. 执行系统调用的过程包括如下主要操作：① 返回用户态；② 执行陷入(trap)指令；③ 传递系统调用参数；④ 执行相应的服务程序。正确的执行顺序是( )。  
A. ②→③→①→④                                   B. ②→④→③→①  
C. ③→②→④→①                                   D. ③→④→②→①

3. 在进程转换时，下列()转换是不可能发生的。
- A.就绪态→运行态                  B.运行态→就绪态  
C.运行态→阻塞态                  D.阻塞态→运行态
4. 下列关于管道(Pipe)通信的叙述中，正确的是()。
- A.一个管道可实现双向数据传输  
B.管道的容量仅受磁盘容量大小限制  
C.进程对管道进行读操作和写操作都可能被阻塞  
D.一个管道只能有一个读进程或一个写进程对其操作

5. ( )有利于CPU繁忙型的作业，而不利于I/O繁忙型的作业。

- A.时间片轮转调度算法
- C.短作业(进程)优先算法

- B.先来先服务调度算法
- D.优先权调度算法

6. 下列进程调度算法中，综合考虑进程等待时间和执行时间的是( )。

- A.时间片轮转调度算法
- C.先来先服务调度算法

- B.短进程优先调度算法
- D.高响应比优先调度算法

7. 以下不是同步机制应遵循的准则的是( )。

- A.让权等待
- C.忙则等待

- B.空闲让进
- D.无限等待

8. 下列有关基于时间片的进程调度的叙述中，错误的是( )
- A. 时间片越短，进程切换的次数越多，系统开销越大
  - B. 当前进程的时间片用完后，该进程状态由执行态变为阻塞态
  - C. 时钟中断发生后，系统会修改当前进程在时间片内的剩余时间
  - D. 影响时间片大小的主要因素包括响应时间、系统开销和进程数量等
9. 用P、V操作实现进程同步，信号量的初值为( )。
- A. -1
  - B. 0
  - C. 1
  - D. 由用户确定

10. 下列关于管程的叙述中，错误的是( )。

- A. 管程只能用于实现进程的互斥
- B. 管程是由编程语言支持的进程同步机制
- C. 任何时候只能有一个进程在管程中执行
- D. 管程中定义的变量只能被管程内的过程访问

11. 在9个生产者、6个消费者共享容量为8的缓冲器的生产者-消费者问题中，互斥使用缓冲器的信号量初始值为( )。

- A. 1
- B. 6
- C. 8
- D. 9

12. 在操作系统中，死锁出现是指( )。

- A.计算机系统发生重大故障
- B.资源个数远远小于进程数
- C.若干进程因竞争资源而无限等待其他进程释放已占有的资源
- D.进程同时申请的资源数超过资源总数

13. 死锁的四个必要条件中，无法破坏的是( )。

- A.环路等待资源
- B.互斥使用资源
- C.占有且等待资源
- D.非抢夺式分配

14. 采用段式存储管理时，一个程序如何分段是在( )时决定的。

- A.分配主存
- B.用户编程
- C.装作业
- D.程序执行

15. 操作系统实现( )存储管理的代价最小。

- A. 分区
- B. 分页
- C. 分段
- D. 段页式

16. 当系统发生抖动时，可以采取的有效措施是( )。

- I. 撤销部分进程
  - II. 增加磁盘交换区的容量
  - III. 提高用户进程的优先级
- A. 仅I
  - B. 仅II
  - C. 仅III
  - D. 仅I、 II

17. 在磁盘中读取数据的下列时间中，影响最大的是( )

- A. 处理时间
- B. 延迟时间
- C. 传送时间
- D. 寻找时间

18. 在页式虚拟存储管理系统中，采用某些页面置换算法会出现Belady异常现象，即进程的缺页次数会随着分配给该进程的页框个数的增加而增加。下列算法中，可能出现Belady异常现象的是( )。

I. LRU 算法      II. FIFO 算法      III. OPT 算法

- A. 仅 II  
B. 仅 I、 II  
C. 仅 I、 III  
D. 仅 II、 III

19. 下列优化方法中，可以提高文件访问速度的是( )。

I 提前读    II. 为文件分配连续的簇    III. 延迟写    IV. 采用磁盘高速缓存

- A. 仅 I、 II  
B. 仅 II、 III  
C. 仅 I、 III、 IV  
D. I、 II、 III、 IV

20. 若文件f1的硬链接为f2，两个进程分别打开f1和f2，获得对应的文件描述符为fd1和fd2，则下列叙述中，正确的是( )。

- I.f1和f2的读写指针位置保持相同    II.f1和f2共享同一个内存索引结点
- III.fd1和fd2分别指向各自的用户打开文件表中的一项

A.仅III

B.仅II、 III

C.仅I、 II

D.I、 II和III

21. 位示图可用于( )。

A.文件目录的查找

B.磁盘空间的管理

C.主存空间的管理

D文件的保密

22. 若某文件系统索引结点(imode)中有直接地址项和间接地址项，则下列选项中，与单个文件长度无关的因素是()

- A. 索引结点的总数
- B. 间接地址索引的级数
- C. 地址项的个数
- D. 文件块大小

23. 在操作系统中，()指的是一种硬件机制。

- A. 通道技术
- B. 缓冲池
- C. SPOOLing技术
- D. 内存覆盖技术

24. 用户程序发出磁盘I/O请求后,系统的正确处理流程是( )。
- A. 用户程序→系统调用处理程序→中断处理程序-设备驱动程序
  - B. 用户程序→系统调用处理程序→设备驱动程序→中断处理程序
  - C. 用户程序→设备驱动程序→系统调用处理程序→中断处理程序
  - D. 用户程序→设备驱动程序→中断处理程序→系统调用处理程序
25. 在系统内存中设置磁盘缓冲区的主要目的是( )。
- A. 减少磁盘 I/O 次数
  - B. 减少平均寻道时间
  - C. 提高磁盘数据可靠性
  - D. 实现设备无关性

• 在一个磁盘上，有 1000 个柱面，编号为 0~999，用下面的算法计算为满足磁盘队列中的所有请求，磁盘臂必须移过的磁道的数目。假设最后服务的请求是在磁道 345 上，并且读写头正在朝磁道 0 移动。在按 FIFO 顺序排列的队列中包含了如下磁道上的请求：123, 874, 692, 475, 105, 376。

1) FIFO; 2) SSTF; 3) SCAN; 4) LOOK; 5) C-SCAN; 6) C-LOOK.

- 某个文件系统中，外存为硬盘。物理块大小为512B，有文件A包含598条记录，每条记录占255B，每个物理块放2条记录。文件A所在的目录如下图所示。文件目录采用多级树形目录结构，由根目录结点、作为目录文件的中间结点和作为信息文件的树叶组成，每个目录项占127B，每个物理块放4个目录项，根目录的第一块常驻内存。试问：
  - 1)若文件的物理结构采用链式存储方式，链指针地址占 2B，则要将文件 A 读入内存，至少需要存取几次硬盘？
  - 2)若文件为连续文件，则要读文件A的第487条记录至少要存取几次硬盘？
  - 3)一般为减少读盘次数，可采取什么措施，此时可减少几次存取操作？

- 某文件系统采用索引结点存放文件的属性和地址信息簇大小为 4KB。每个文件索引结点占 64B，有 11 个地址项，其中直接地址项 8 个，一级、二级和三级间接地址项各 1 个，每个地址项长度为 4B 请回答下列问题：
- 1) 该文件系统能支持的最大文件长度是多少？(给出计算表达式即可)
- 2) 文件系统用 1M( $1M=2^{20}$ ) 个簇存放文件索引结点，用 512M 个簇存放文件数据若一个图像文件的大小为 5600B，则该文件系统最多能存放多少个这样的图像文件？
- 3) 若文件 F1 的大小为 6KB，文件 F2 的大小为 40KB，则该文系统获取 F1 和 F2 最后一个簇的簇号需要的时间是否相同？为什么？