Memory Mapping

逻辑地址、线性地址和物理地址

Part1、确定逻辑地址

- 指令和全局变量是符号
  - 符号的地址是编译器和链接器，以及动态链接过程确定的

- 局部变量的地址是调用子程序时确定的

一、确定符号的逻辑地址

- 可执行程序引用的符号
  - 自己定义的符号
    - 地址由编译器根据编译规则确定
  - 共享库（动态链接库）中的符号
    - 地址由 编译器 和 ld 程序共同确定

静态链接过程 1
（确定应用程序自己定义的符号的地址）

- 虚地址空间首部空着，应用程序的代码段起始于一个固定的值imageBegin。
- 先排代码段、再排数据段。分配给所有逻辑段落的是整数个4K字节。
- 第一次扫描
  - 确定子程序地址：

    从imageBegin开始，排列源程序中的所有子程序。先是第一个源程序，然后第二个。。。第一个子程序的地址是imageBegin，第二个子程序的地址是imageBegin+length（length是第一个子程序的长度）。将所有子程序地址登记在符号表中。
  - 确定全局变量地址：

~

静态链接过程 2
（确定应用程序自己定义的符号的地址）

- 第二次扫描
  - 修改代码段中的每条指令
    - 应用程序自己定义的符号
      用已经确定的地址，取代符号
    - 共享库中定义的符号
      编译器无法确定它们的地址，填？

动态链接过程（确定共享库中符号的地址） 1

1. 编译器将指令中引用的共享库中的符号打？，登记在符号表里。
2. Linux系统规定，每个进程用户态代码和数据可以使用的逻辑地址空间是[0,3G]。
3. 若进程引用一个共享库，这个库的代码段、数据段和BSS段的长度分别为X，Y，Z字节。

ld在[0,3G]空间内找空闲的，可以容纳它们的连续空间，记起始地址为Xaddr、 Yaddr和Zaddr。这是ld在把共享库完全映射射到[Xaddr, Xaddr+X)， [Yaddr, Yaddr+Y) 。。。那么，共享库的第1个子程序逻辑地址是Xaddr，第2个子程序的逻辑地址是Xaddr+length， length是第一个子程序的长度。

得以确定，共享库中所有符号的逻辑地址~

动态链接过程 2

- ld修改可执行程序代码段中的指令
  - 根据可执行程序的符号表，找到程序引用的所有共享库中的符号，以及引用它们的所有指令。用上一步确定的逻辑地址取代？地址。
  - 递归修改共享库中被引用的函数，其中的？符号，代之以已确定的逻辑地址。
- 这就是动态链接过程：确定可执行程序运行所需的一切符号的逻辑地址

- 确定所有符号的逻辑地址后，动态链接的程序就可以跑了
- ld程序运行结束。进程 jmp至main函数的入口，开始执行应用程序

- Linux的段级映射：让所有段，起始地址是0；

  指令/变量的段内偏移量是　动态链接过程给出的逻辑地址

∴ Linux 系统中，指令和变量的线性地址等于它们的逻辑地址

二、确定局部变量的逻辑地址

- 局部变量的逻辑地址是进程运行过程中动态确定的
  - 内核确定堆栈的栈底。 ESP = 3G
  - gcc在每个子程序的头尾，插入用来构造栈帧、赋值栈帧定位指针EBP（当然还有ESP）的指令

  它们共同确定子程序局部变量的地址

- PS：CPU用 [EBP-] 和 [EBP+] 给出当前正在执行的那个子程序的局部变量和入口参数的逻辑地址。线性地址等于逻辑地址。

Part2、线性地址

- 线性地址是指令和变量在进程中的地址
- 线性地址空间中，装着某个进程需要执行的所有程序
  - 可执行程序
  - ld
  - 共享库（动态链接库）

  装在用户空间
  - Linux内核

  装在内核空间

- 线性地址空间的尺寸一般由机器的字长决定
  - 32位机，32位地址，线性地址空间尺寸：4G字节
    用户空间3G字节，内核空间1G字节。
  - 64位机（x86-64），48位地址，线性地址空间尺寸：2^48字节
    用户空间2^47字节，内核空间2^47字节。

Part3：Linux系统执行动态链接的（应用程序）的过程

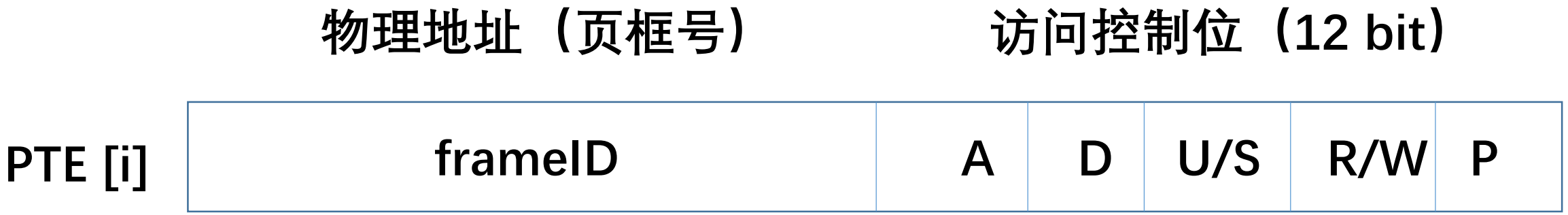
1. 分派1个进程，承担这个任务。将进程线性地址空间用户部分清空。
2. 为应用程序的代码段、数据段+BSS段分配线性地址空间
3. 为 ld 程序的代码段、数据段+BSS段分配线性地址空间
4. 这个新进程第一次上台运行，进用户态，开始跑。先跑ld。
   ld为应用程序include的所有共享库分配线性地址空间
   确定进程执行需要访问的所有共享库符号的地址
   修改代码段，将？改成确定的逻辑地址
5. ld一个远jmp，至应用程序main入口，进程开始跑应用程序

细节

Part4、线性地址映射为物理地址
——i386芯片的页级映射 和 Linux的页表

解读页表项：　　PTE（Page Table Entry）

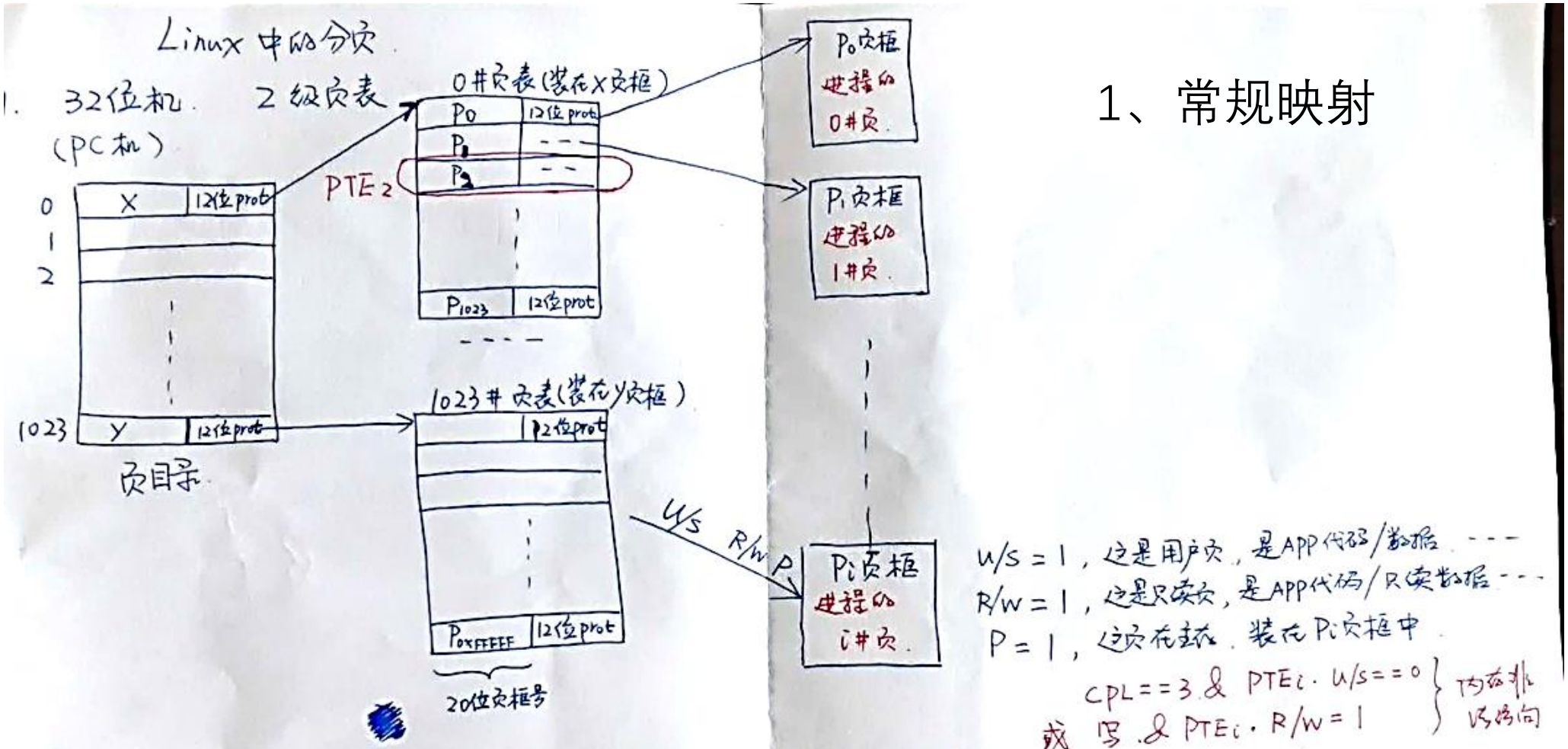
PA进程



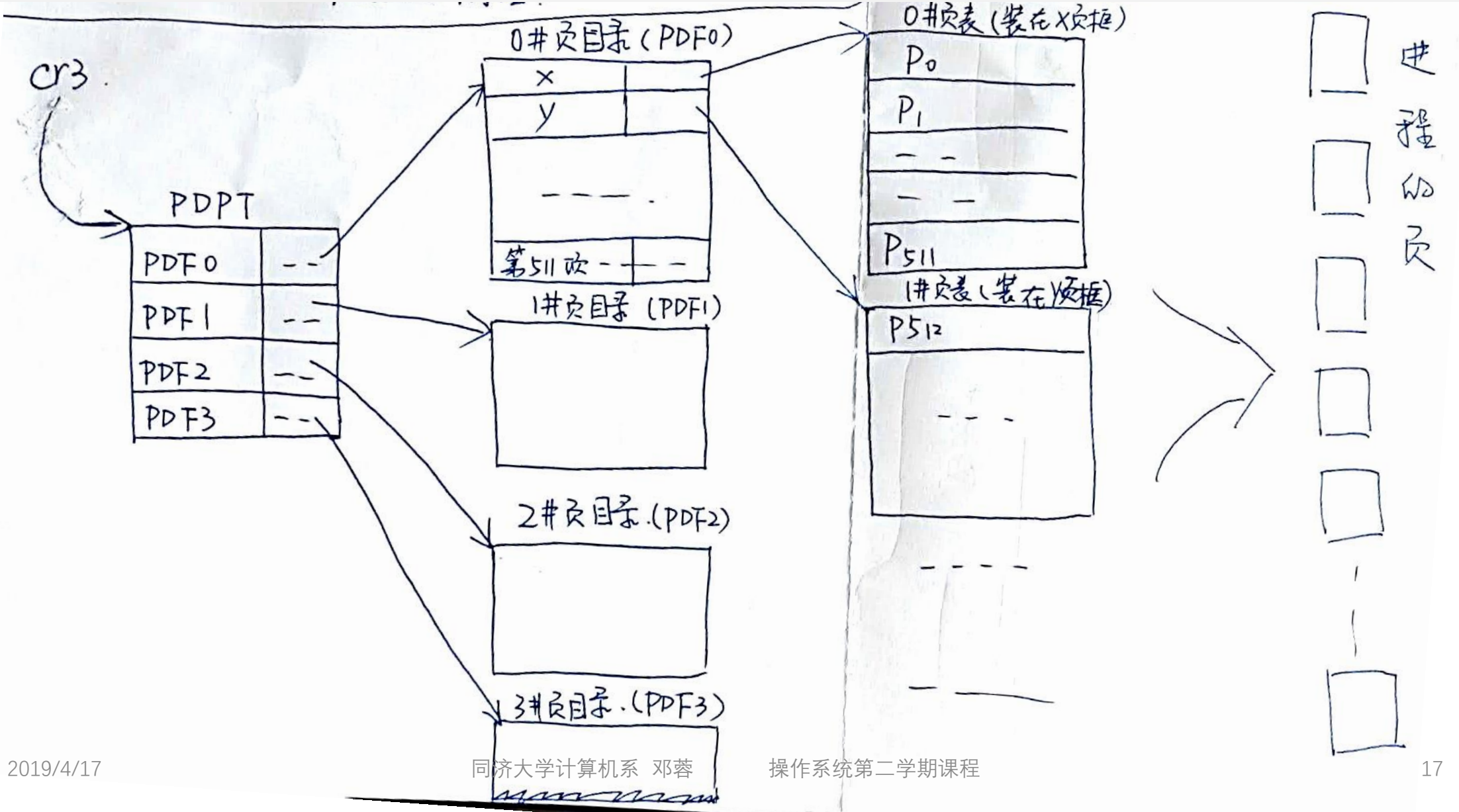
1、常规映射

- 32位系统。32位逻辑地址。
- PC机，32根内存总线。配置的内存条容量：4G字节以下
- 32位逻辑地址映射至32位物理地址

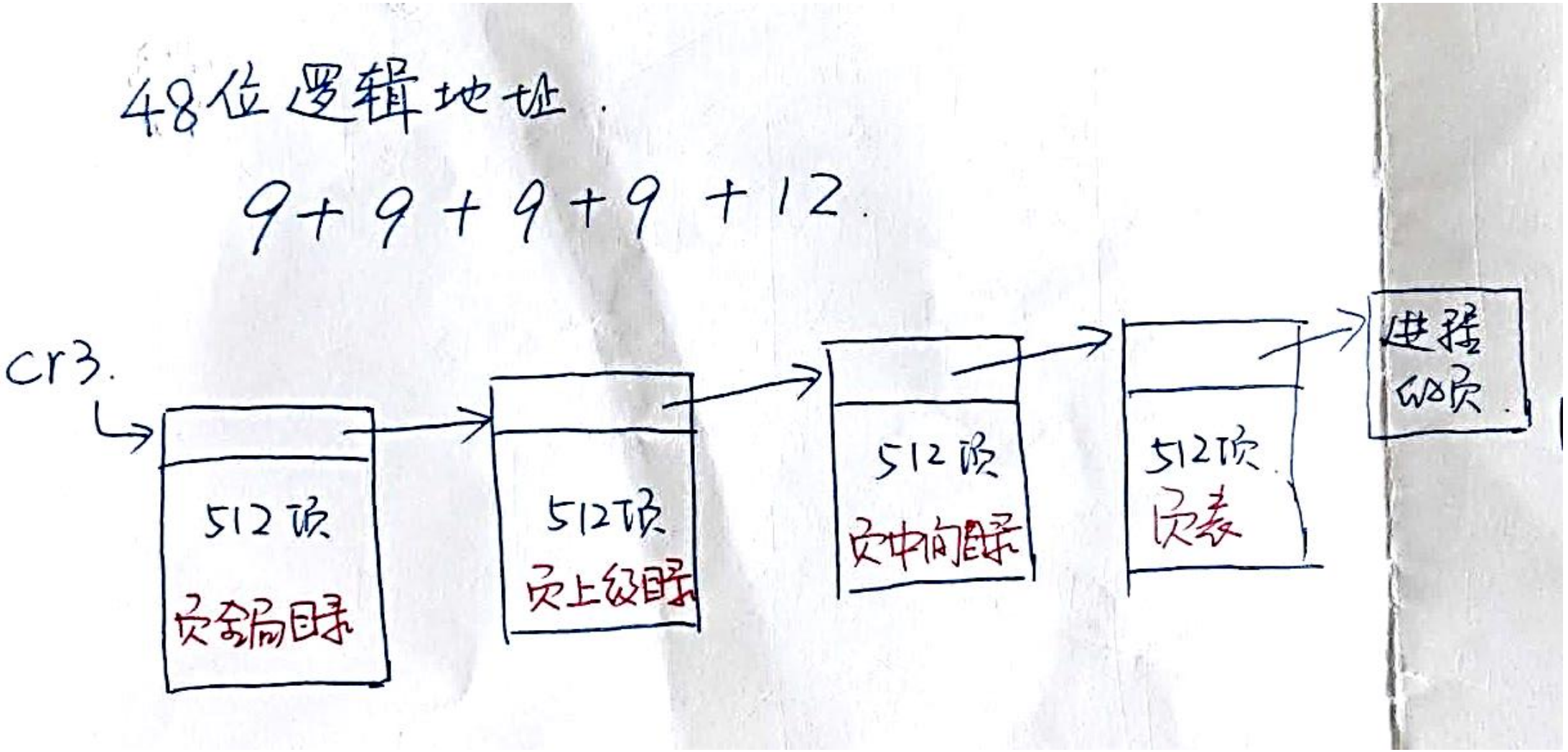
1、常规映射


Linux 中的分页
1. 32位机 2级页表
(PC机)
0
1
2
1023
X
Y
12位prot
页目录
0#页表(装在X页框)
PTE2
P0
P1
P2
P1023
1023#页表(装在Y页框)
P0xFFFFF
20位页框号
P0页框
进程的
0#页
P1页框
进程的
1#页
Pi页框
进程的
i#页
U/S
R/W
P
U/S = 1，这是用户页，是APP代码/数据 ---
R/W = 1，这是R页，是APP代码/R页数据 ---
P = 1，这页在主存，装在Pi页框中
CPL==3 & PTEi.U/S==0
或 写 & PTEi.R/W=1
内存非法访问
PTEi P=1，这个页面在内存中，装在Pi页框中
P=0 本进程在在i#页，但这个页还未占用主存页框
本进程不存在i#页。
① 需要追加一页堆栈
② 不是这种情况
缺页
内存非法访问
禁

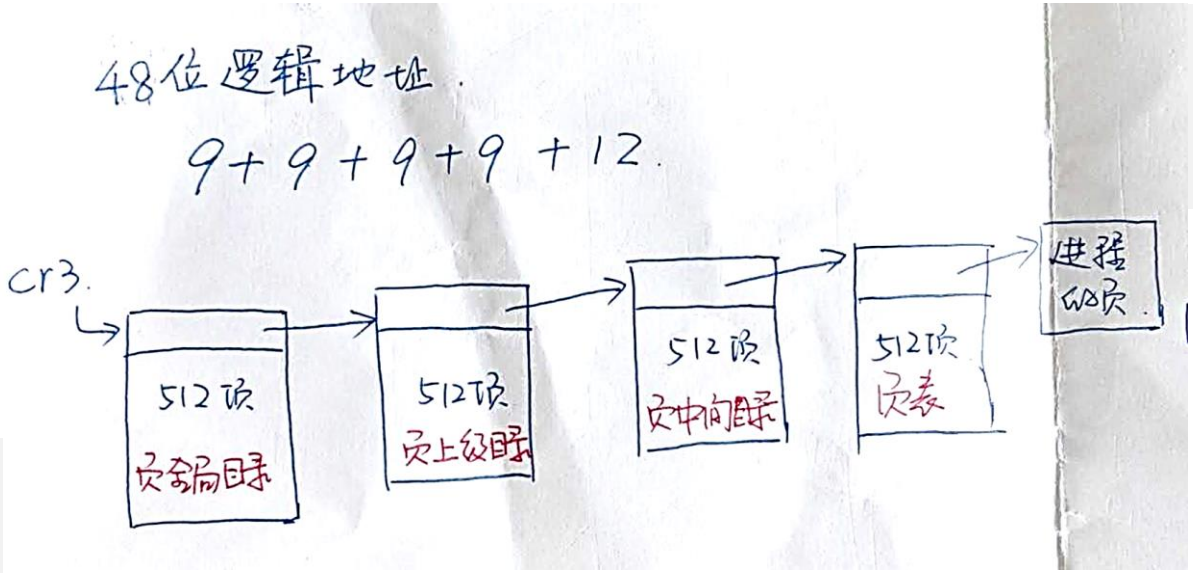
2、32位机　使能 PAE　　2+9+9+12

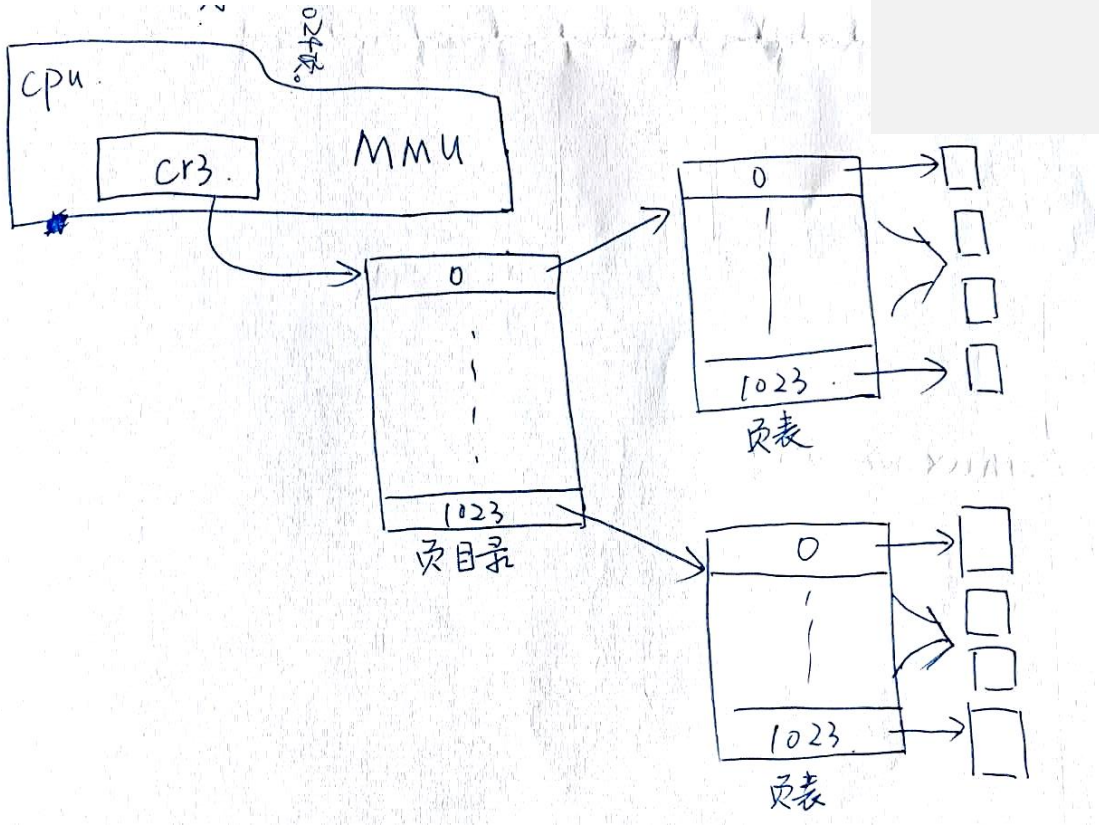


3、64位机　使能 PAE


48位逻辑地址.
9+9+9+9+12.
cr3.
512项
页全局目录
512项
页上级目录
512项
页中间目录
512项
页表
进程
4K页.

4、Linux通用页表模型





• 宏	x86-64	80x86
PGDIR_SHIFT	39	22
PUD_SHIFT	30	22
PMD_SHIFT	21	22
PAGE_SHIFT	12	12

取线性地址的高位（从47bit到PGDIR_SHIFT），查页全局目录，得PGD。

比较 PGDIR_SHIFT 和 PUD_SHIFT，

- 相等，页全局目录就是继续映射需要的页上级目录。
- 不相等，PGD.address指向的页框，装有继续映射需要的页上级目录。

Part5、硬件缓存（CPU中的cache）

1. 微处理器时钟频率几个GHz，而动态RAM（DRAM）的存取时间是时钟周期的数百倍。

↓

CPU在取内存变量或取下条指令，会等待很长时间。

2. 局部性原理. locality principle.

局部性原理适用于

程序结构.(指令)　　循环结构

数据结构(变量).　　数据经常组织成数组.

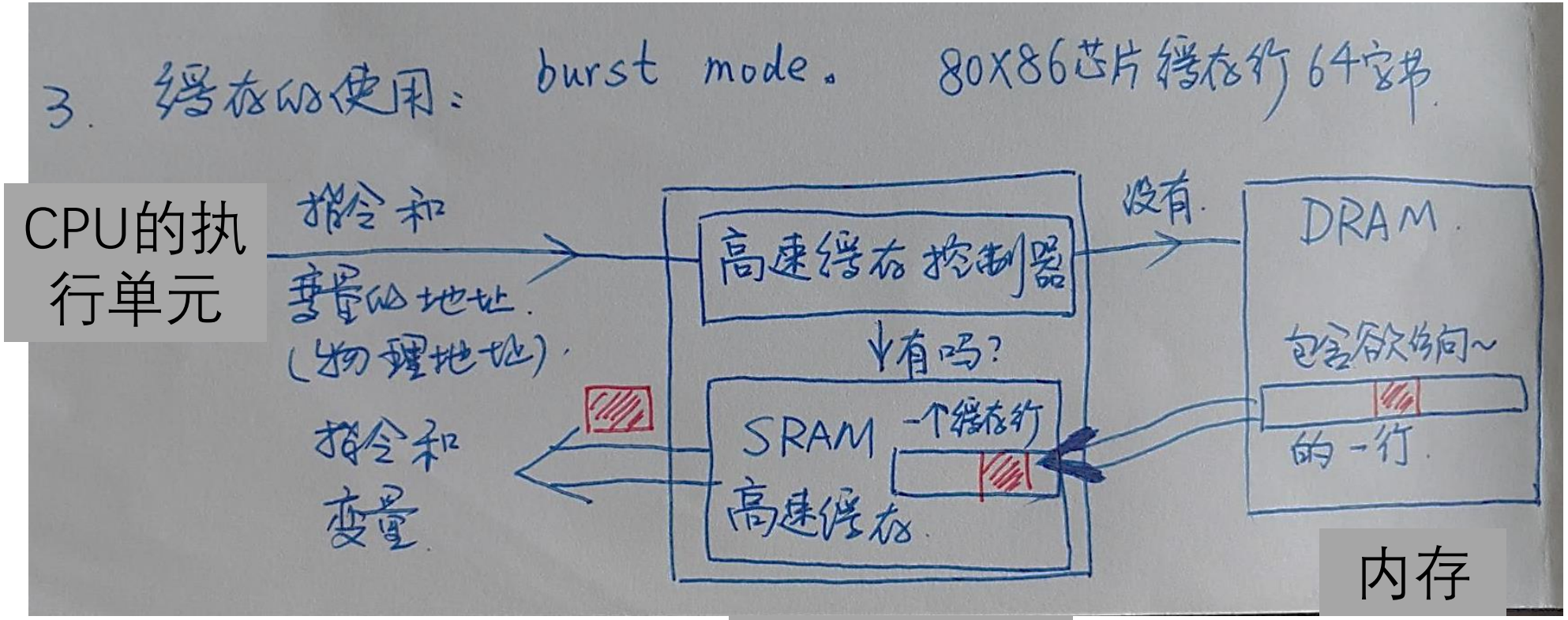
局部性原理 ⟹ 内存中最近常用的指令和数据.

未来，又被用到的可能性很大.

∴，把它们装进缓存.

缓存中 cpu 内部的 SRAM.

cpu 访问 SRAM，存取时间比 DRAM 快许多.



CPU的缓存包括2个元件：

- SRAM，用来缓存内存数据
- 高速缓存控制器，从SRAM或者DRAM中读写内存变量 & SRAM空间管理

4. [redacted] 缓存[redacted] 组织.

① SRAM是缓存行[redacted]集合.

② 每一行[redacted]缓存 内存指令(变量). 64字节.

③ 缓存行和内存单元的映射关系

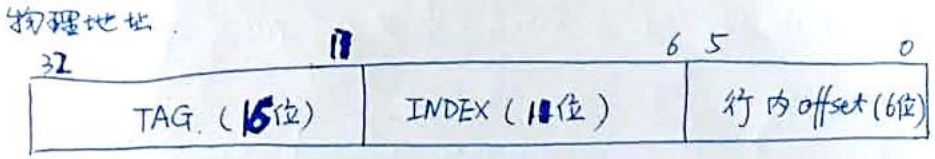
a. 直接映射 (direct mapped).
每个内存单元只能装在一个固定的缓存行.

b. 全关联 (fully associative)
每个内存单元可以装在SRAM中, 任意的缓存行

C. N路关联 (N-way set associative).
每个内存单元可以装在SRAM中, 固定的N个缓存行中的任一个缓存行.

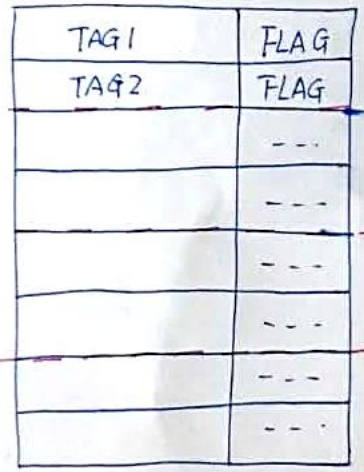
5. 缓存结构

假设缓存行长64字节，SRAM可存放4K个缓存行，2路关联。

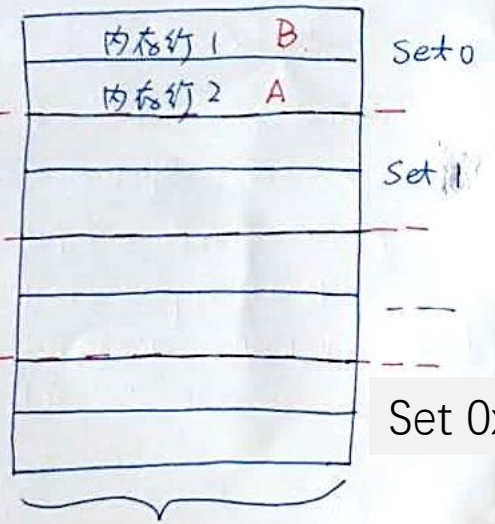


缓存结构

高速缓存控制器



SRAM



装着一张表，登记SRAM中都装着哪些内存行

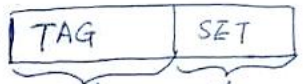
64字节

装着缓存着的内存指令和变量

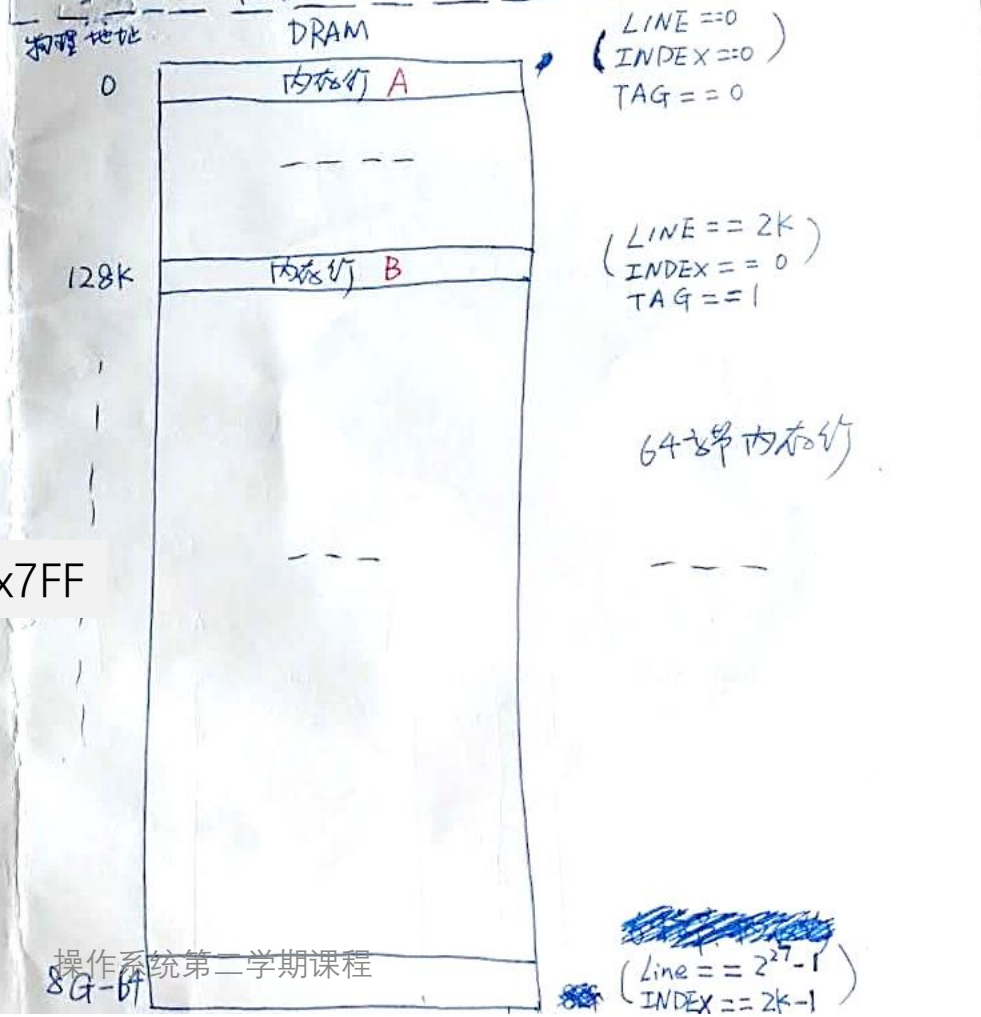
DRAM中，INDEX == i 的所在行，可以存放在 set = i 的集合。

每个set可以装2个内存行，这2个内存行的行号LINE，由TAG和SET共同决定

即 缓存的行号 ==



例：图中 TAG1 == 1
TAG2 == 0



6. 缓存访问.

取物理地址中的INDEX，确定set.

读高速缓存物理块中的表，找set对应的那2个表项。

取物理地址中的TAG，这个值

↳ TAG1或TAG2相等吗？ Y. 命中，读SRAM中数据

N. 不命中，

将DRAM中的内存行

读入set ⓐ

读入SRAM中的内存行

ⓐ 如果set中有空行，直接读入内存行

------没有空行，就会覆盖一个缓存行。把其中原有数据淘汰。如何选择被覆盖的缓存行？

LRU. 成本高.

Random. 实际采用的

7. 读操作和写操作

缓存命中，直接从SRAM中读数据

写操作，改变的是SRAM中的数据，需要考虑的问题是内存中的这个数据什么时候改？

① 写上：写穿透（write_through）

② 过一会：写回（write_back）

单cpu，写回时机：执行将数据刷回内存的指令

被修改的缓存行被淘汰

多处理器系统，缓存要同步，使用额外的硬件电路

每颗cpu有自己的L1 cache

改动L1 cache中的一个缓存行，硬件电路必须通知其它cpu，修改缓存里的这个缓存行

cache snooping

8. 缓存控制.

① Cr0 CD 标志启用/禁用高速缓存电路.
NW -- 指明缓存回写还是马穿透.

② Pentium允许内核为每一个页框指定不同的缓存策略.

每个PTE都有 { PCD. 缓存吗?
PWT. 马回?写穿透?

Linux为了代码的通用性
清除PCD和PWT。
⇒所有页缓存&写回策略

9. TLB.

① TLB 缓存最近使用过的 PTE。

每个 cpu 有自己的 TLB。

不同的 cpu 之间 TLB 不用同步。因为不同 cpu 跑不同的进程，相同的逻辑地址映射到不同的物理页框。

② 当 cpu 的 cr3 控制寄存器被修改时，硬件无效所有 TLB 表项。这是因为启用新页表，TLB 中所有缓存的 PTE 包含的是旧的映射关系。对内存中的新页表而言，它们全是脏数据。

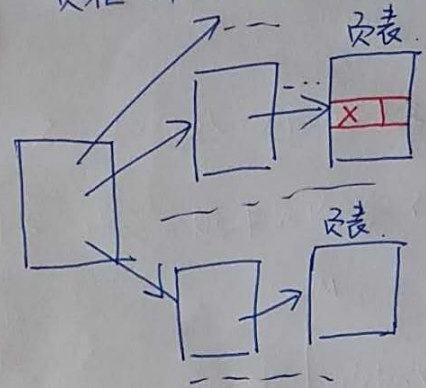
结论：启用新页表一定要 very、very、very lazy!

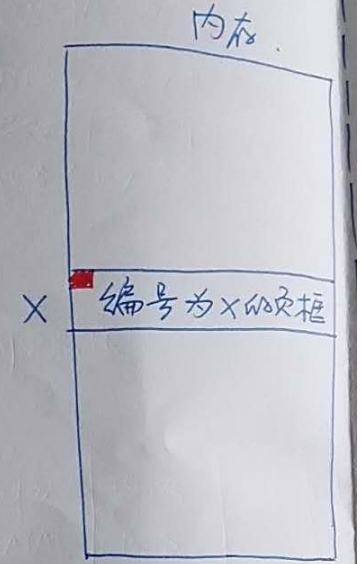
Part6、Linux物理内存映射

基础知识

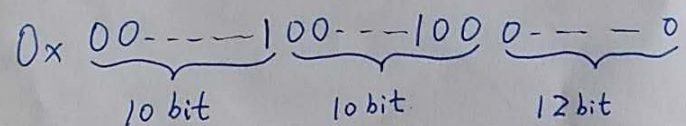
一、

无论是内核，还是APP，想要使用页框X中的代码和数据，必须

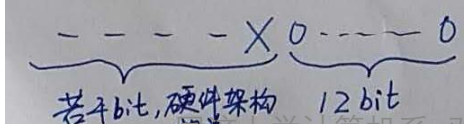




若突出的页表是1#页表，PTE是4# PTE，访问目标单元，内核给出的[redacted]线性地址应该是



MMU映射后，得物理地址



一、内核保留一些页框，剩下的页框动态分配

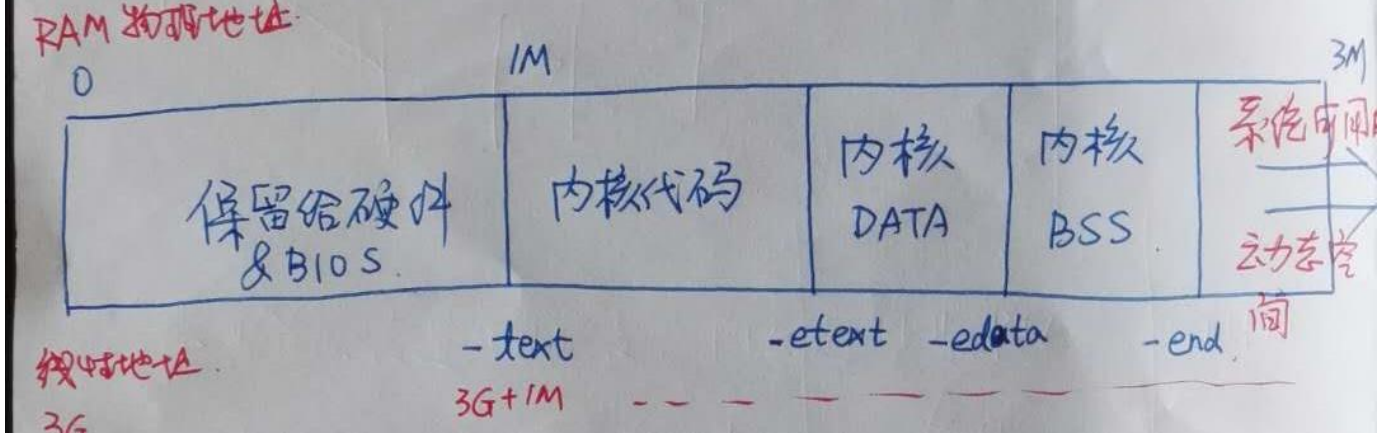
1. 保留的页框里存放着 BIOS代码和数据

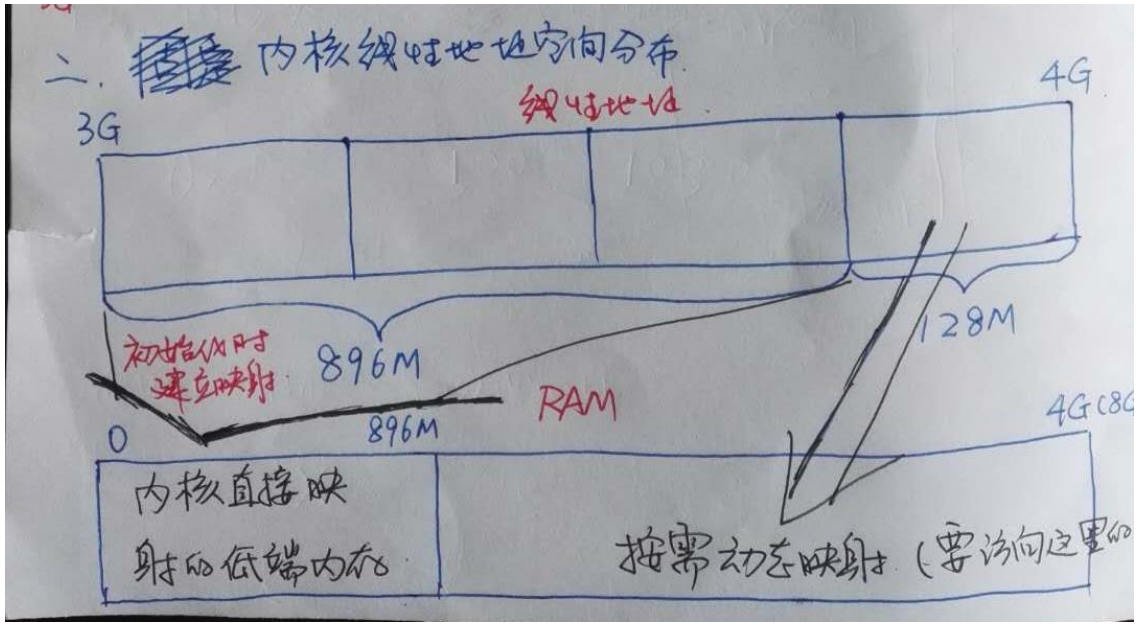
~~IO硬件设备所需内存~~

IO设备的内存

2. 内存代码和全局变量

PS：保留页框中的内容不可交换到磁盘上





二. swapper_pg_dir 变量

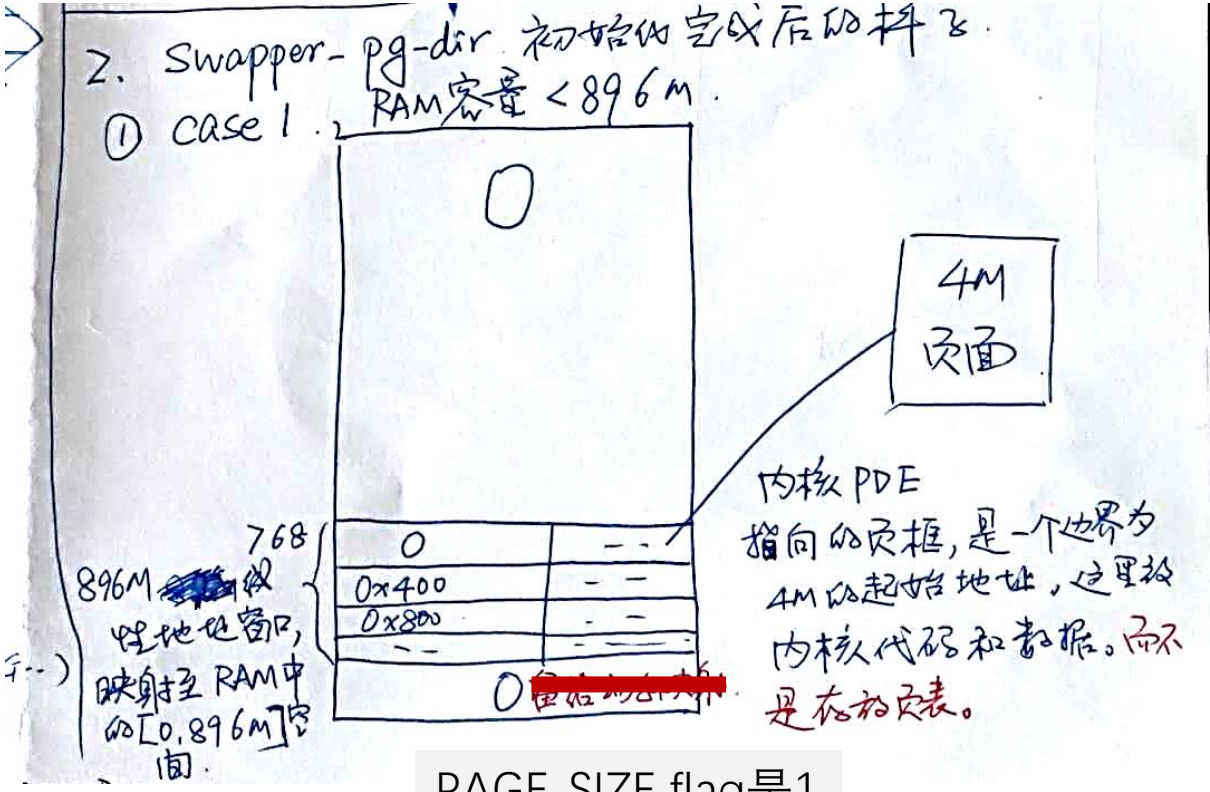
1. 主内核页全局目录

系统初始化时，用swapper_pg_dir管理的页表系统进行映射。随后，swapper_pg_dir不用做映射。但页目录的后面256项建立的映射关系是系统中所有进程映射内核空间所需的参考模型。

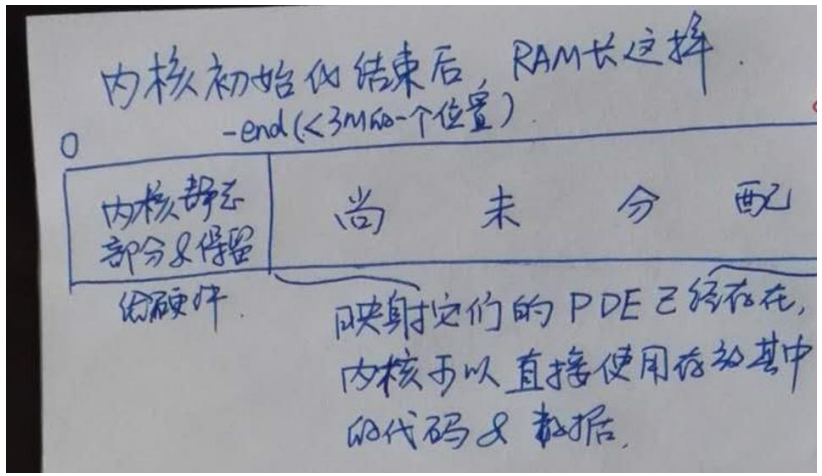
也就是：swapper_pg_dir中，表项改动了，进程用来映射的核心态页表的表项也就改动了。如何做到改动传递下去，chapter 8有说。

2. swapper_pg_dir初始化完成后的样子

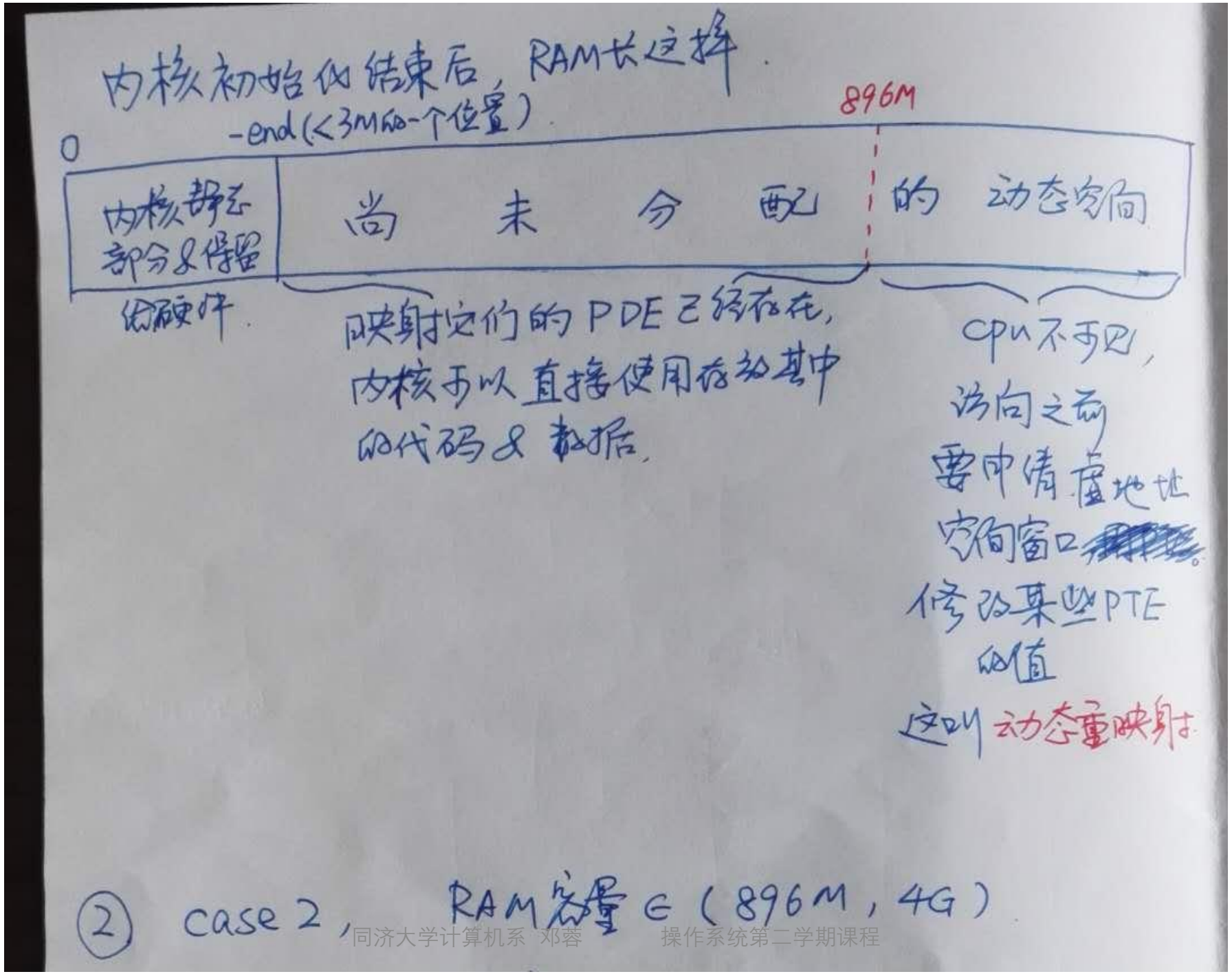
① case 1. RAM容量 < 896M



PAGE_SIZE flag是1



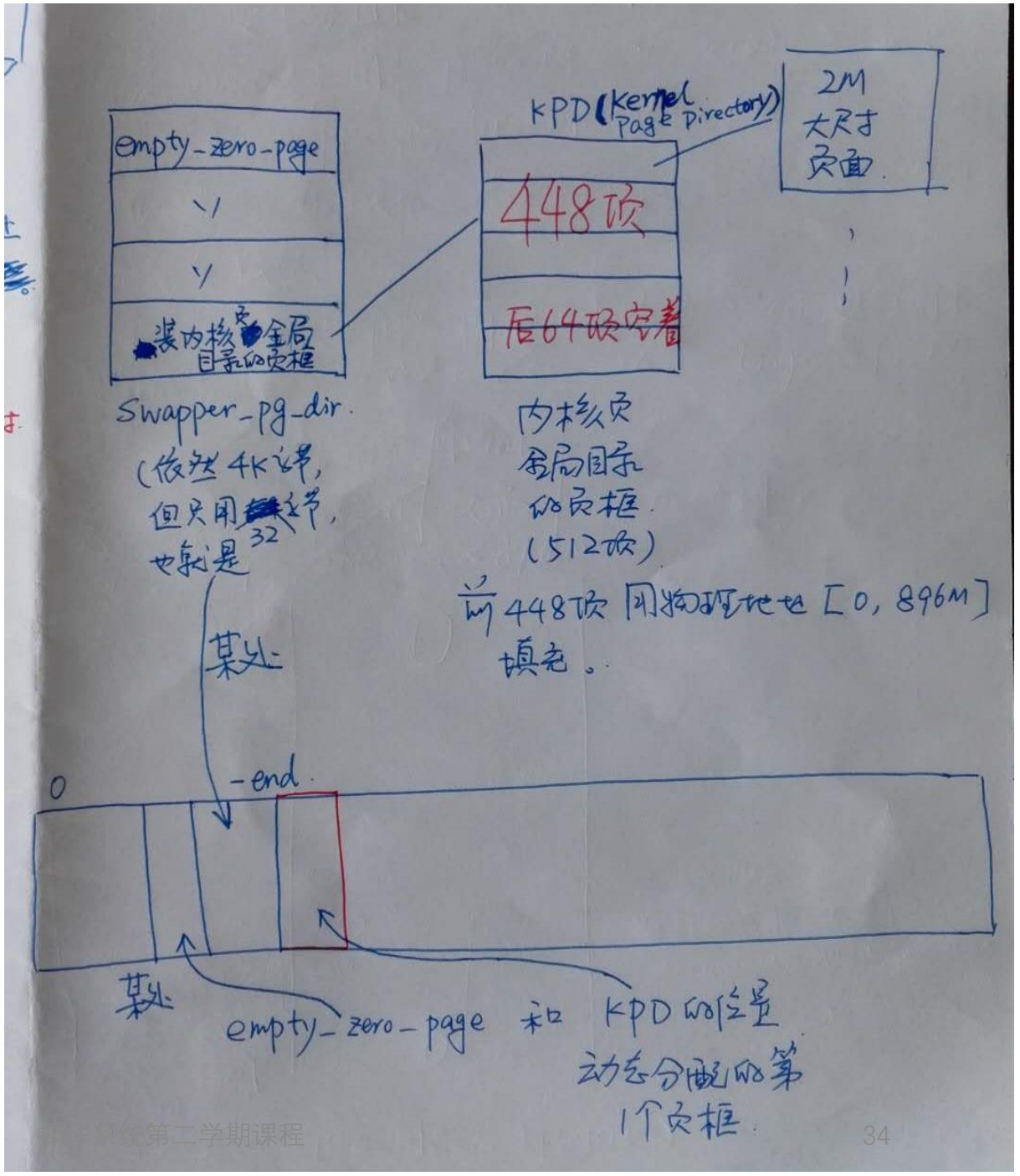
内存总容量


内核初始化结束后，RAM长这样
-end(<3M的一个位置)
896M
0
内核部分
部分保留
尚未分配
的动态空间
物理内存
映射它们的PDE已经存在，
内核可以直接使用存放其中
的代码&数据
CPU不可见，
访问之前
要申请虚地址
空间窗口
修改某些PTE
的值
这叫动态重映射
② case 2，RAM容量∈(896M，4G)

③ case3，RAM容量 > 4G （使能PAE，内核编译时支持PAE）

也就是内核页表需能够把32位逻辑地址映射到36位物理地址，也就是初始化时，内核对主内核页全局目录需用三个分页模型

Swapper_pg_dir 长成这样



活动页表集

- 活动页表集，装着进程执行的应用程序的代码和数据

- 普通进程的页表有自己的活动页表集
- 内核线程的页表没有自己的活动页表集

- 进程切换的时候，
  - 普通进程A→另一个普通进程B，CPU切换页表。
    方法：B的页目录起始地址赋值给cr3。
  - 普通进程→内核线程，CPU无需切换页表。

三. 非直接映射的后 128M 线性地址空间

1. 用途.

实现非连续内存分配 和 固定映射 的线性地址。

2. 直接映射的低端内存，线性地址 VA 和物理地址 PA 之间的对应关系是：

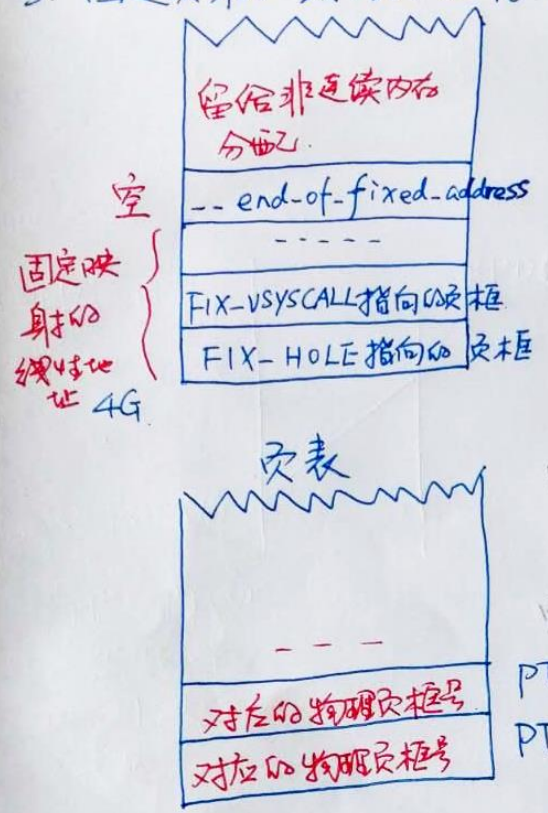
$$VA = PA + 0xC0000000$$

固定映射的内存，线性地址 VA 映射到 物理地址 PA。

但系统中固定映射的线性地址是有限的，登记在一张编译器用的表里.

```
enum fixed_addresses {
    FIX_HOLE,                   // 0
    FIX_VSYSCALL,               // 1
    FIX_APIC_BASE,              // 2
    FIX_IO_APIC_BASE_0,         // 3
    ......                      ; 是常数 index
    __end_of_fixed_addresses
};
```

3. 固定映射的线性地址在 4G 空间的高端



虚地址空间，每个格子 4K 字节.

每个 PTE 对应的物理框号是固定的，有 IO 用的地址——它们，编译时，就确定了。

PTE[0xFFFFF - FIX_VSYSCALL]

PTE[0xFFFFF - FIX_HOLE]

宏 Set_fixmap(idx, phys) 写 PTE

宏 fix_to_virt(idx) 计算 idx 对应的线性地址

常数 FIX_XXX 是 idx

PS. 固定映射和非连续内存分配在第8章还会说。

Part7、Linux硬件高速缓存和TLB处理

- 主题：内核如何利用硬件高速缓存提高系统性能
- 策略：减少高速缓存和TLB未命中次数

高速缓存

1. 提高 cache 命中率
   ① 用宏L1-CACHE-BYTES取高速缓存行的大小
   ② 内核定义数据结构的时候，将最常用的字段放在前面
   ③ 为大数据结构分配空间的时候，尽力让所有的cache行使用频率和使用模式完全相同。就是，不出现某些硬件cache行上经常出现淘汰的情况，而另一些cache行得不到使用的情况
2. 缓存同步
   ① 保证某个内存变量的值，在所有CPU 核心 cache 里的值是相等的
   ② CPU硬件自动处理缓存同步，所以Linux内核不用操心，去刷硬件高速缓存。

TLB

1. 在TLB中，有2种表项
   1. 全局的　　这是映射内核的PTE
   2. 局部的　　这是映射现运行进程执行的APP代码和数据的PTE。

   PS：发生进程切换的时候，所有局部的PTE表项都没有用了。内核需要发命令刷所有局部TLB表项。

2. 处理器不能同步他们自己的TLB缓存。
3. 内核同步TLB的手段：　无效某个TLB表项中的PTE。

   为什么需要将一些TLB表项置为无效呢？　以为，内核改写的PTE在内存中。
   TLB中缓存的，是相应PTE改写之前的值。

4. 策略：TLB，能不刷新就不刷新，能小范围刷新就小范围刷新。

单 CPU 系统中的 TLB，无效的时机和范围

1. 重写内核页表

   所有TLB表项都是无效的。

2. 缺页中断

   无效TLB表项

3. 创建子进程

   刷新与子进程使用的mm有关的所有TLB。

4. 进程切换

   原则上 刷新全部过期页表

   可以不刷新的情况： PA→PB。 两个进程使用相同的页表

   PA→kernel thread。

5. 为某个用户态进程PA分配页框，并将页框物理地址填入PTE。

与之对应的TLB表项要刷新。

若，有其它CPU上的进程使用相同的页表集，向对方发送处理器间中断，通知对方CPU刷新相关所有TLB。

可以优化，对方如果跑内核线程，TLB不用刷新。

这是因为，内核线程运行结束后，接下去执行的进程非常可能不再和PA使用相同的页表集。

内核懒惰TLB模式。

问题的关键：内核代码修改的是内存中的PTE。TLB中的复本，是旧的值