

计算机系统基础
Programming Assignment

PA 3 存储管理

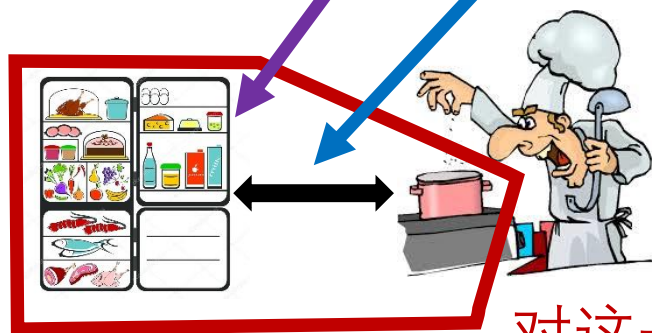
——PA 3-3 分页机制

2017年12月15日

前情提要

PA 3-2 管理太混乱，引入了分段机制，
代码段、数据段、栈段

PA 3-1 食材摆太远，取用太耗时，引入Cache



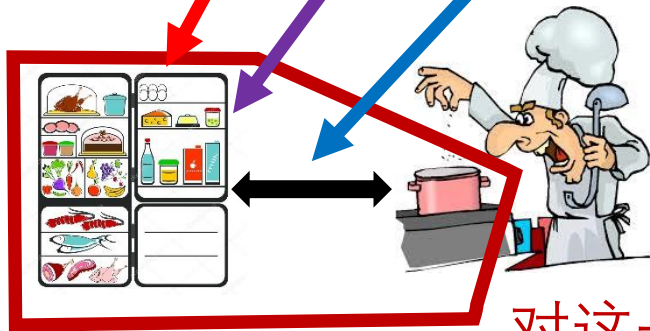
对这一块有点不满意

PA 3-3 分页机制的动机

PA 3-3 主存太小，多进程管理困难，
引入分页机制

PA 3-2 管理太混乱，引入了分段机制，
代码段、数据段、栈段

PA 3-1 食材摆太远，取用太耗时，引入Cache



对这一块有点不满意

分段机制回顾

- PA 3-2 实现分段机制后NEMU的地址转换过程

```
uint32_t vaddr_read(vaddr_t vaddr, uint8_t sreg, size_t len) {  
    assert(len == 1 || len == 2 || len == 4);  
    #ifndef IA32_SEG  
        return laddr_read(vaddr, len);  
    #else  
        uint32_t laddr = vaddr;  
        if(cpu.cr0.pe == 1) {  
            laddr = segment_translate(vaddr, sreg);  
        }  
        return laddr_read(laddr, len);  
    #endif  
}
```

做段级地址转换：
逻辑地址 -> 线性地址

线性地址直接作为物理地址使用

```
uint32_t laddr_read(laddr_t laddr, size_t len) {  
    return paddr_read(laddr, len);  
}
```

分段机制回顾

- 分段机制有什么优点？
 - 提供了保护机制（可以做权限和越界检查）
 - 扩展了可寻址的空间（考虑不同的段基地址）
 - 将不同类型的数据（代码、栈、数据）分开管理
- NEMU工作在ring 0的扁平模式
 - 虽然以上的优点都没有直接的体现
 - 但是充分发挥了教育意义

分段机制回顾

- 分段机制有什么局限性？
 - 物理内存大小的限制

y=线性地址

x=有效地址

$$y = \text{seg}(x) = \text{seg.base} + x < \text{HWADDR_MAX}$$

在分段机制下，可寻址的空间无法突破物理内存大小的限制

分段机制回顾

- 分段机制有什么局限性？
 - 物理内存大小的限制

y=线性地址

x=有效地址

$$y = \text{seg}(x) = \text{seg.base} + x < \text{HWADDR_MAX}$$

在分段机制下，可寻址的空间无法突破物理内存大小的限制

在“扁平模式”下（段基址设为0，大小设为最大），可寻址的最大空间为？

分段机制回顾

- 分段机制有什么局限性？
 - 物理内存大小的限制

y=线性地址

x=有效地址

$$y = \text{seg}(x) = \text{seg.base} + x < \text{HWADDR_MAX}$$

在分段机制下，可寻址的空间无法突破物理内存大小的限制

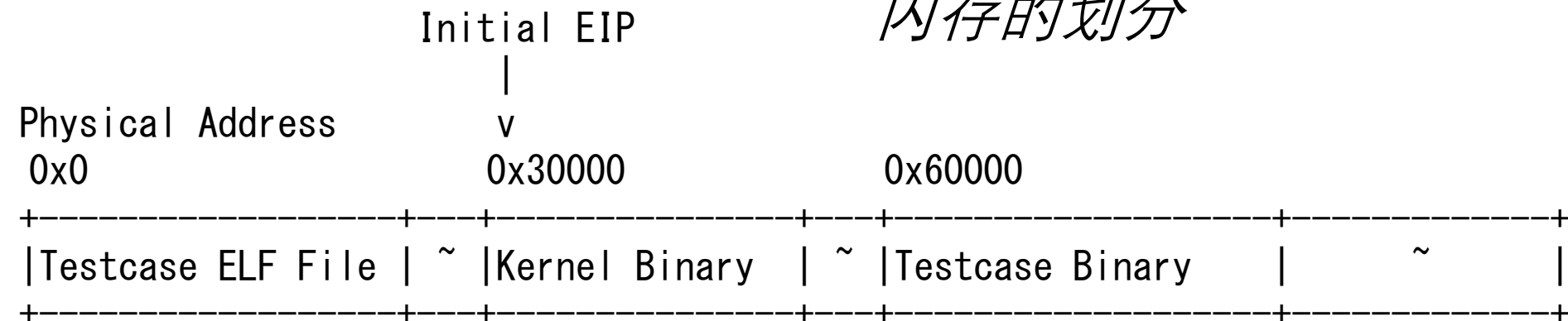
$$2^{32} \text{ B} = 4\text{GB}$$

在“扁平模式”下（段基址设为0，大小设为最大），可寻址的最大空间为？

分段机制回顾

- 分段机制有什么局限性？
 - 物理内存大小的限制
 - 多进程并行

*Kernel和Testcase并存时
内存的划分*

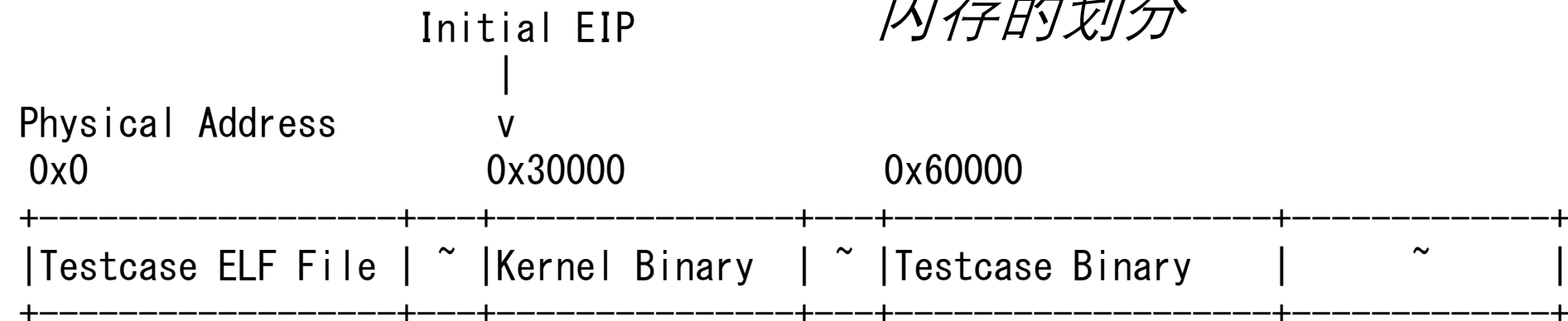


我们知道Kernel占多少字节，通过修改testcase/Makefile设置-Ttext为0x60000来避免内存区域的冲突（事实上0x60000也并不是一个安全的值）。

分段机制回顾

- 分段机制有什么局限性？
 - 物理内存大小的限制
 - 多进程并行

*Kernel和Testcase并存时
内存的划分*



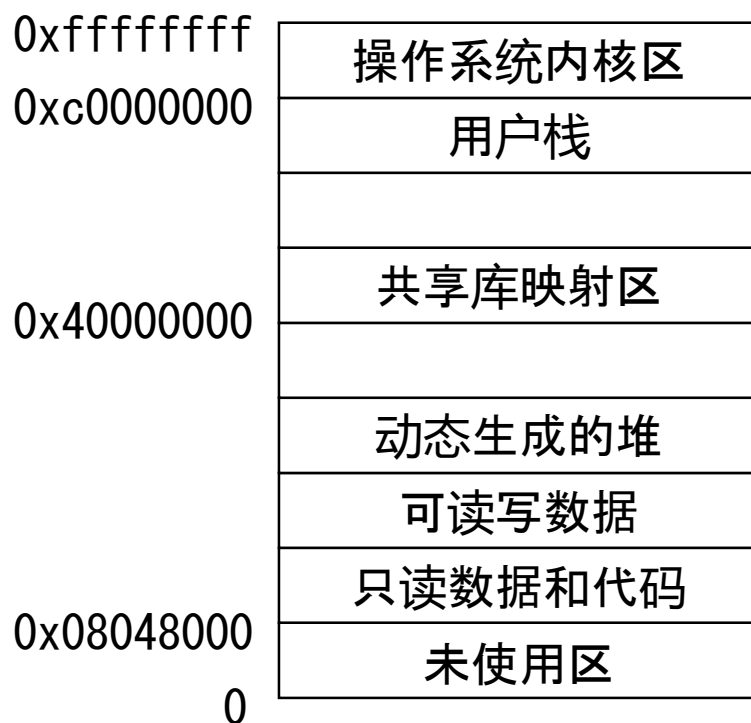
如果同时存在多个testcase要同时运行，且装载顺序在运行时动态确定，就没办法在编译时就确定各个进程内存起始地址，管理起来非常麻烦。

解决方法：分页机制

1. 每个进程有自己独占的虚拟地址空间
2. 虚拟地址空间和物理地址之间以“页”为单位对应
3. 主存中放不下的“页”放到磁盘上去

分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间



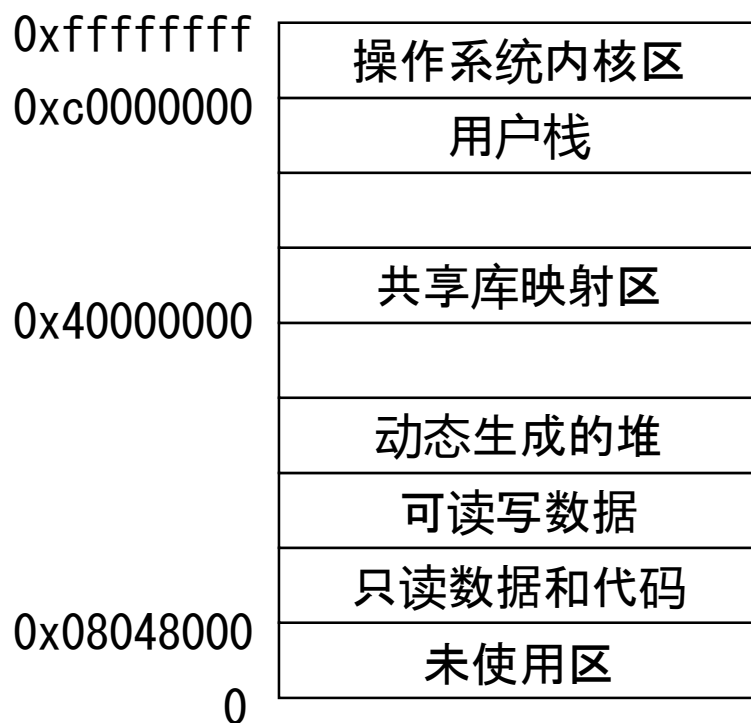
Linux中一个进程的虚拟地址空间

每个进程独占0x0 - 0xffffffff

类比：对餐厅的每个客户而言，
整个冰箱好像都归一个人用

分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间



Linux中一个进程的虚拟地址空间

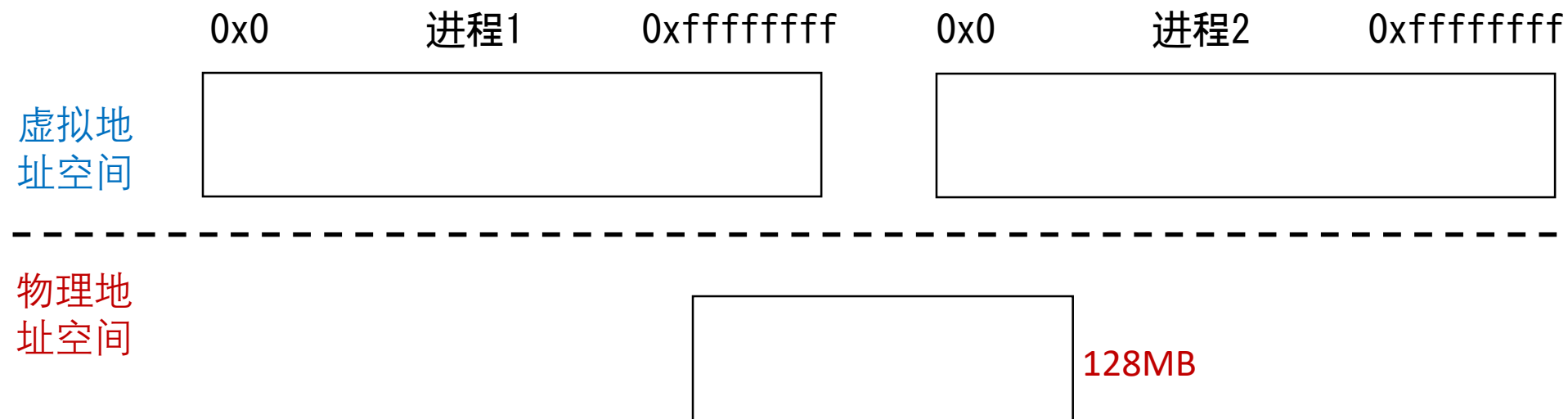
每个进程独占0x0 - 0xffffffff

类比：对餐厅的每个客户而言，
整个冰箱好像都归一个人用

程序员无需考虑运行时内存的分配情况，只需针对虚拟地址空间进行编译就可以了

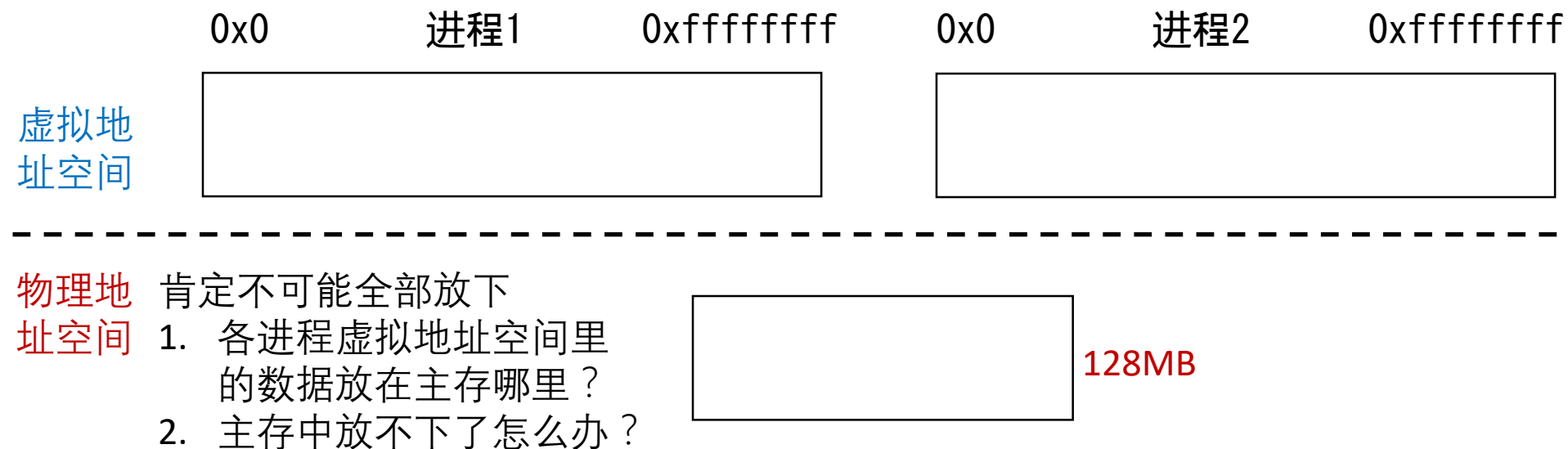
分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间



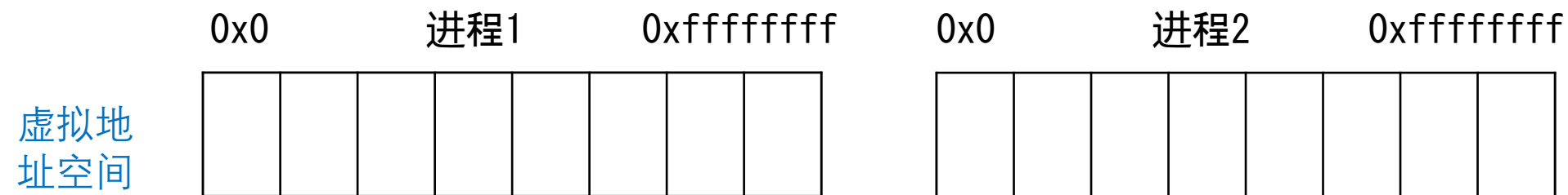
分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间



分页机制 - 原理

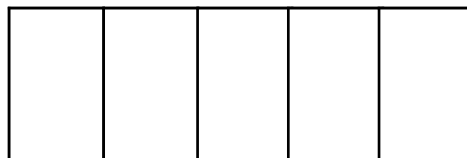
- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为**每个进程**提供了一个独立的、极大的虚拟地址空间



物理地址空间 肯定不可能全部放下

1. 各进程虚拟地址空间里的数据放在主存哪里？

2. 主存中放不下了怎么办？

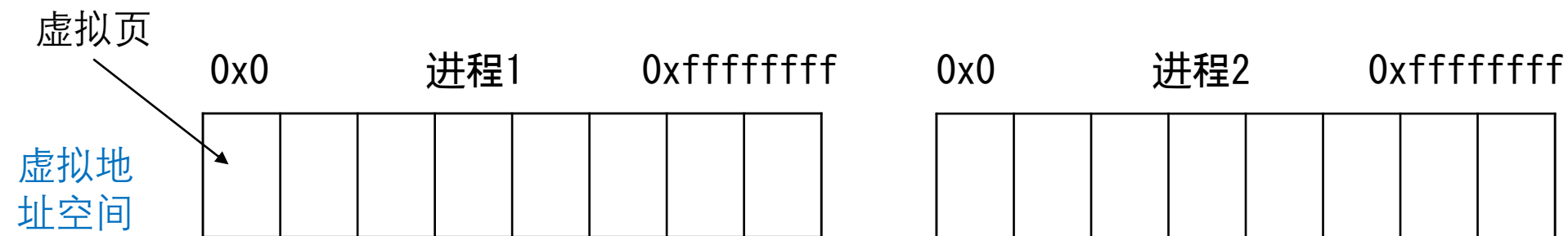


128MB

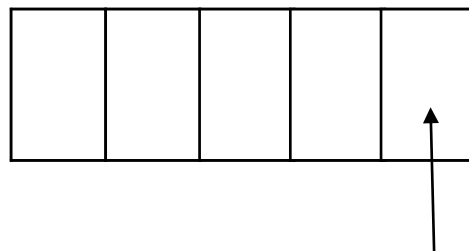
1) 将虚拟地址划分成相同大小的单元（页）

分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为**每个进程**提供了一个独立的、极大的虚拟地址空间



- 物理地址空间
- 肯定不可能全部放下
1. 各进程虚拟地址空间里的数据放在主存哪里？
 2. 主存中放不下了怎么办？

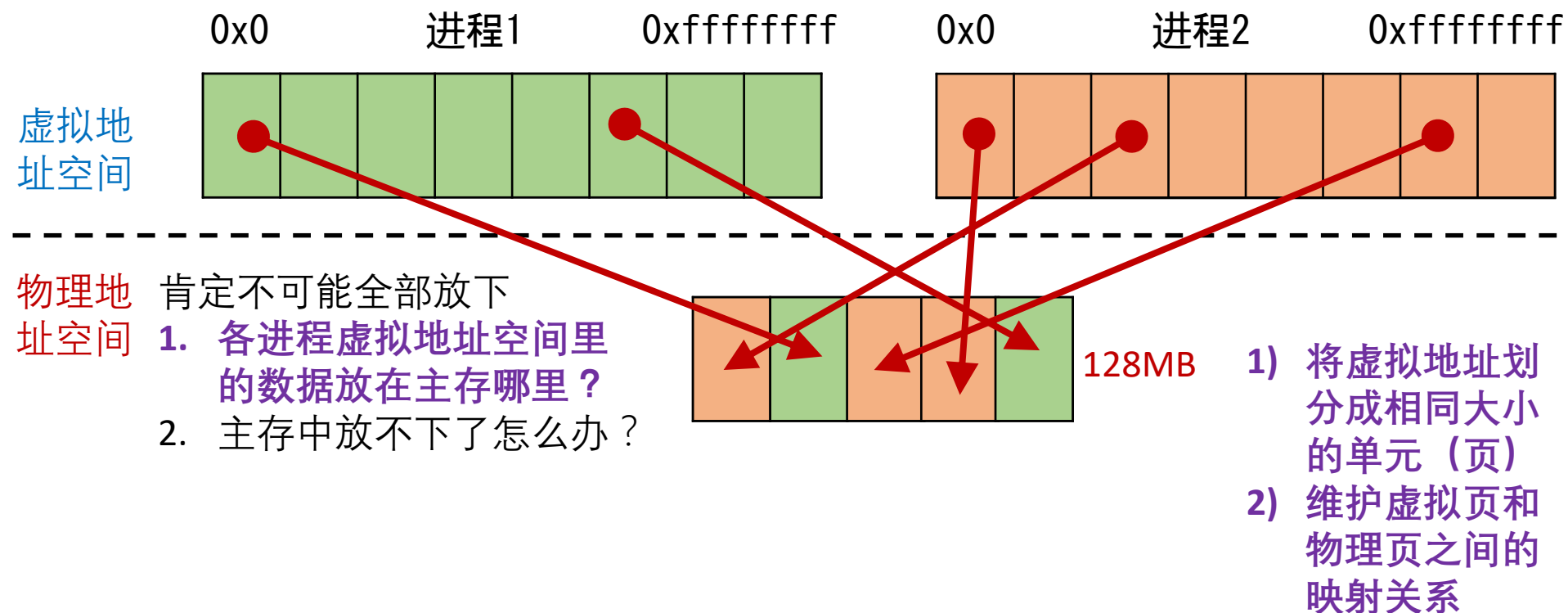


物理页（页框），约定一个页的大小为4KB

- 1) 将虚拟地址划分成相同大小的单元（页）

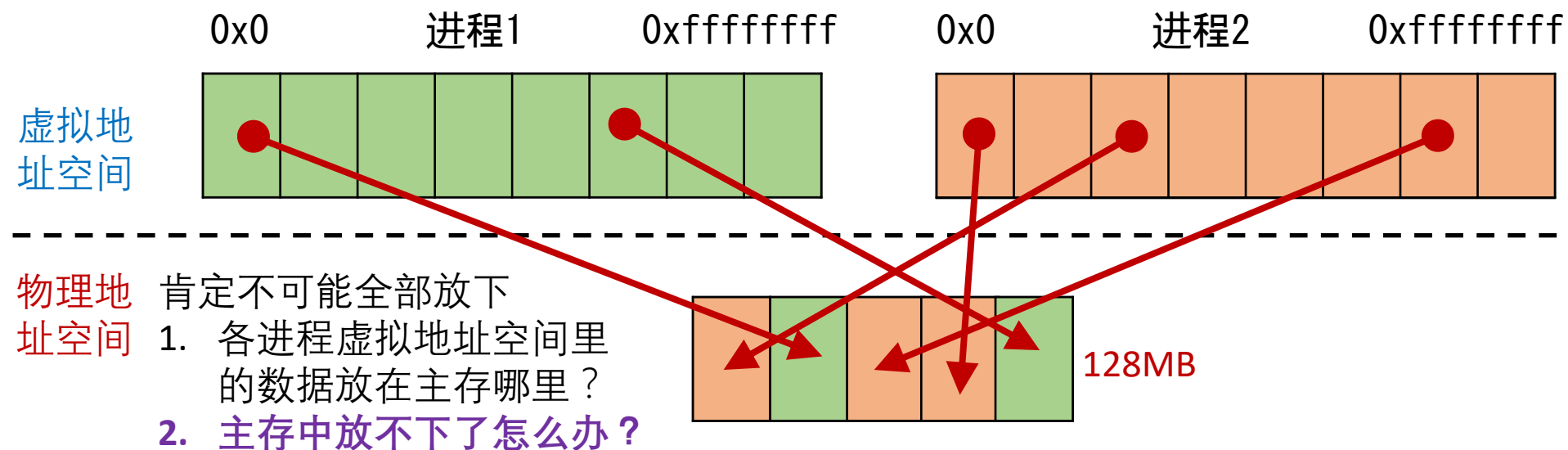
分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为**每个进程**提供了一个独立的、极大的虚拟地址空间



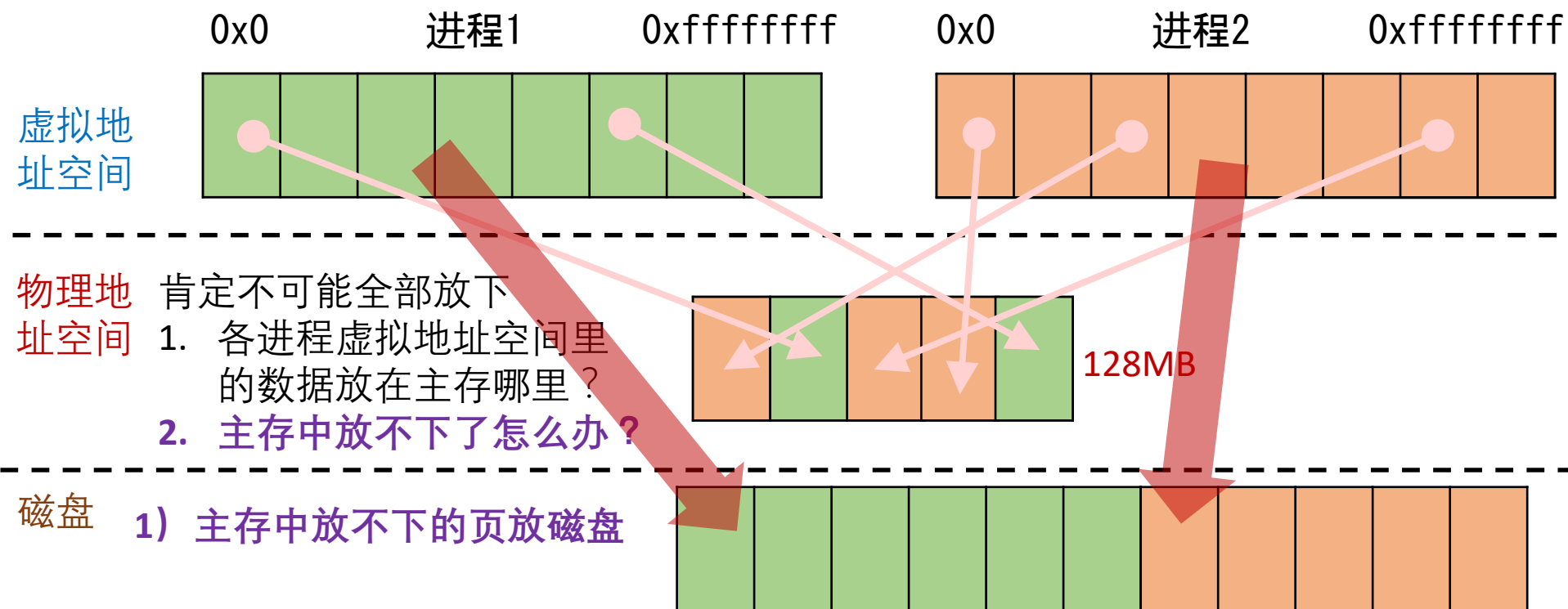
分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间



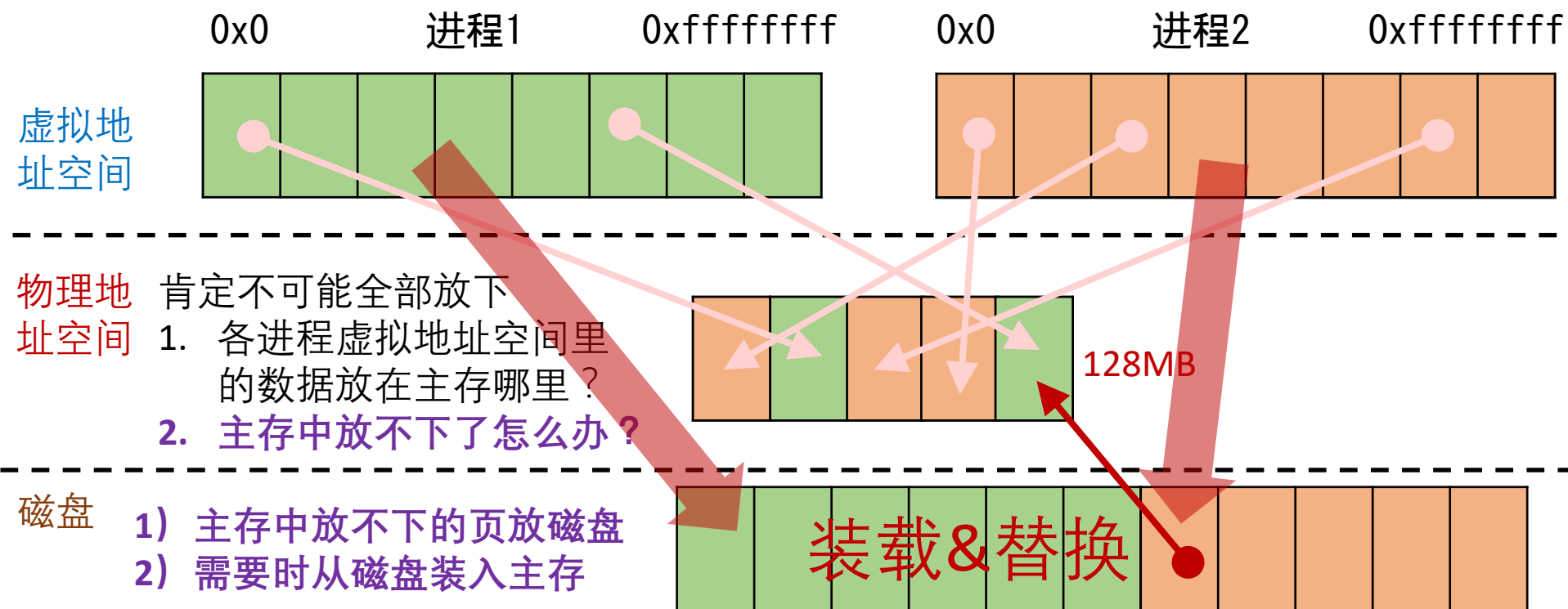
分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间



分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为**每个进程**提供了一个独立的、极大的虚拟地址空间

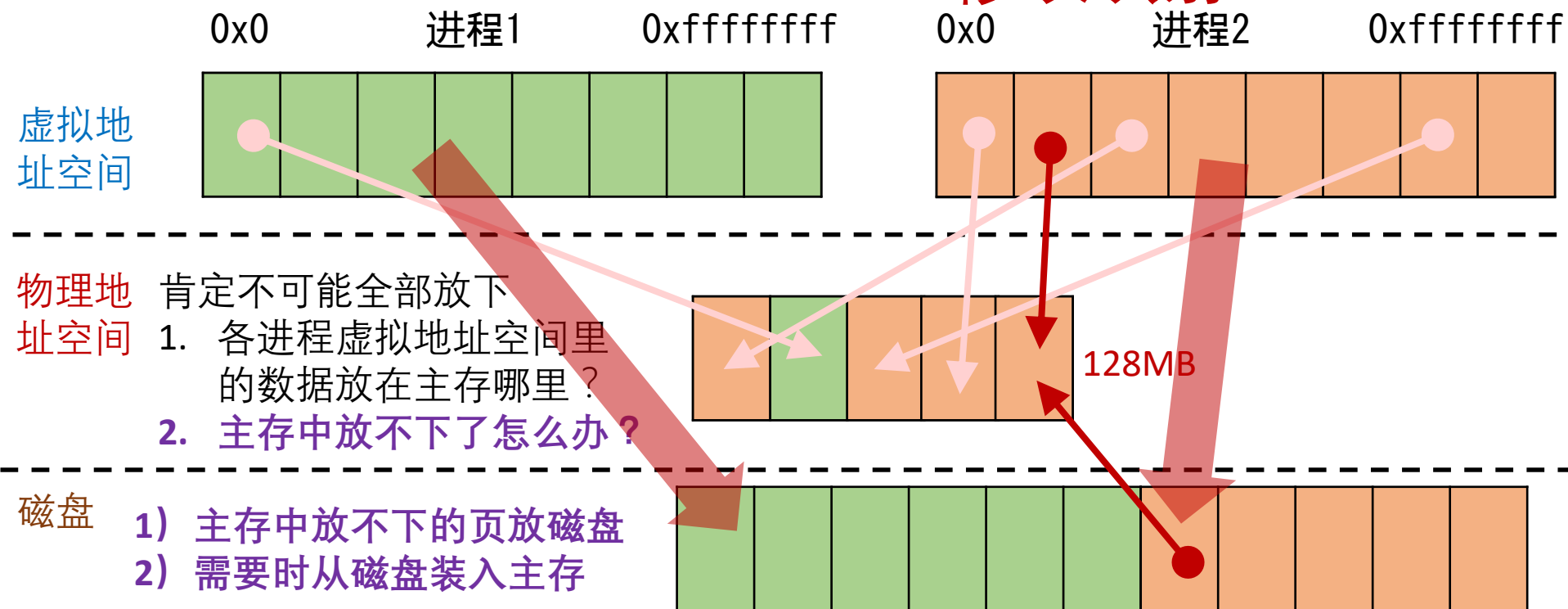


分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？

- 为每个进程提供了一个独立的、极大的虚拟地址空间

修改映射



分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间
 - 每个进程拥有一致的虚拟地址空间，方便管理
 - 将主存看做磁盘的缓存，主存中只保留当前活动的程序段和数据区，有效利用主存空间
 - 每个进程的虚拟地址空间为私有，提供保护

分页机制 - 原理

- 如何突破分段机制下对可寻址空间的限制？和多进程内存管理的困难？
 - 为每个进程提供了一个独立的、极大的虚拟地址空间
 - 每个进程拥有一致的虚拟地址空间，方便管理
 - 将主存看做磁盘的缓存，主存中只保留当前活动的程序段和数据区，有效利用主存空间
 - 每个进程的虚拟地址空间为私有，提供保护
- 机制的核心是什么？
 - 映射表：
 - 某个进程的某个虚拟页在不在物理内存中
 - 某个进程的某个虚拟页映射到哪个物理页（页框）

分页机制 – 页表

- 映射表：
 - 某个进程的某个虚拟页在不在物理内存中
 - 某个进程的某个虚拟页映射到哪个物理页（页框）

虚拟页号	在不在主存	对应哪个物理页
0x0	在	0x100
0x1	在	0x30
0x2	不在	N/A
0x3	不在	N/A
0x4	不在	N/A
0x5	在	0x1234
...

每个进程都有一个表格

这个表叫‘页表’

各进程表格每一项对应的物理页之间不产生冲突

进程间相互隔离

操作系统管理着所有进程的页表



分页机制 – 页表

- 映射表：
 - 某个进程的某个虚拟页在不在物理内存中
 - 某个进程的某个虚拟页映射到哪个物理页（页框）

虚拟页号	在不在主存	对应哪个物理页
0x0	在	0x100
0x1	在	0x30
0x2	不在	N/A
0x3	不在	N/A
0x4	不在	N/A
0x5	在	0x1234
...

页表在内存中存储为一个页表项的数组



一个页表项

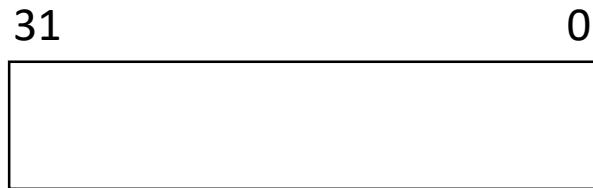
分页机制 – 页表

页表在内存中存储为一个页表项的数组

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U / S	R / W	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U / S	R / W	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U / S	R / W	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U / S	R / W	P
⋮										
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U / S	R / W	P

分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



页表在内存中存储为一个页表项的数组

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R
								/	/
								S	P
								W	

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R
								/	/
								S	P
								W	

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R
								/	/
								S	P
								W	

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R
								/	/
								S	P
								W	

⋮

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R
								/	/
								S	P
								W	

分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B

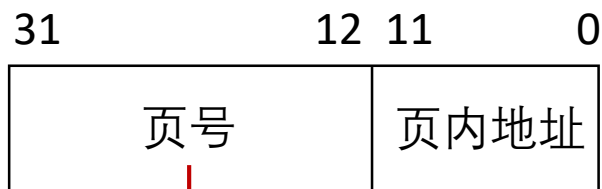
31	12	11	0
页号			页内地址

页表在内存中存储为一个页表项的数组

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
⋮												
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P

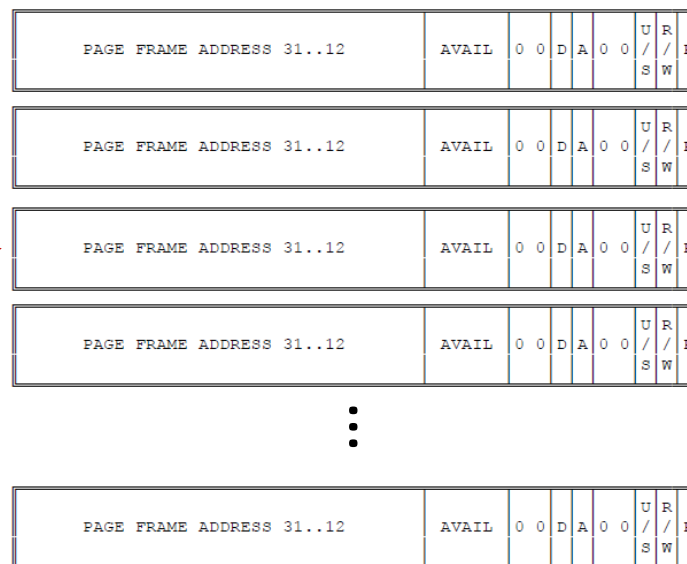
分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



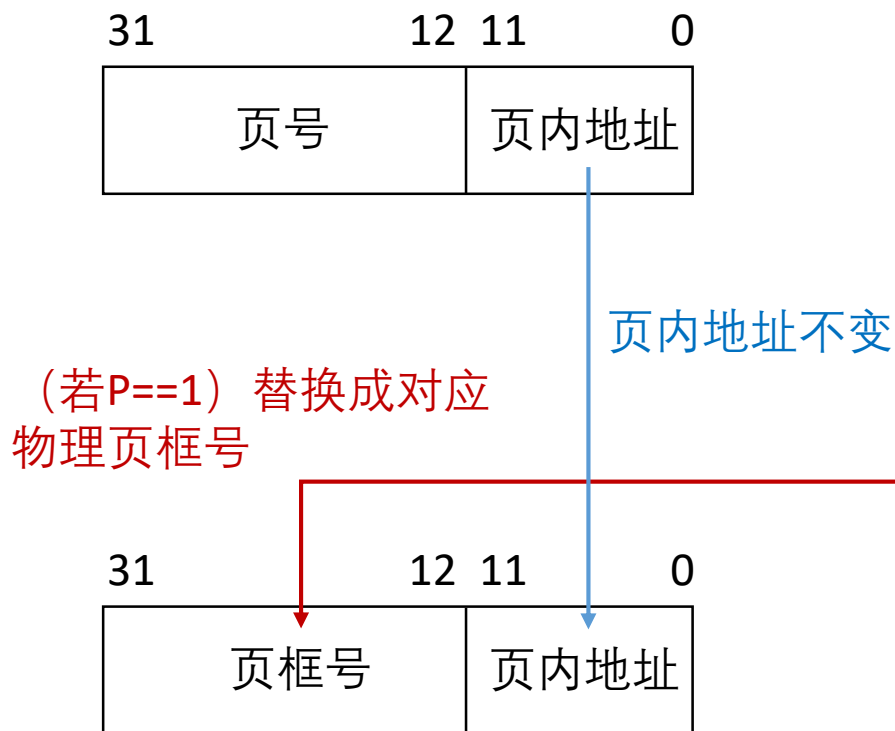
以页号为索引（下标）
找到页表项

页表在内存中存储为一个页表项的数组



分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



页表在内存中存储为一个页表项的数组

PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P
⋮												
PAGE FRAME ADDRESS 31..12	AVAIL	0	0	D	A	0	0	U	R	/	/	P

页级地址转换得到32位物理地址

分页机制 – 页表

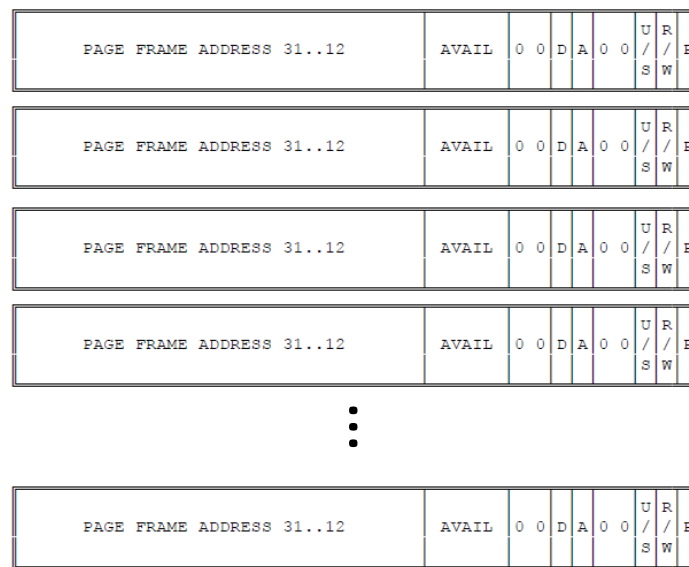
给出32位线性地址，约定一个页为4KB = 2^{12} B



页表在内存中存储为一个页表项的数组

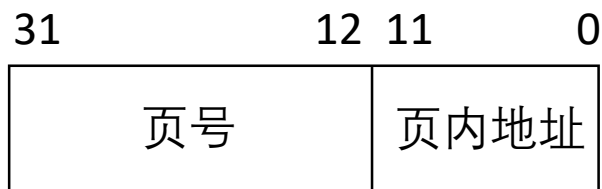
因为磁盘太慢了，不能浪费任何可能的物理页，因此虚拟页和物理页之间采用全相联映射

多少项？



分页机制 – 页表

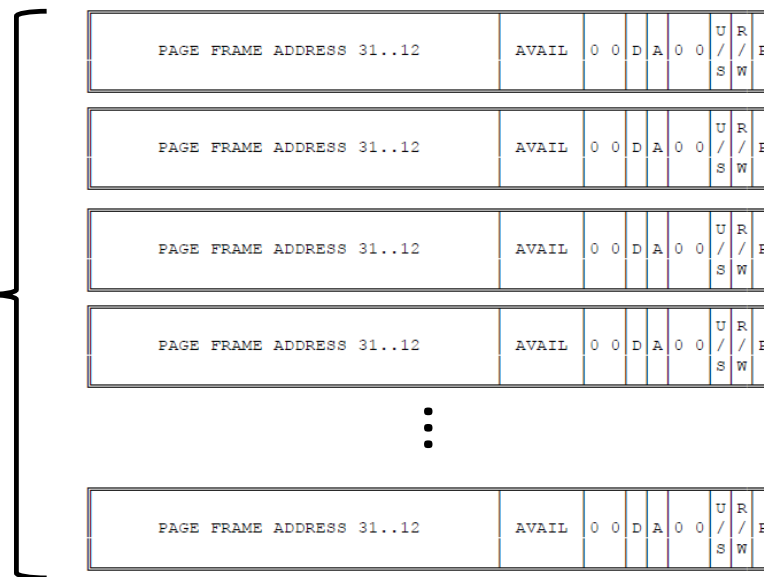
给出32位线性地址，约定一个页为4KB = 2^{12} B



页表在内存中存储为一个页表项的数组

因为磁盘太慢了，不能浪费任何可能的物理页，因此虚拟页和物理页之间采用全相联映射

2^{20} 项

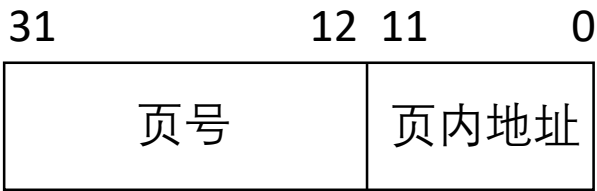


32位 = 4字节一项

分页机制 – 页表

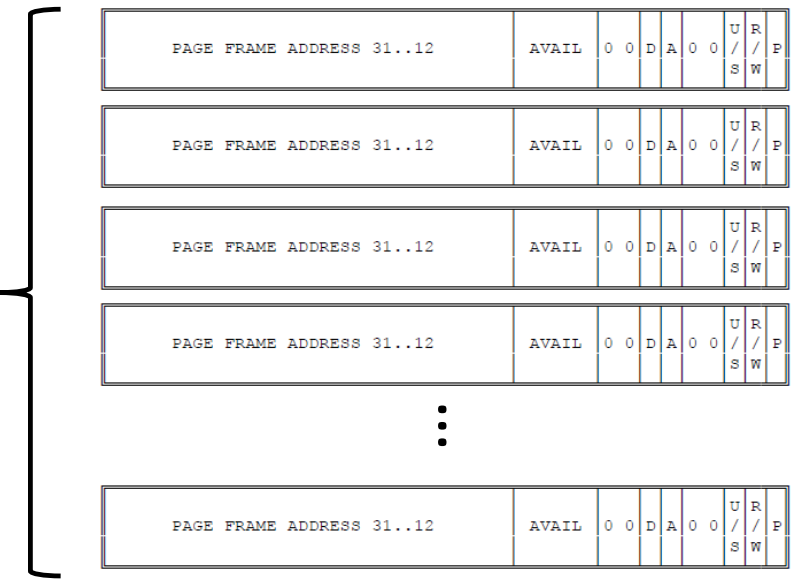
一个页表**4MB**，在内存中找到连续的4MB的数组空间不容易

给出32位线性地址，约定一个页为4KB = 2¹²B



页表在内存中存储为一个**页表项**的数组

2²⁰项

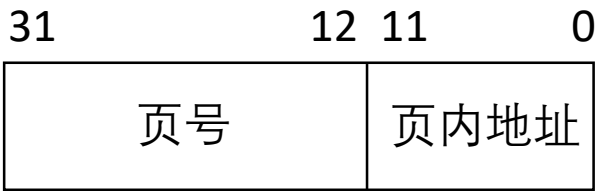


32位 = 4字节一项

因为磁盘太慢了，不能浪费任何可能的物理页，因此虚拟页和物理页之间采用全相联映射

分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2¹²B

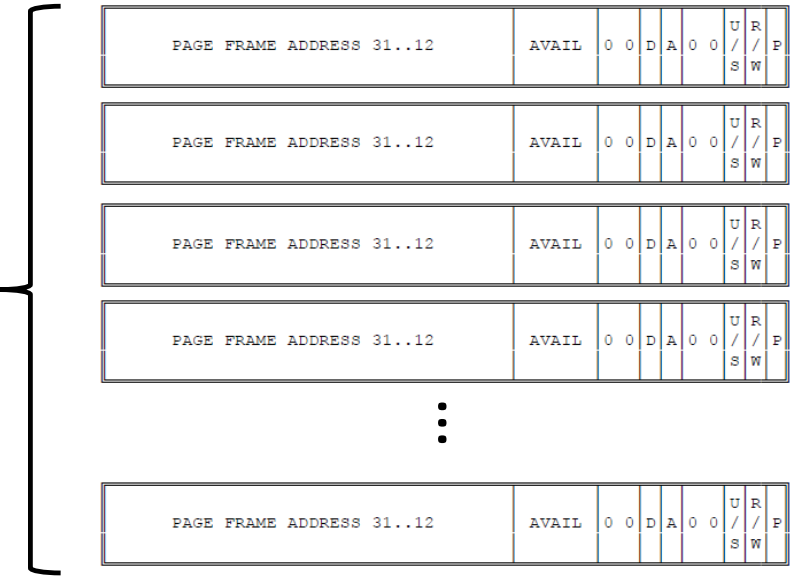


一个页表**4MB**，在内存中找到连续的4MB的数组空间不容易

拆分为两级页表

页表在内存中存储为一个**页表项**的数组

2²⁰项

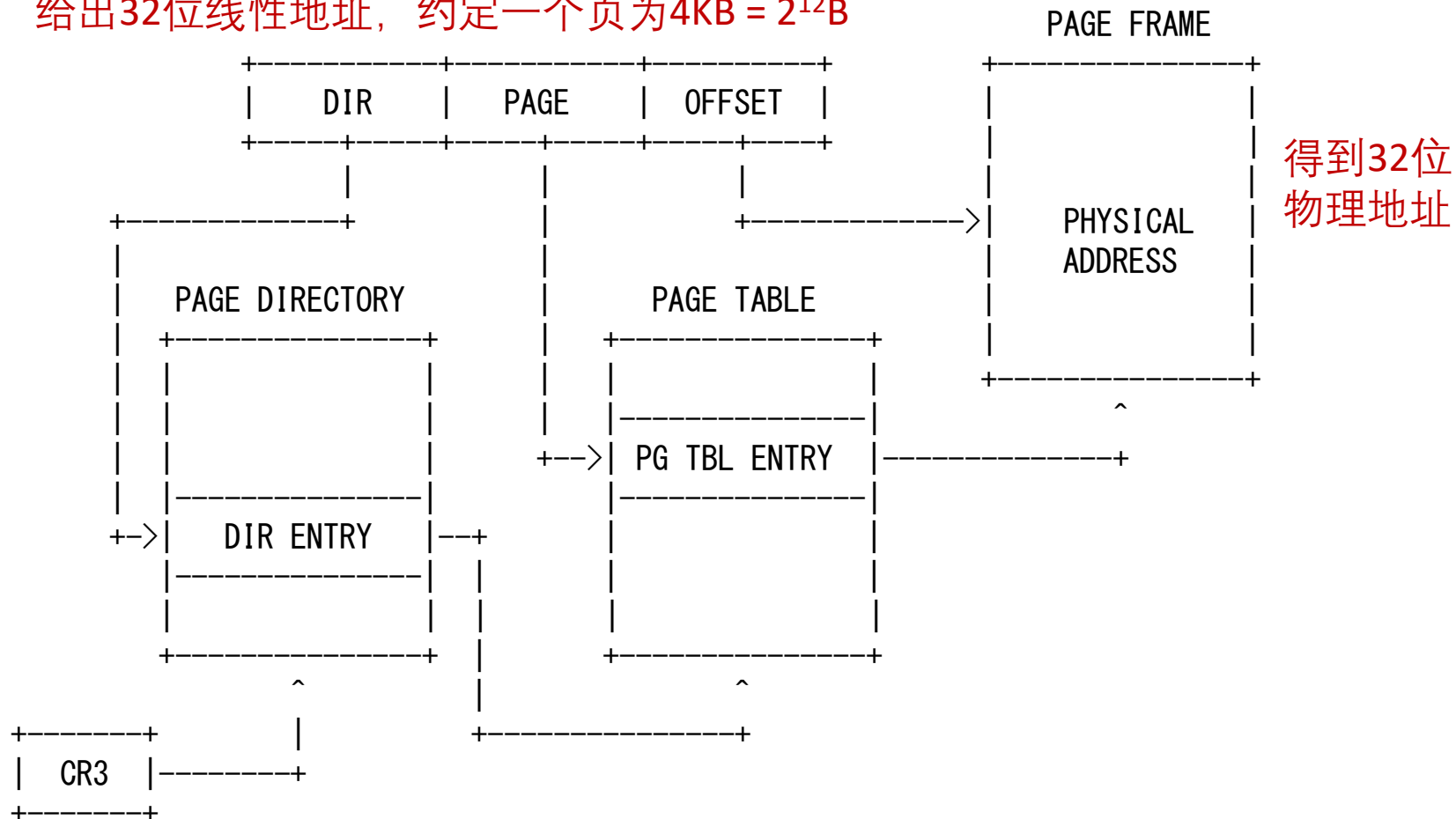


因为磁盘太慢了，不能浪费任何可能的物理页，因此虚拟页和物理页之间采用全相联映射

32位 = 4字节一项

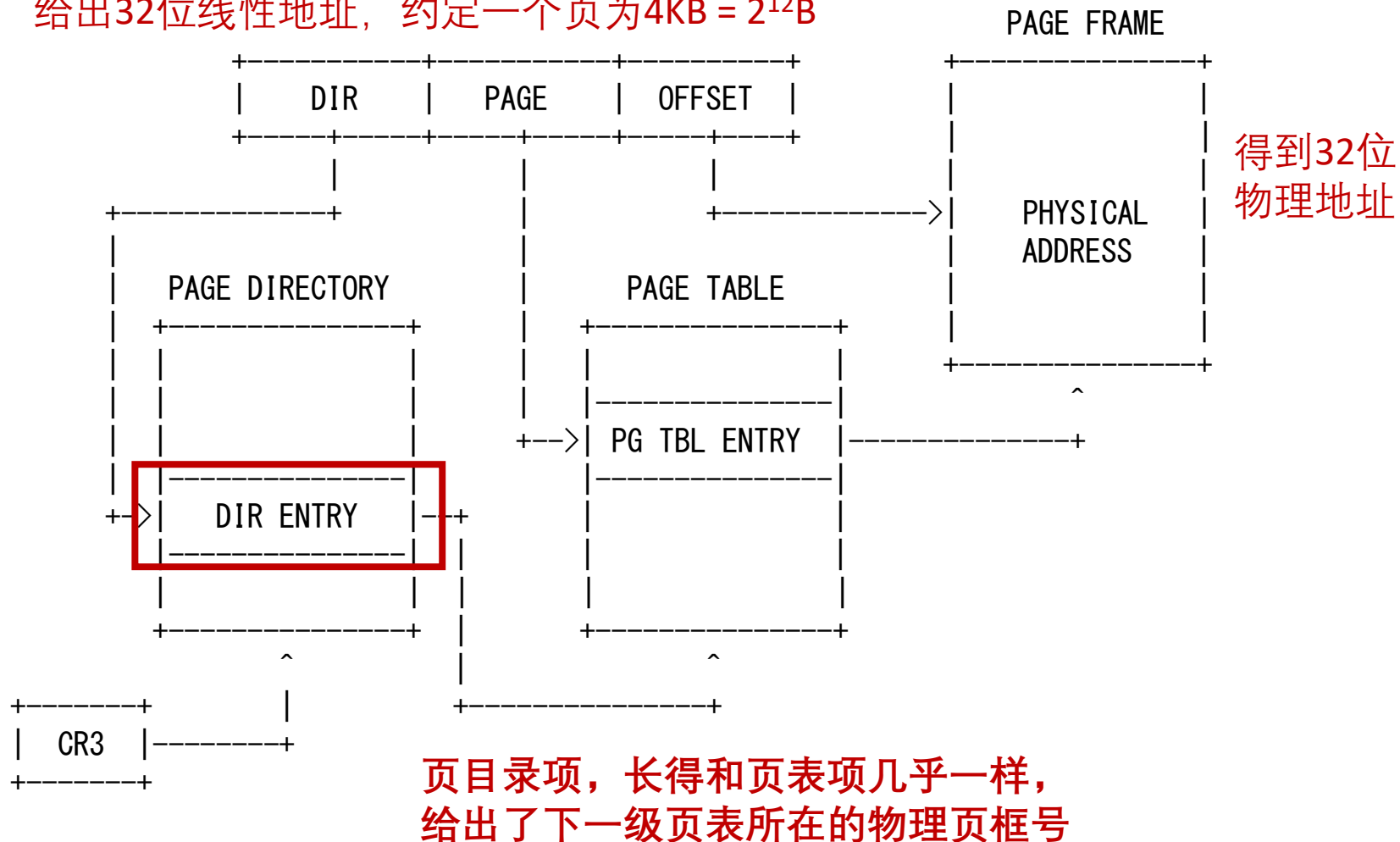
分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



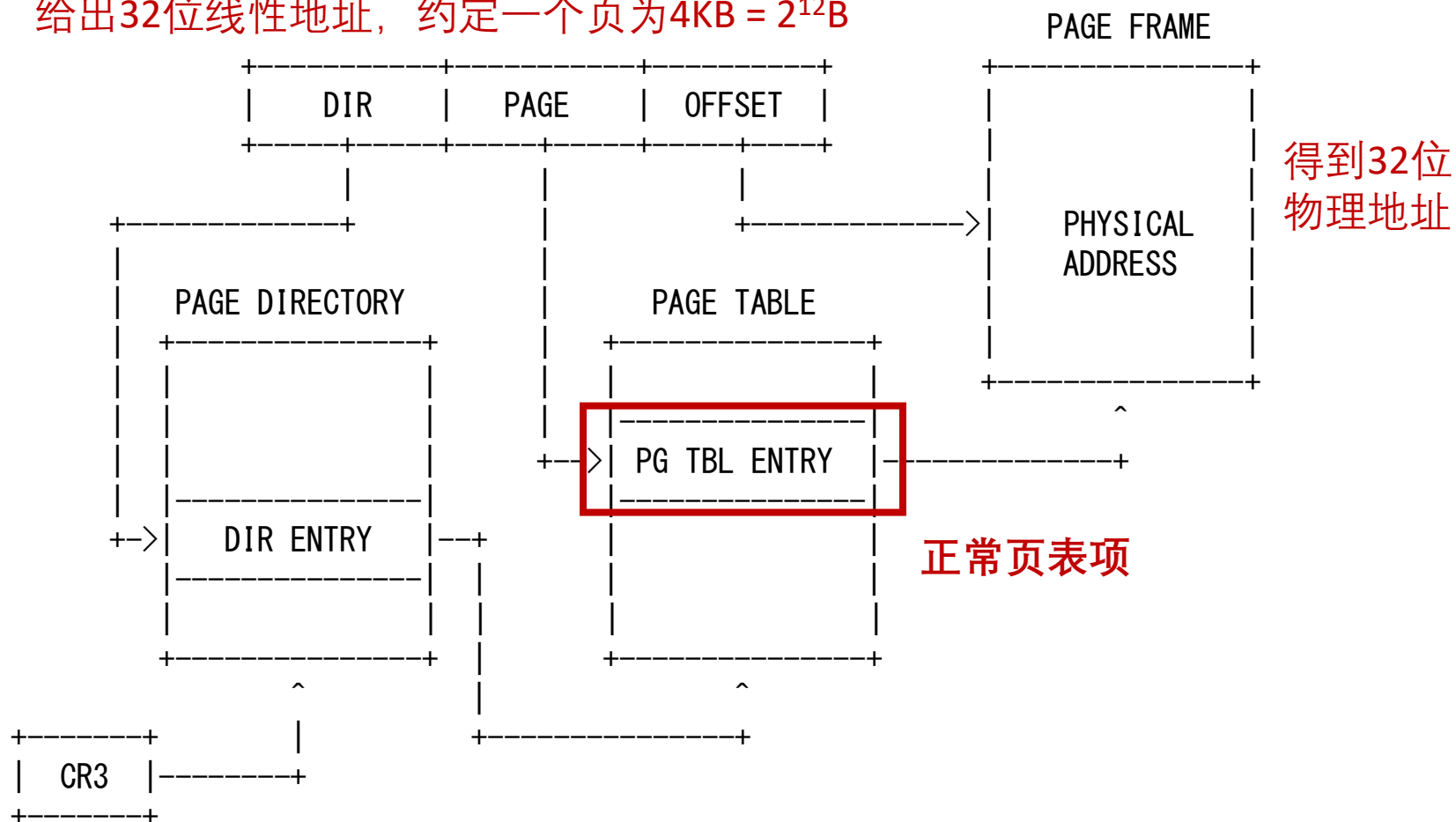
分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



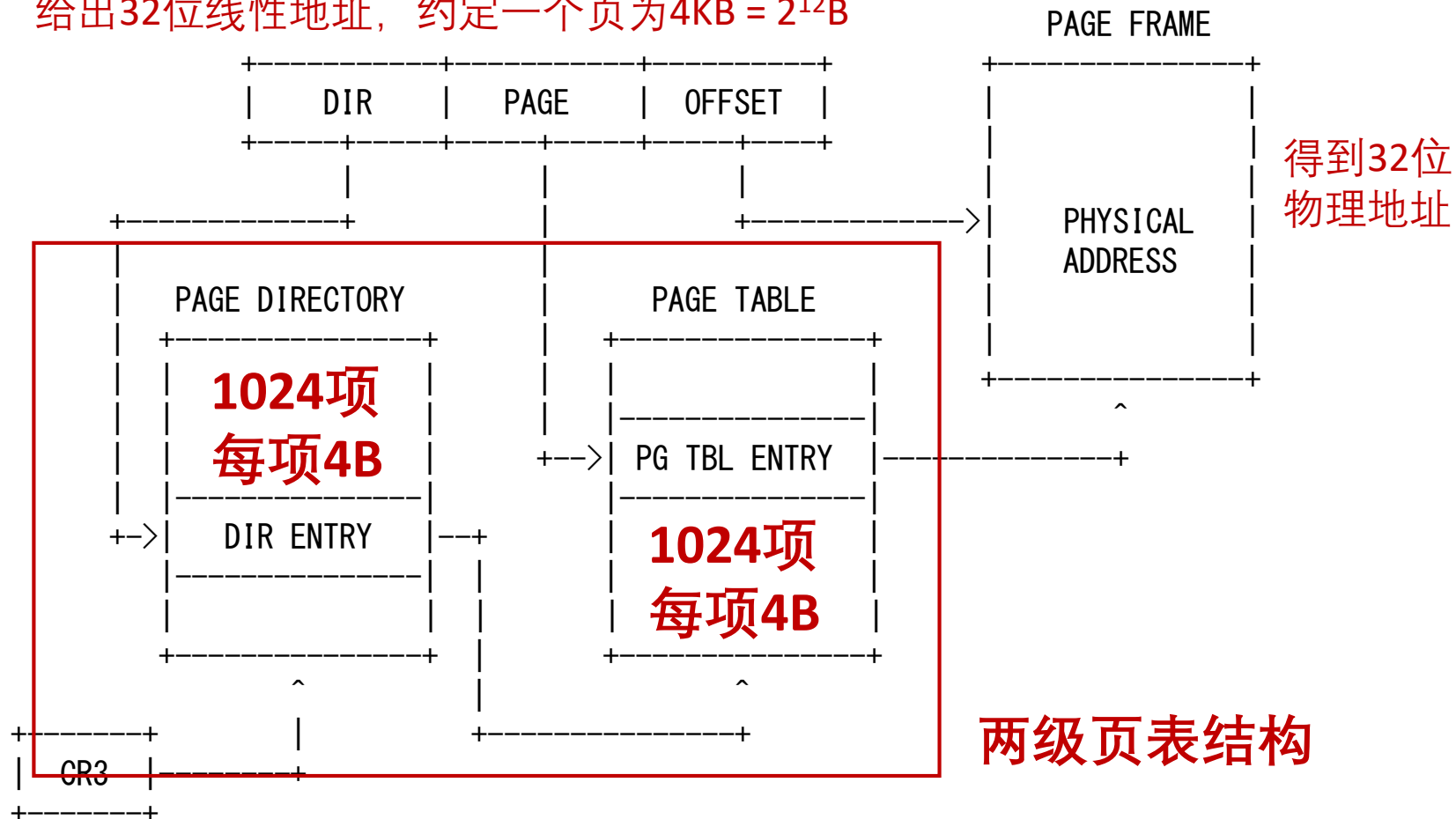
分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



分页机制 – 页表

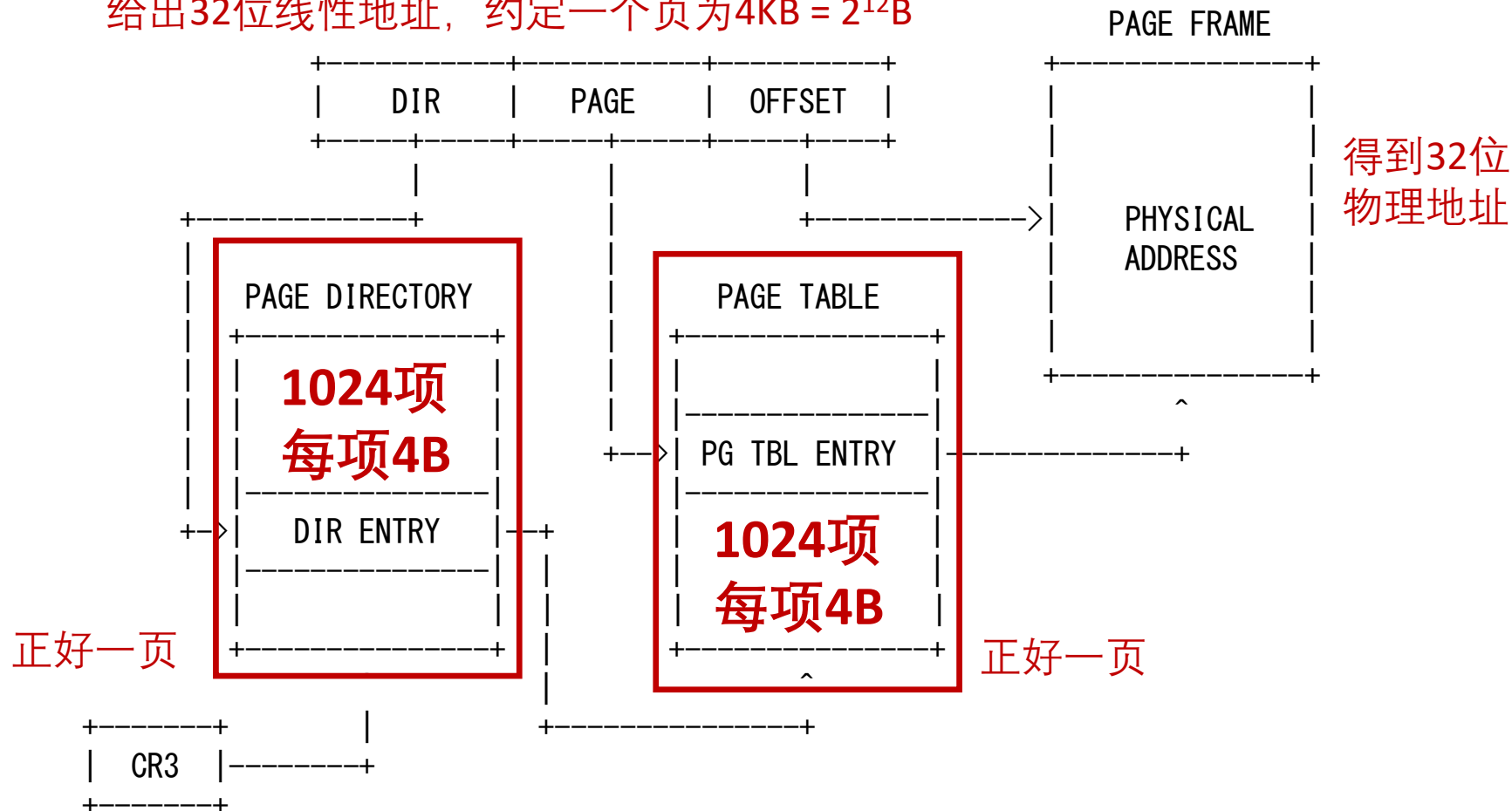
给出32位线性地址，约定一个页为4KB = 2^{12} B



分页机制 – 页表

DIR 位?
PAGE 位?
OFFSET 位?

给出32位线性地址，约定一个页为4KB = 2^{12} B

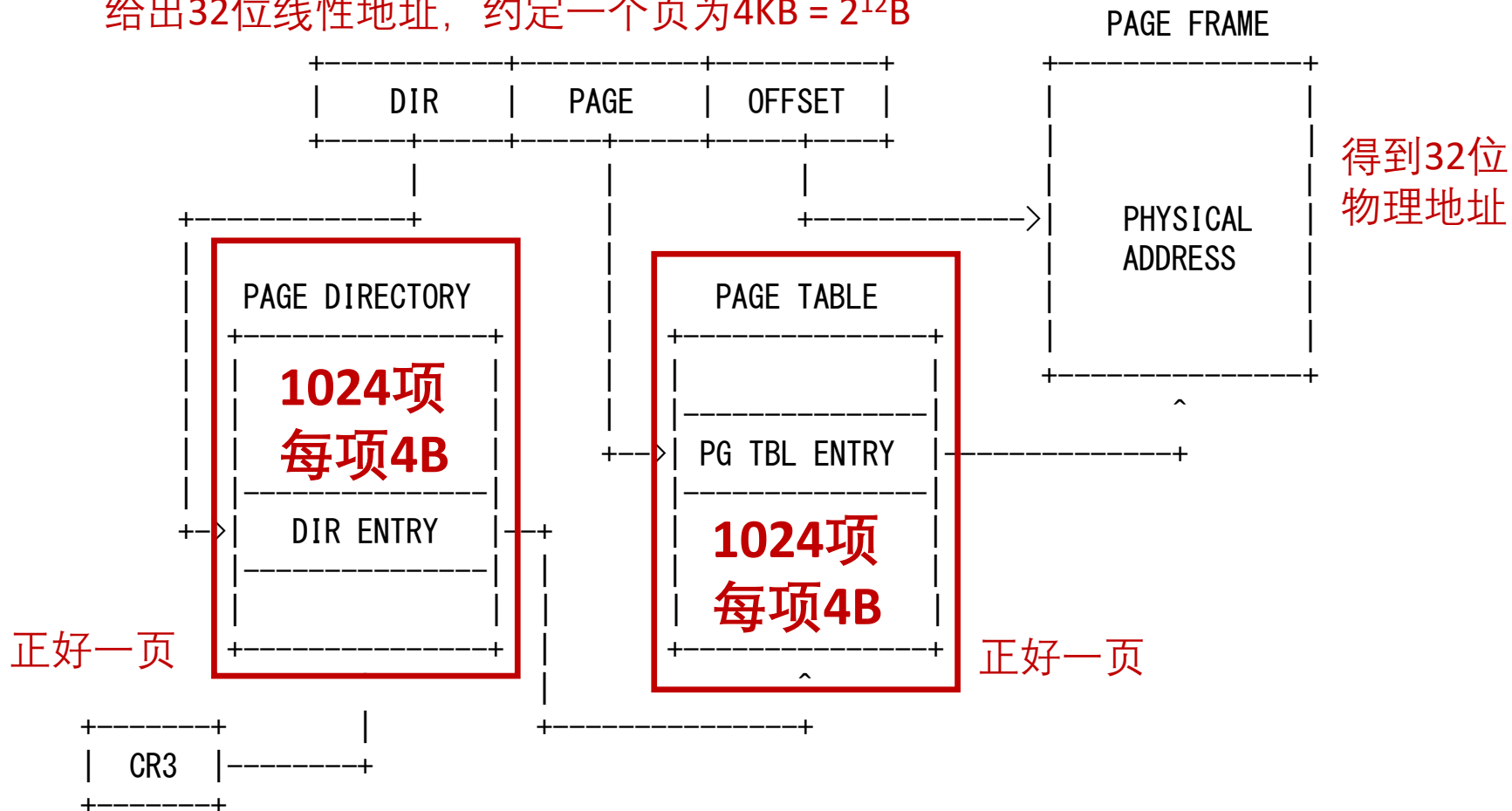


分页机制 – 页表

DIR 10 位 ?
PAGE 10 位 ?
OFFSET 12 位 ?

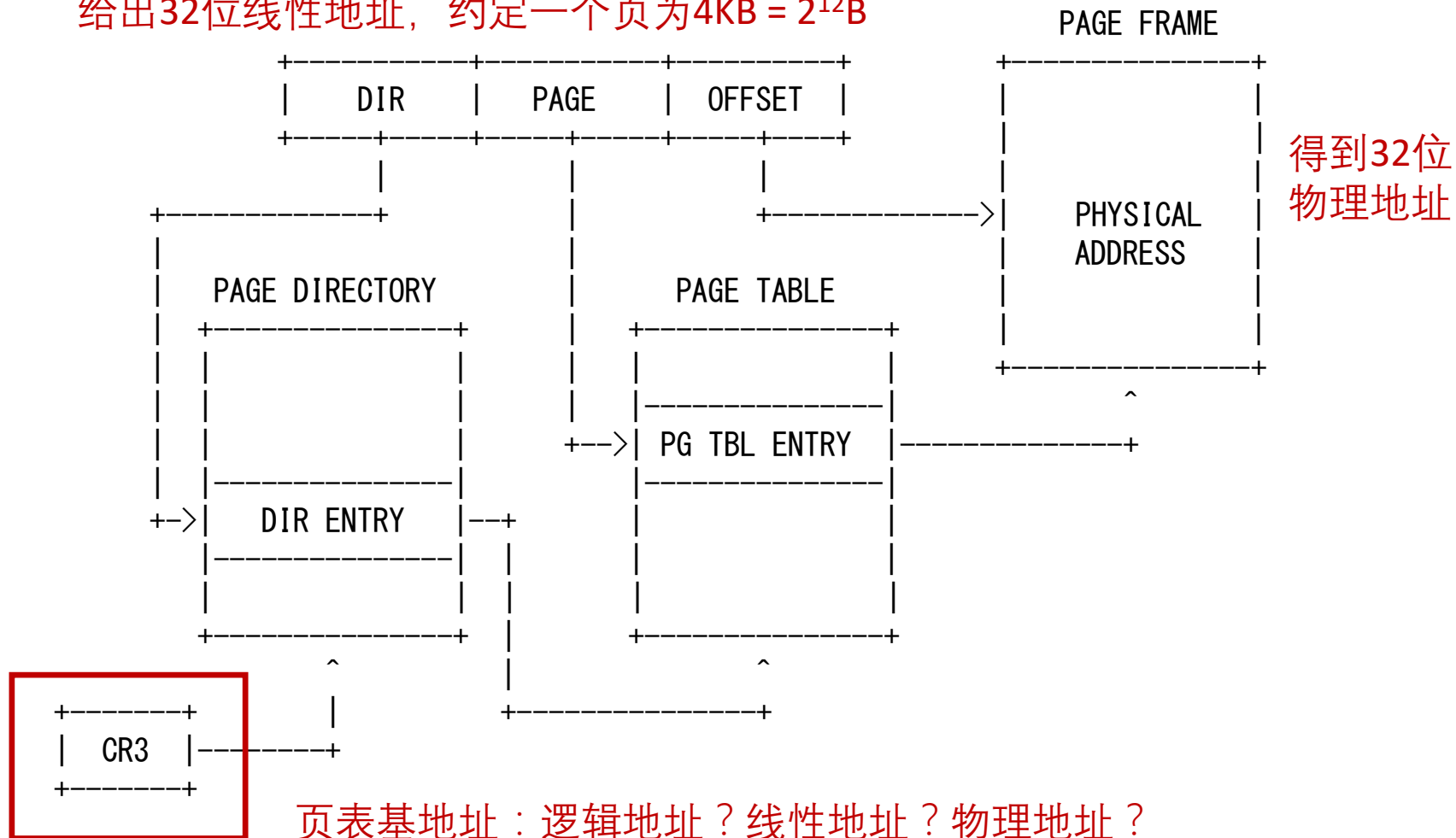
简单说就是前面pg.28 – 35的
20位页号被拆成了两个10位

给出32位线性地址，约定一个页为4KB = 2^{12} B



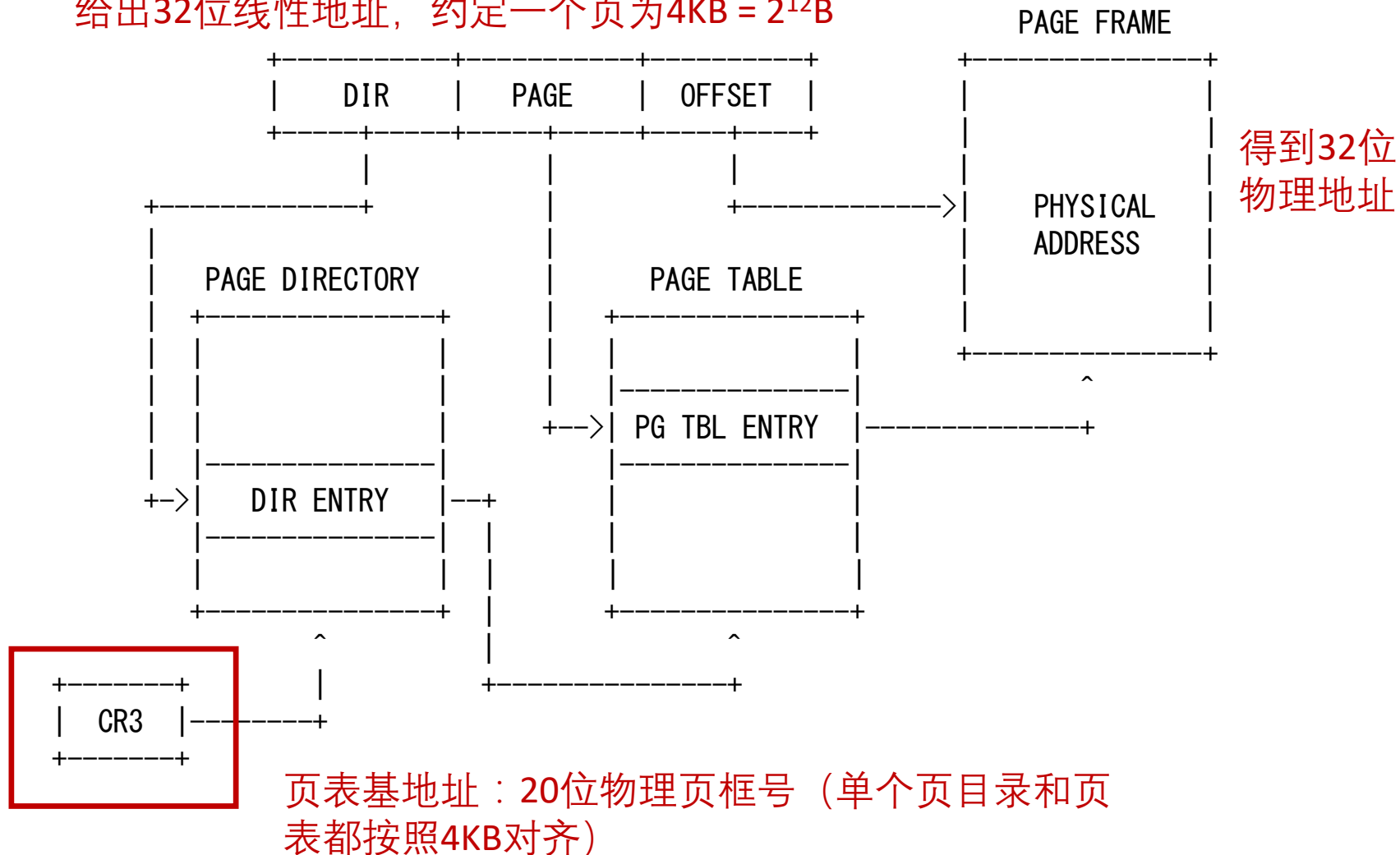
分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



分页机制 – 页表

给出32位线性地址，约定一个页为4KB = 2^{12} B



下面讲代码

分页机制 – 实现大体步骤

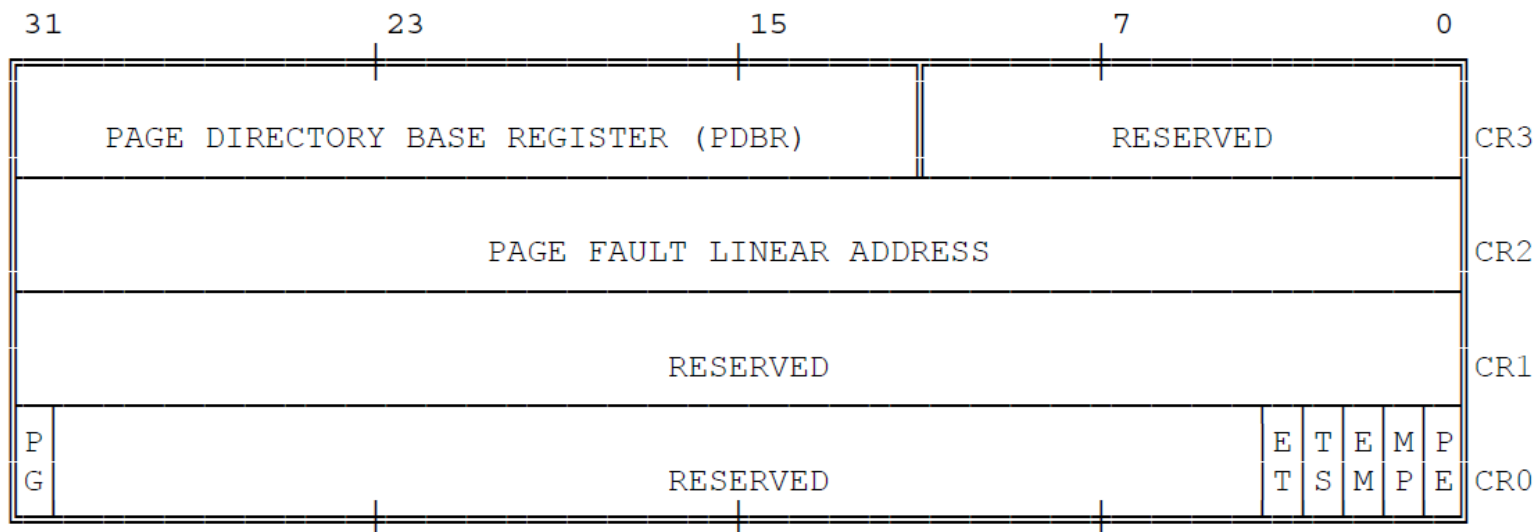
- 在NEMU中实现分页机制
 - 在include/config.h中

#define IA32_PAGE

- 在nemu中添加页级地址转换支持
- testcase和kernel链接选项的改变
- kernel的行为变化

分页机制 - nemu

- 在NEMU中实现分页机制
 - 从线性地址到物理地址的转换
 - 添加CR3寄存器，实现对页表基地址（物理页框号）的存储
 - 添加CR0的PG位，实现开启/关闭分页机制



PG和PE都初始化为0，若PG=PE=1则开启分页模式

分页机制 - nemu

- 在NEMU中实现分页机制
 - 从线性地址到物理地址的转换
 - 修改laddr_read()与laddr_write()函数

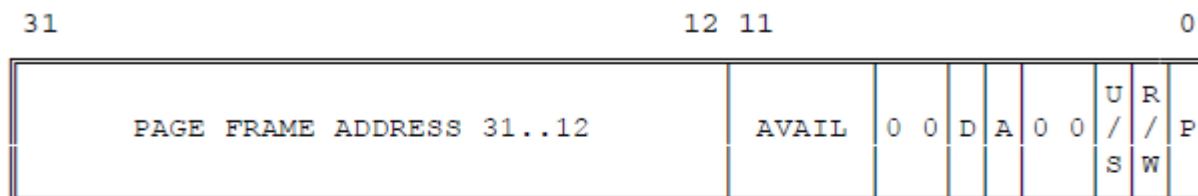
```
uint32_t laddr_read(laddr_t addr, size_t len) {  
    assert(len == 1 || len == 2 || len == 4);  
    if( CRO什么状态 ) {  
        if (data cross the page boundary) {  
            /* this is a special case, you can handle it later. */  
            assert(0);  
        } else {  
            hwaddr_t hwaddr = page_translate(addr);  
            return hwaddr_read(hwaddr, len);  
        }  
    } else { ... }  
}
```

page_translate() 定义在
nemu/src/memory/mmu/page.c

分页机制 - nemu

- 在NEMU中实现分页机制
 - 从线性地址到物理地址的转换
 - 了解页表项的实现

nemu/include/memory/mmu/page.h



P	-	PRESENT
R/W	-	READ/WRITE
U/S	-	USER/SUPERVISOR
D	-	DIRTY
AVAIL	-	AVAILABLE FOR SYSTEMS PROGRAMMER USE

NOTE: 0 INDICATES INTEL RESERVED. DO NOT DEFINE.

在实现page_translate()时, 务必使用assertion检查目录项和页表项的present位, 如果发现了一个无效的表项, 及时终止NEMU的运行, 否则调试将会异常困难

分页机制 – 链接选项

- 修改 kernel/Makefile 中的链接选项

```
- LDFLAGS = -Ttext=0x30000 -m elf_i386 # before page  
+ LDFLAGS = -Ttext=0xc0030000 -m elf_i386
```

让kernel的代码从虚拟地址 0xc0030000 开始

- 修改testcase/Makefile中的链接选项

```
- LDFLAGS := -m elf_i386 -e start -Ttext=0x60000  
+ LDFLAGS := -m elf_i386 -e start
```

让testcase的虚拟地址空间符合Linux的约定

分页机制 – kernel行为变化

start.S

```
#ifdef IA32_PAGE
#    define KOFFSET 0xc0000000
#    define va_to_pa(x) (x - KOFFSET)
#else
#    define va_to_pa(x) (x)
#endif
```

```
.globl start
```

```
start:
```

```
...
```

```
    lgdt    va_to_pa(gdt_desc)
```

```
...
```

改变了kernel/Makefile后，在
kernel初始化页表前，为啥要用
va_to_pa宏？

分页机制 – kernel行为变化

main.c

```
void init() {
#ifdef IA32_PAGE
    /* ...
    * Before setting up correct paging, no global variable can be used. */
    init_page(); // 初始化kernel页表, src/memory/kvm.c, 建议读代码画页映射关系

    /* After paging is enabled, transform %esp to virtual address. */
    asm volatile(“addl %0, %%esp” : : “i” (KOFFSET)); // esp指向高地址
#endif

    /* Jump to init_cond() to continue initialization. */
#ifdef IA32_PAGE
    asm volatile(“jmp *%0” : : “r” (init_cond + 0xc0000000)); // gcc-6有bug
#else
    asm volatile(“jmp *%0” : : “r”(init_cond));
#endif

    /* Should never reach here. */
    nemu_assert(0);
}
```

Why?

分页机制 – kernel行为变化

main.c

```
void init_cond() {
    ...
    #ifdef IA32_PAGE
        /* Initialize the memory manager. */
        init_mm(); // 初始化用户程序页表, src/memory/mm.c
                  // 拷贝了哪一部分? 对比一下PPT pg. 12的虚拟地址空间
    #endif
    ...
        /* Load the program. */
        uint32_t eip = loader(); // 在装载程序时使用mm_malloc() 接口,
                                // 具体看教程
    ...
    #ifdef IA32_PAGE
        /* Set the %esp for user program, which is one of the
         * convention of the "advanced" runtime environment. */
        asm volatile("movl %0, %%esp" : : "i"(KOFFSET));
    #endif

    /* Here we go! */
    ((void(*) (void))eip) ();
}
```

loader()需要修改!!!

// 在装载程序时使用mm_malloc() 接口,
// 具体看教程

/* Set the %esp for user program, which is one of the
* convention of the "advanced" runtime environment. */
asm volatile("movl %0, %%esp" : : "i"(KOFFSET));

/* Here we go! */
((void(*) (void))eip) ();

TLB – 页表的cache

缓存页表项的，善自体会，课本pg. 269，图6.34是个好图

PA 3-3 任务 - 编码

1. 修改Kernel和testcase中Makefile的链接选项；
2. 在include/config.h头文件中定义宏IA32_PAGE并make clean；
3. 在CPU_STATE中添加CR3寄存器；
4. 修改laddr_read()和laddr_write(), 适时调用page_translate()函数进行地址翻译；
5. 修改Kernel的loader(), 使用mm_malloc来完成对用户进程空间的分配；
6. 通过make testkernel执行并通过各测试用例。

PA 3-3 任务 – 报告

1. Kernel的虚拟页和物理页的映射关系是什么？
请画图说明；
2. 以某一个测试用例为例，画图说明用户进程的虚拟页和物理页间映射关系又是怎样的？
Kernel映射为哪一段？你可以在loader()中通过Log()输出mm_malloc的结果来查看映射关系，并结合init_mm()中的代码绘出内核映射关系。
3. “在Kernel完成页表初始化前，程序无法访问全局变量”这一表述是否正确？在init_page()里面我们对全局变量进行了怎样的处理？

PA 3-3截止时间，整体截止啦！！！！
2017年12月27日24时，即，28日0时
各种节日快乐，22日的课可能由我上

PA 3-3到此结束

祝大家学习快乐，身心健康！

欢迎大家踊跃参加问卷调查