



# 操作系统

## 第5章 存储器管理

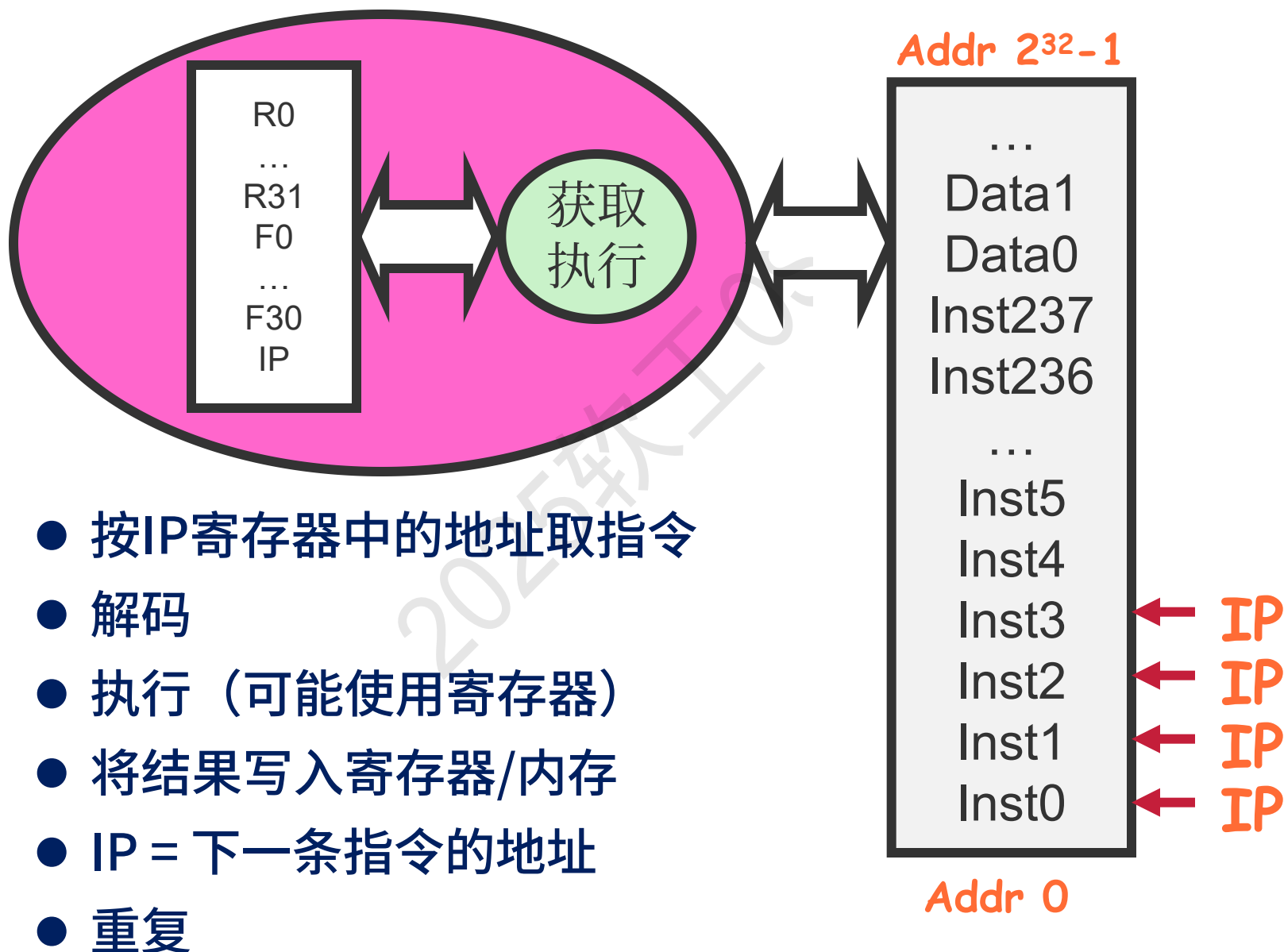
朱小军, 教授  
<https://xzhu.info>  
南京航空航天大学  
计算机科学与技术学院  
2025年春

# 存储器管理

- 内存管理的需求
- 物理内存分配方案
  - 连续分配存储管理
  - 分页存储管理
  - 分段存储管理
- 虚拟存储器

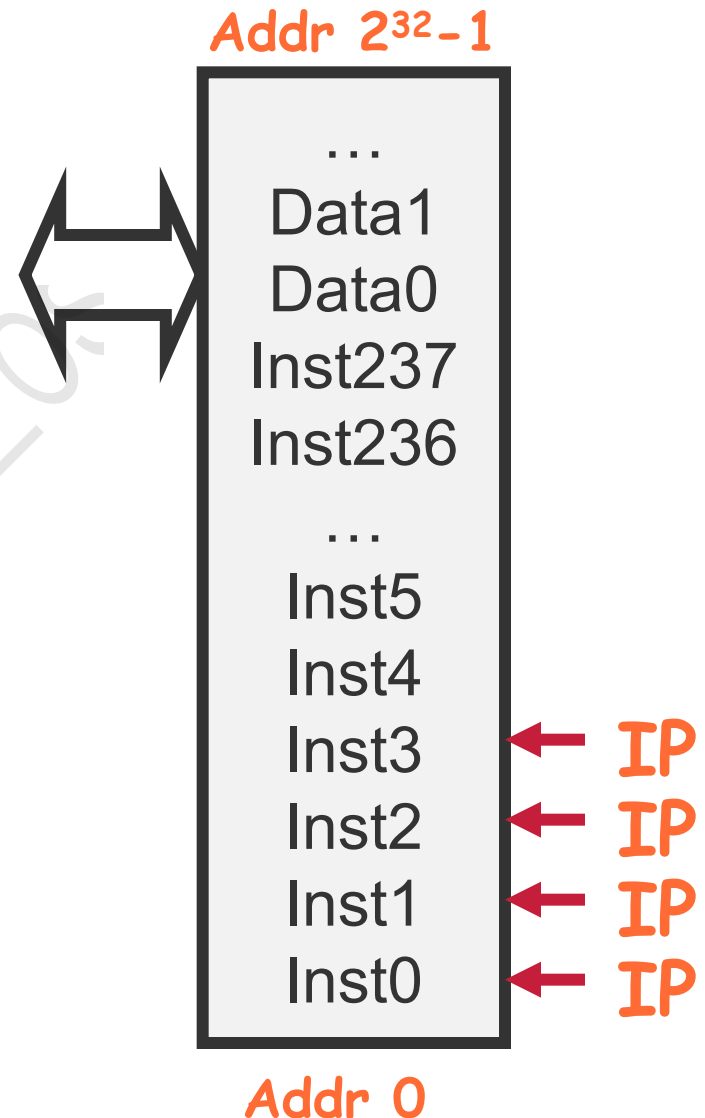
在程序员、进程的眼中，内存长什么样？

# 回忆：程序是怎么被执行的？

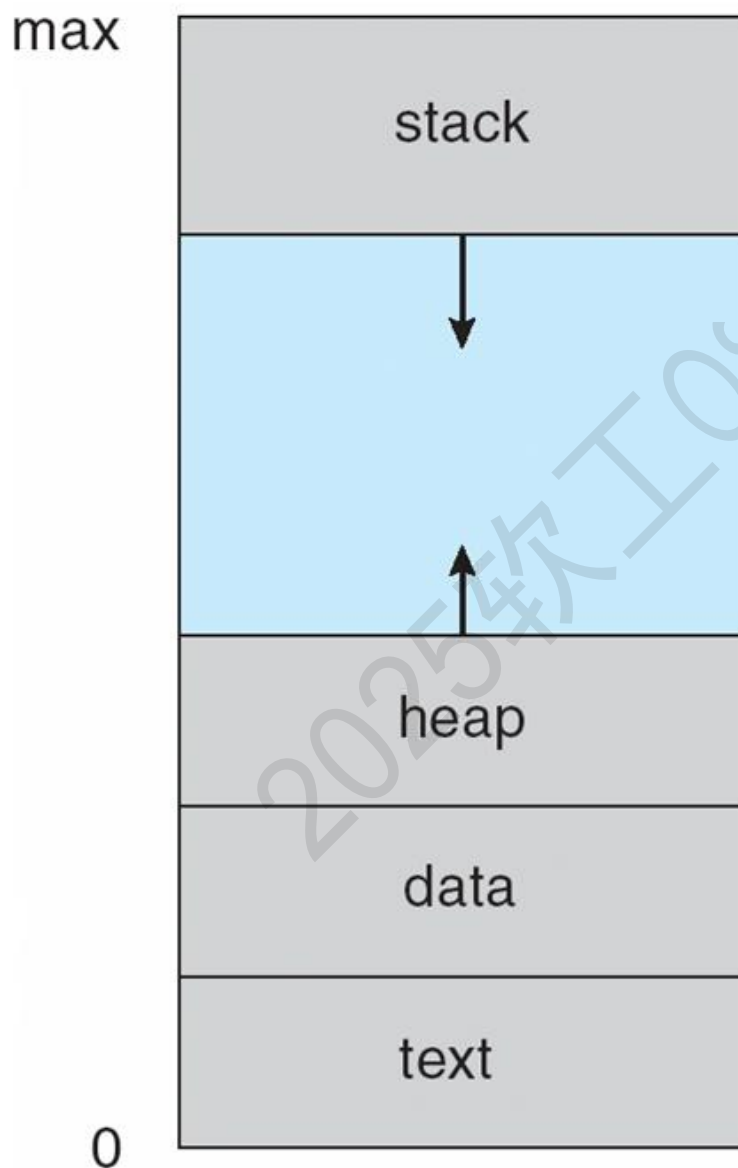


# 回忆：程序是怎么被执行的？

- 按IP寄存器中的地址取指令
- 解码
- 执行（可能使用寄存器）
- 将结果写入寄存器/内存
- IP = 下一条指令的地址
- 重复



# 前面的“内存”，就是OS提供的内存抽象




# 地址空间

- Logical Address Space

- the set of all possible addresses that the process can generate

- 逻辑地址空间

- 进程执行过程中，从CPU发出的地址的集合
- 逻辑地址空间中的地址**不一定是合法的**
- 下面这个程序会终止吗？



```
#include "stdio.h"
```

```
void main(){  
    int *i,*heap;  
    i=&i;  
  
    heap=malloc(4);  
    printf("%x %x\n",i,heap);  
    while(1){  
        if(*i>=0)  
            i--;  
        else  
            i--;  
        if((int)i%1000==0)  
            printf("%x\n",i);  
    }  
}
```



# 问题出在哪？

- CPU发出去了一系列地址，其中一个地址不合法，触发了系统的保护机制(课程结束之后，你应当能够回答整个触发流程)

```
while(1){  
    if(*i>=0)  
        i--;  
    else  
        i--;
```

- 再次强调：逻辑地址是CPU发出的地址

此外，操作系统提供的内存抽象，应当支持“动态链接”

# 回忆：编译与（静态）链接

```
int main() {  
    int x = 10, y = 20;  
    printf("Before swap: x = %d, y = %d\n", x, y);  
    swap(&x, &y);  
    printf("After swap:  x = %d, y = %d\n", x, y);  
    return 0;  
}
```

main.c

```
void swap(int *a, int *b)  
{  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

swap.c

```
gcc -m32 main.c swap.c -o swap
```

```
$ ./swap  
Before swap: x = 10, y = 20  
After swap:  x = 20, y = 10
```

```
int main() {
    int x = 10, y = 20;
    printf("Before swap: x = %d, y = %d\n", x, y);
    swap(&x, &y);
    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

编译 gcc -c

**main.o**

```
00000000 <main>:
0:  f3 0f 1e fb      endbr32
4:  8d 4c 24 04      lea    0x4(%esp),%ecx
8:  83 e4 f0          and    $0xffffffff0,%esp
b:  ff 71 fc          pushl  -0x4(%ecx)
e:  55               push   %ebp
f:  89 e5             mov    %esp,%ebp
11: 53               push   %ebx
12: 51               push   %ecx
13: 83 ec 10          sub    $0x10,%esp
```

**swap.o**

```
00000000 <swap>:
0:  f3 0f 1e fb      endbr32
4:  55               push   %ebp
5:  89 e5             mov    %esp,%ebp
7:  83 ec 10          sub    $0x10,%esp
a:  e8 fc ff ff ff   call   b <swap+0xb>
f:  05 01 00 00 00    add    $0x1,%eax
14: 8b 45 08          mov    0x8(%ebp),%eax
17: 8b 00             mov    (%eax),%eax
```

```
000011ed <main>:
11ed:  f3 0f 1e fb      endbr32
11f1:  8d 4c 24 04      lea    0x4(%esp),%ecx
11f5:  83 e4 f0          and    $0xffffffff0,%esp
11f8:  ff 71 fc          pushl  -0x4(%ecx)
11fb:  55               push   %ebp
11fc:  89 e5             mov    %esp,%ebp
11fe:  53               push   %ebx
11ff:  51               push   %ecx
1200:  83 ec 10          sub    $0x10,%esp
...
0000128e <swap>:
128e:  f3 0f 1e fb      endbr32
1292:  55               push   %ebp
1293:  89 e5             mov    %esp,%ebp
```

链接

gcc main.o swap.o -o swap

# 前面的链接方式称为静态链接

## ● 静态链接

- 在程序运行前，将各目标模块及它们所需的库函数链接成一个完整的装配模块，以后不再拆开
- 对相对地址进行修改；变换外部调用符号

## ● 如果库函数被很多进程采用呢？

- 比如，很多程序都需要调用swap函数，链接进所有程序浪费空间
- 最常见的，glibc
- 动态链接：将链接过程延迟到装入或者运行时

# 装入时动态链接 (Load-time linking)

## ● 编译共享库

➤ `gcc -m32 -fPIC -shared -o libswap.so swap.c`

## ● 编译可执行文件

➤ `gcc -m32 -o main main.c -L. -lswap`  
■ 链接器记录了动态依赖, 可以查验 `readelf -d main`

## ● 执行程序

■ `export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH`

➤ `./main`

## ● 加载main; 加载libswap.so; 重定位swap 函数; 运行

# 运行时动态链接 (Run-time linking)

## ● 编译共享库

➤ `gcc -m32 -fPIC -shared -o libswap.so swap.c`

## ● 修改源文件

## ● 编译可执行文件

➤ `gcc -m32 -o main  
main.c -ldl`

## ● 执行程序

➤ `./main`

## ● 加载main; 运行; dlopen加载libswap.so; dlsym解析符号地址; 重定位swap函数

```
#include <stdio.h>
#include <dlfcn.h>

int main() {
    int x = 10, y = 20;
    printf("Before swap: x = %d, y = %d\n", x, y);
    void *handle = dlopen("./libswap.so", RTLD_LAZY);
    void (*swap)(int*, int*) = dlsym(handle, "swap");
    swap(&x, &y);
    printf("After swap: x = %d, y = %d\n", x, y);
    return 0;
}
```

# 两种动态链接

- 装入时动态链接

- 装入内存时链接
- 可以实现对目标模块的共享

- 运行时动态链接

- 将目标模块的链接推迟到执行时
- 相对于装入时动态链接，避免装入太多无用模块，如游戏中动态加载地图、场景



但是，动态链接的目标模块，映射到虚拟地址空间的哪个区？

➤ data? text? heap? stack?

**stack**



**shared  
library**



**heap**

**data**

**text**

# 静态链接与动态链接

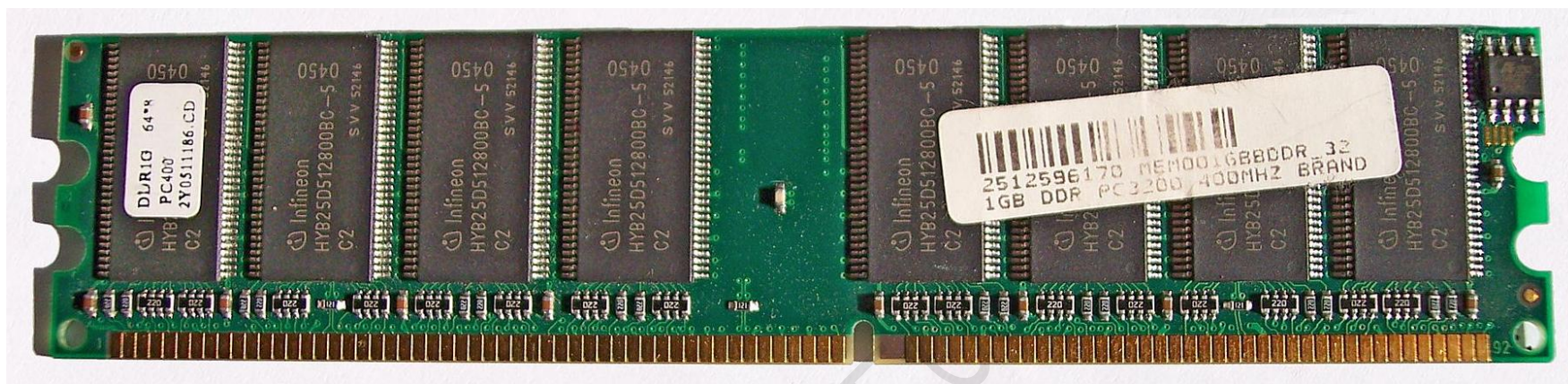
- 动态链接相对于静态链接的优点？

- 便于修改和更新
- 便于共享
- 可执行文件小
- 节省内存

- OS怎么提供动态链接支持呢？

OS可以通过哪些资源满足用户需求?

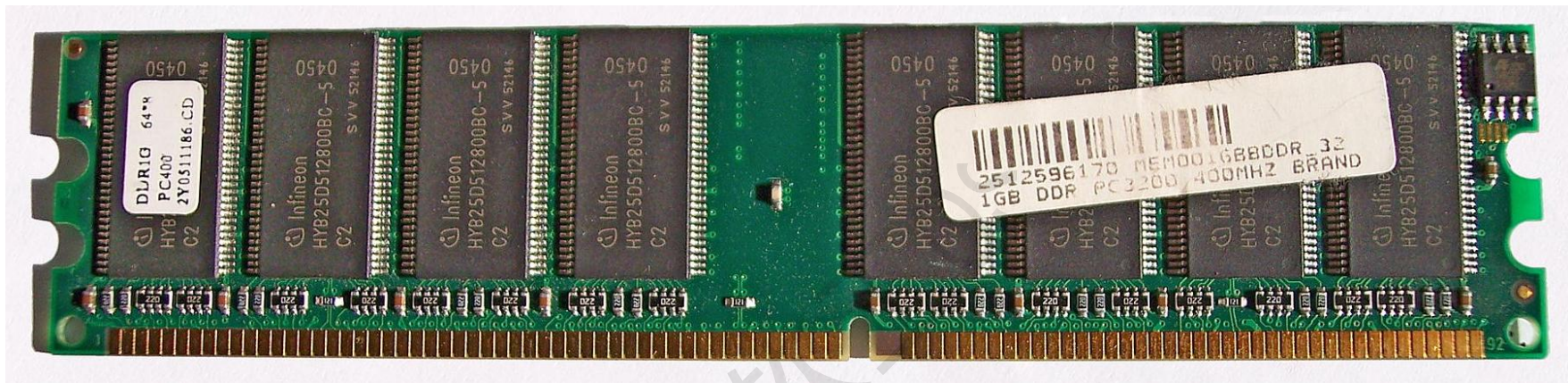
# 物理地址空间



## ● 物理地址空间

- 物理内存的实际地址的集合
- 1G内存:  $0x0 \sim 0x3FFF\ FFFF$
- 4G内存:  $0x0 \sim 0xFFFF\ FFFF$
- 物理地址是MMU发出的地址 (??? 后续会讲)

# 物理硬件



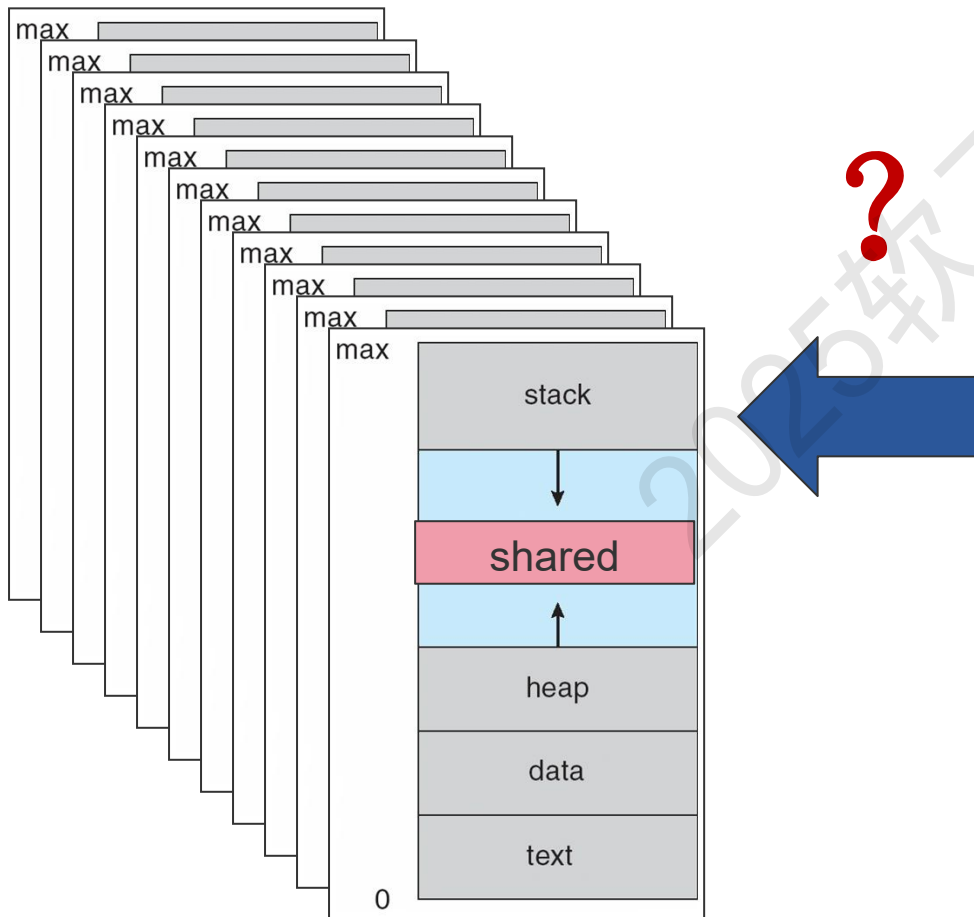
- CPU只可访问物理内存
- 物理内存不足可以借助磁盘





# 内存管理的任务

- 给一块物理内存和磁盘，为多个进程提供内存抽象（逻辑地址空间），让它们正常运行



# 存储器管理

- 内存管理的需求
- 物理内存分配方案
  - 连续分配存储管理
  - 分页存储管理
  - 分段存储管理
- 虚拟存储器



# 物理内存分配

- 预留部分内存给操作系统，其余分配给进程
- 应当做到（内存保护）
  - 用户进程不可访问OS的内存
  - 分配给一个用户进程的内存不可被其他进程读取和修改（除非向OS请求）
- 留意以下问题
  - 如何记录哪些内存已/未被分配？（此信息存哪？）
    - 新的请求如何分配内存？ 进程消亡时如何收回内存？
  - 如何进行地址转换？
  - 能否支持动态链接？

# 存储器管理

- 内存管理的需求
- 物理内存分配方案
  - 连续分配存储管理
  - 分页存储管理
  - 分段存储管理
- 虚拟存储器

# 连续分配存储管理

# 单一连续分配

- 最简单的存储管理方式

- 只能用于单用户、单任务的操作系统

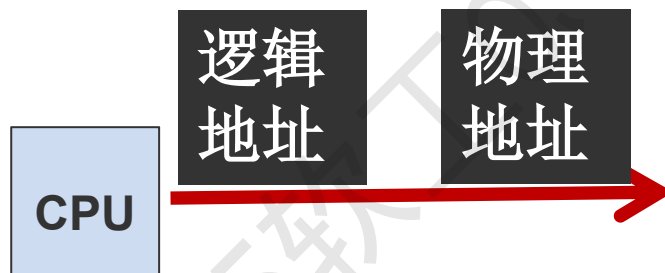
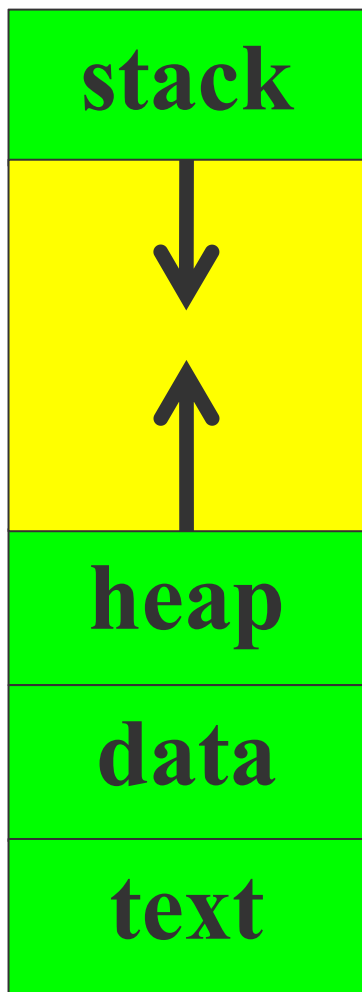
- 方法：

- 将可分配内存全部分配给用户进程

- 逻辑地址 = 物理地址

- 如果没有保护机制，用户进程可修改OS内存，如  
MS-DOS

# 单一连续分配下，逻辑地址即为物理地址



# 单一连续分配的优缺点

## ● 优点

- 简单
- 基本不需要硬件支持

## ● 缺点

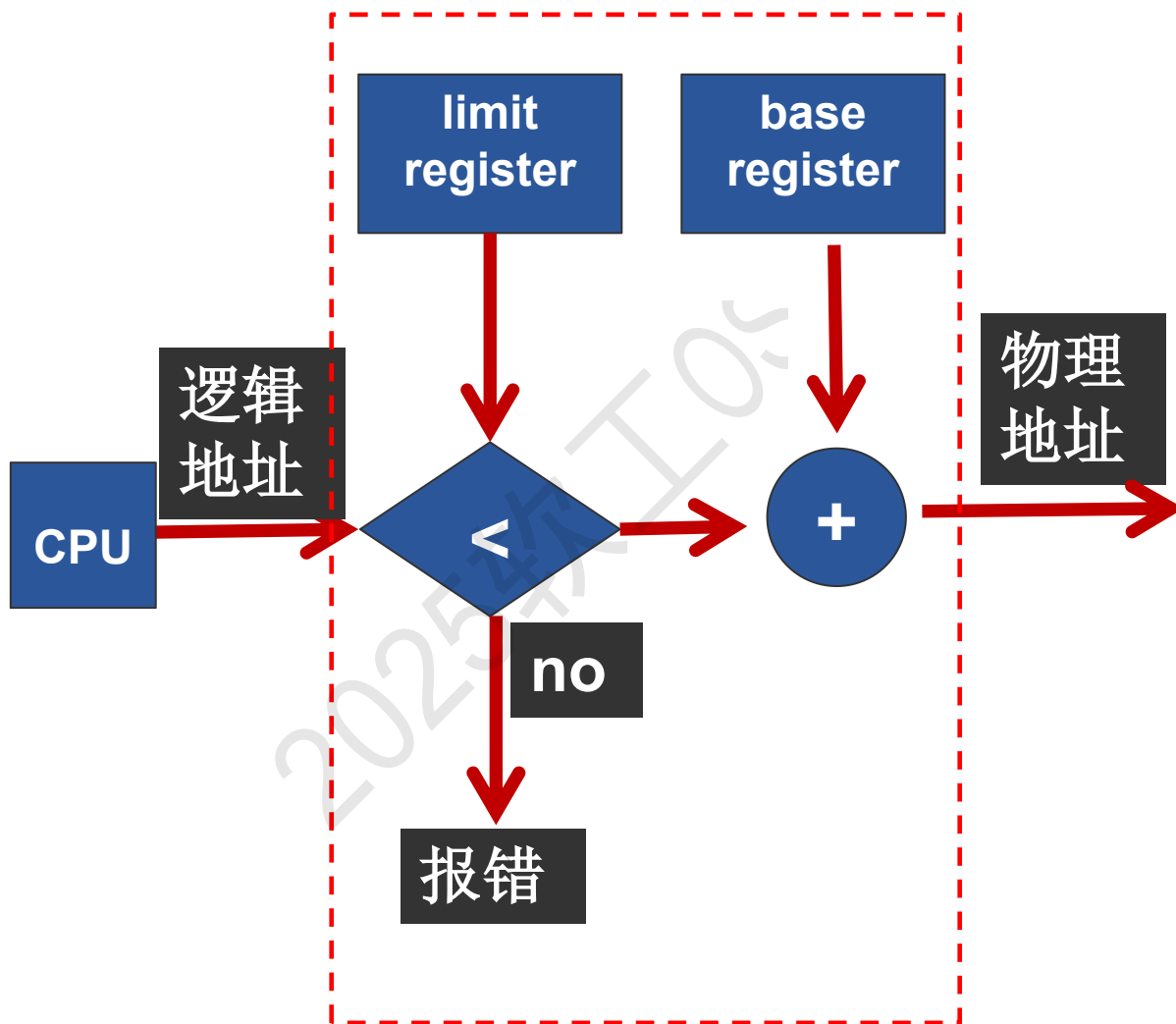
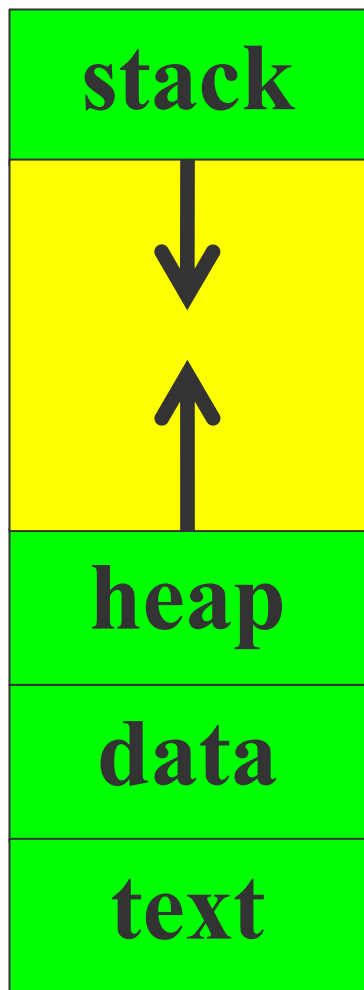
- 单任务
- 无法支持动态链接 (代际技术鸿沟)

# 固定分区分配

- 将内存划分为固定大小的区域（分多大？）
- 每个区域装入一道程序
- 分配与回收（how？）

分区号	大小 (KB)	起址 (K)	状态
1	12	20	已分配
2	32	32	已分配
3	64	64	未分配
4	128	128	已分配

# 固定分区下逻辑地址与物理地址的转换



(一个简单的)  
MMU



# 固定分区分配的优缺点

## ● 优点

- 简单、支持多道程序
- 开始支持内存保护

## ● 缺点

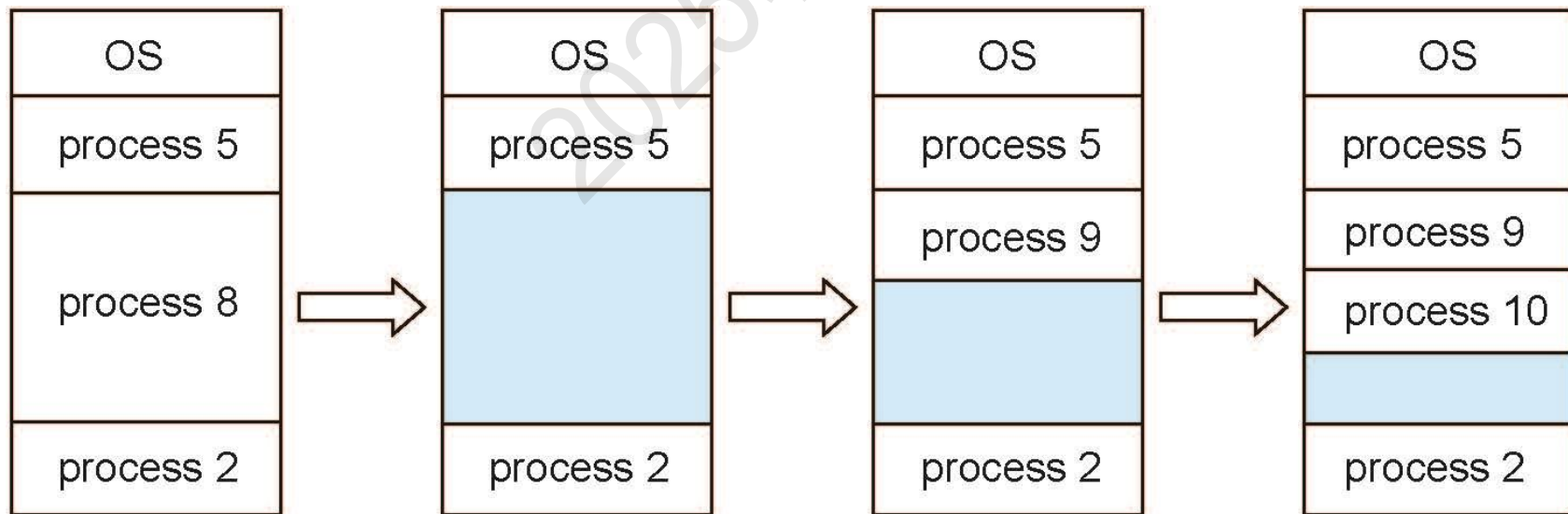
- 分区多大合适?
- 无法支持动态链接 (代际技术鸿沟)

# 可变分区分配

- 分区大小可变

- Hole的形成与动态变化

- 开始时内存是完整一块，一个hole
- 请求内存-> 从hole中选出一个分配，更新hole
- 释放 -> 更新hole



## ● 空闲内存分配算法

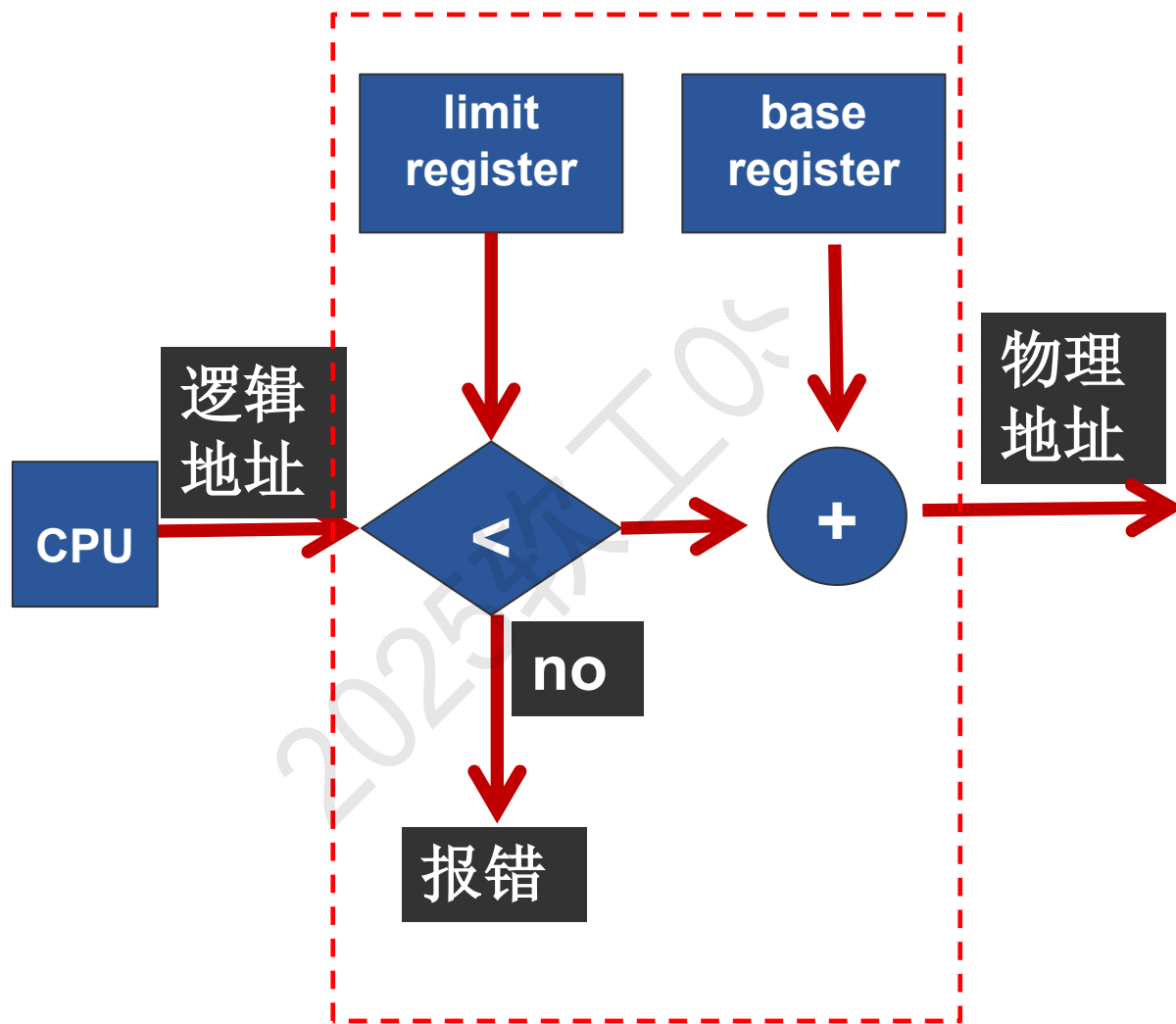
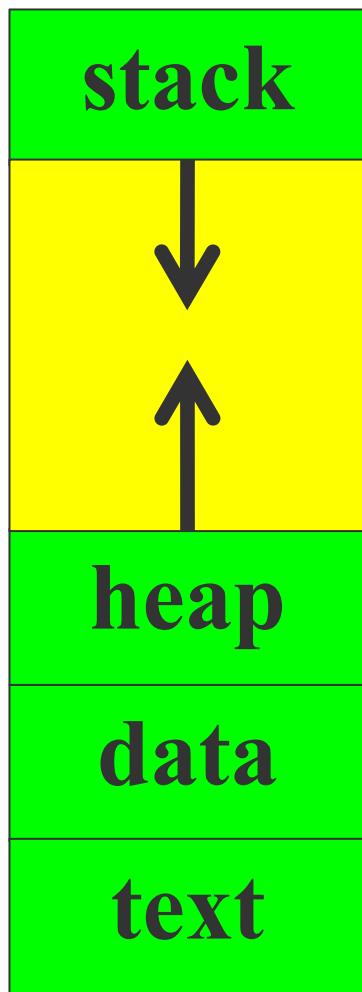
- 首次适应
- 循环首次适应
- 最佳适应
- 最坏适应 (why? )

## ● 碎片与紧凑 (compaction)

- 运行一段时间后会生成碎片
- 紧凑需要消耗一定时间

## ● 思考：malloc的实现是否类似？

# 可变分区下逻辑地址与物理地址的转换



(一个简单的)

“紧凑”操作时物理地址会发生改变，怎么办？

# 连续分配存储管理总结

- 单一连续分配
- 固定分区分配
- 可变分区分配
- 优点
  - 简单、较少的硬件支持
- 缺点
  - 碎片问题，紧凑带来的新问题
  - 无法支持动态链接（代际技术鸿沟）

为什么要连续分配?

可以离散地分配吗?

2025软工105

# 存储器管理

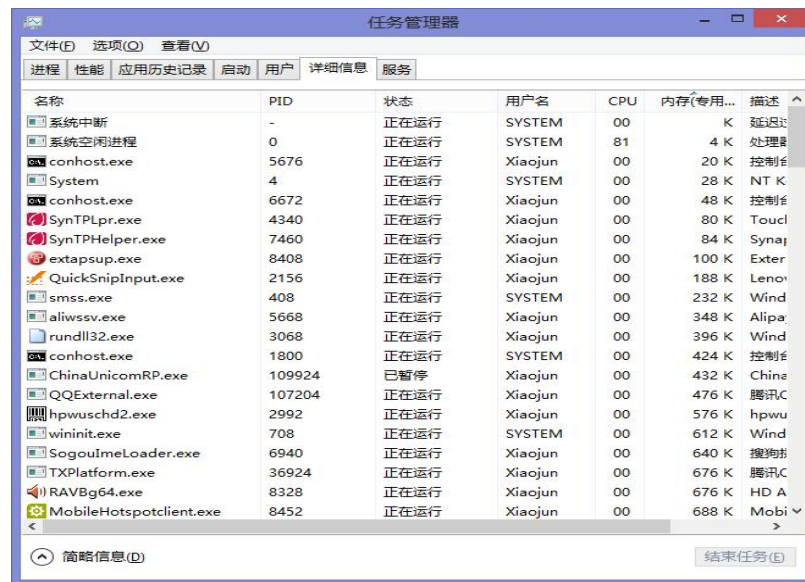
- 内存管理的需求
- 物理内存分配方案
  - 连续分配存储管理
  - 分页存储管理
  - 分段存储管理
- 虚拟存储器

# 分页存储管理



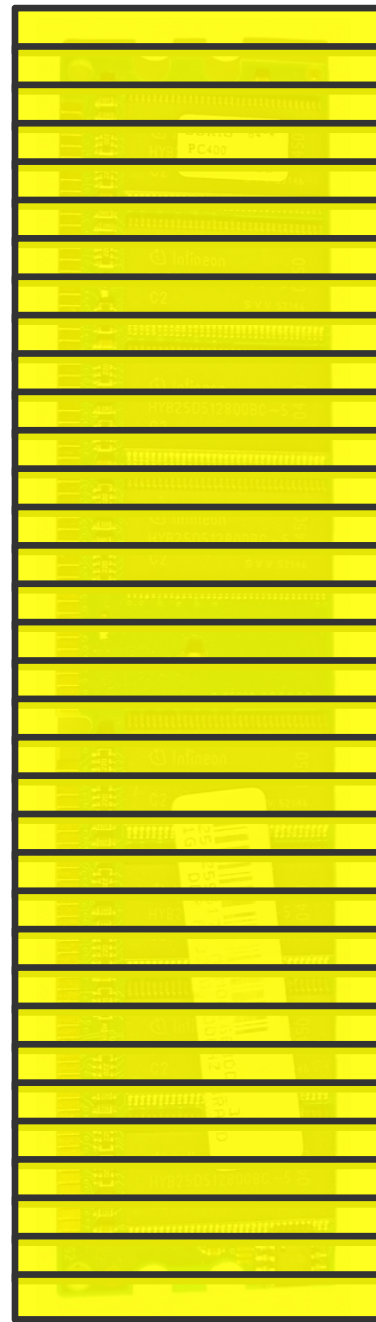
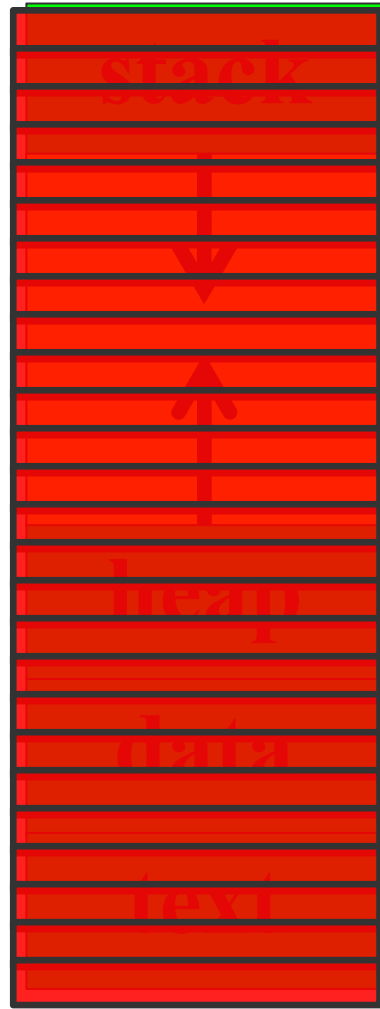
# 页面与页框

- 逻辑地址空间划分为多个页面 (page)
- 物理地址空间划分为多个页框 (page frame)  
，或物理块
- 页面大小与页框大小一致
- 进程请求页面数应不超过系统剩余页框数
- 页面大小设置
  - 常用大小：4KB
  - 页内碎片



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It displays a list of running processes with columns for Name, PID, Status, Username, CPU, Memory, and Description. The processes listed include system processes like 'System Idle Process' and 'smss.exe', as well as user applications like 'conhost.exe', 'SynTPHelper.exe', and 'QuickSnipInput.exe'.

名称	PID	状态	用户名	CPU	内存(专用...)	描述
系统中断	-	正在运行	SYSTEM	00	4 K	延迟...
系统空闲进程	0	正在运行	SYSTEM	81	4 K	处理...
conhost.exe	5676	正在运行	Xiaojun	00	20 K	控制...
System	4	正在运行	SYSTEM	00	28 K	NT K...
conhost.exe	6672	正在运行	Xiaojun	00	48 K	控制...
SynTPLpr.exe	4340	正在运行	Xiaojun	00	80 K	Toucl...
SynTPHelper.exe	7460	正在运行	Xiaojun	00	84 K	Syna...
extapsup.exe	8408	正在运行	Xiaojun	00	100 K	Exter...
QuickSnipInput.exe	2156	正在运行	Xiaojun	00	188 K	Leno...
smss.exe	408	正在运行	SYSTEM	00	232 K	Wind...
aliwssv.exe	5668	正在运行	Xiaojun	00	348 K	Alipa...
rundll32.exe	3068	正在运行	Xiaojun	00	396 K	Wind...
conhost.exe	1800	正在运行	SYSTEM	00	424 K	控制...
ChinaUnicomRP.exe	109924	已暂停	Xiaojun	00	432 K	China...
QQExternal.exe	107204	正在运行	Xiaojun	00	476 K	腾讯C...
hpwuschd2.exe	2992	正在运行	Xiaojun	00	576 K	hpwu...
wininit.exe	708	正在运行	SYSTEM	00	612 K	Wind...
SogouImeLoader.exe	6940	正在运行	Xiaojun	00	640 K	搜狗...
TXPlatform.exe	36924	正在运行	Xiaojun	00	676 K	腾讯C...
RAVBg64.exe	8328	正在运行	Xiaojun	00	676 K	HD A...
MobileHotspotclient.exe	8452	正在运行	Xiaojun	00	688 K	Mobi...



# 物理内存管理

- 假设物理内存512MB，页框大小为4KB；内核占用128MB，如何管理剩余的空闲内存？
- 用什么数据结构管理空闲页框？
  - 数组（特例，bitmap）：存在什么地方？
  - 链表：存在什么地方？
- 内部碎片问题

**逻辑地址怎么转换为物理地址？**

# 页表（物理内存）

- 记录页面和页框的对应关系
- 每个进程都有一张页表（why?）
  - 可用页号作为数组下标（优点？ 缺点？）

0

1

2

3

4

...

页框号	其他信息
3	读
2	读
5	读写
16	读写
8	读
...	

# xv6中的页表项

- mmu.h
- 与课程联系:如何防止代码段被修改?

```
// Page table/directory entry flags.
#define PTE_P          0x001    // Present
#define PTE_W          0x002    // Writeable
#define PTE_U          0x004    // User
#define PTE_PWT        0x008    // Write-Through
#define PTE_PCD        0x010    // Cache-Disable
#define PTE_A          0x020    // Accessed
#define PTE_D          0x040    // Dirty
#define PTE_PS         0x080    // Page Size
#define PTE_MBZ        0x180    // Bits must be zero

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
#define PTE_FLAGS(pte)  ((uint)(pte) & 0xFFF)

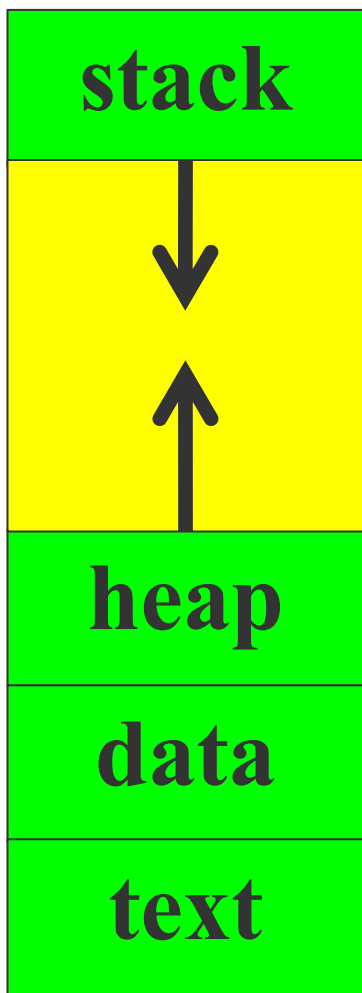
#ifndef __ASSEMBLER__
typedef uint pte_t;
```



Page table and page directory entries are identical except for the D bit.

P - Present  
W - Writable  
U - User  
WT - 1=Write-through, 0=Write-back  
CD - Cache Disabled  
A - Accessed  
D - Dirty (0 in page directory)  
AVL - Available for system use

# 查页表



页号

0

1

2

3

4

...

页框号

3

2

5

16

8

...

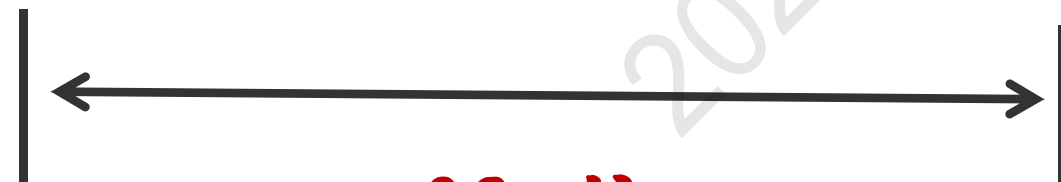


逻辑地址32位



页号

位移量

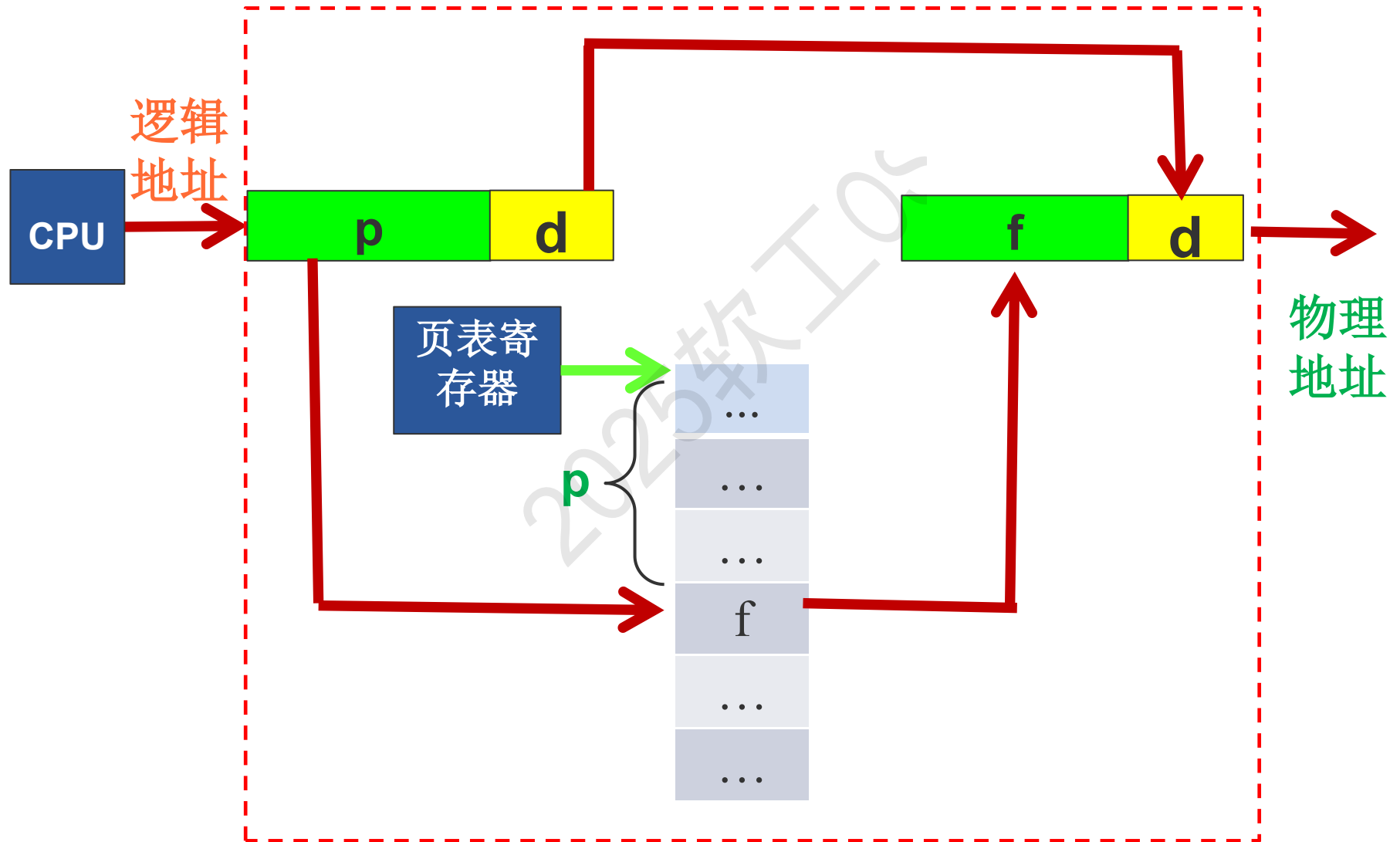


20 位



# 逻辑地址与物理地址的转换

MMU



# 逻辑地址与物理地址的转换

MMU



# 课程回顾：存储器管理

- 内存管理的需求
- 物理内存分配方案
  - 连续分配存储管理
  - 分页存储管理（页面、页框、页表、地址转换）
  - 分段存储管理
- 虚拟存储器

## 分页方法有两个缺点

- 太慢（页表放在内存中）访问内存两次
- 页表多大？若1个页框装不下页表，会造成什么后果？

# 提高速度：快表

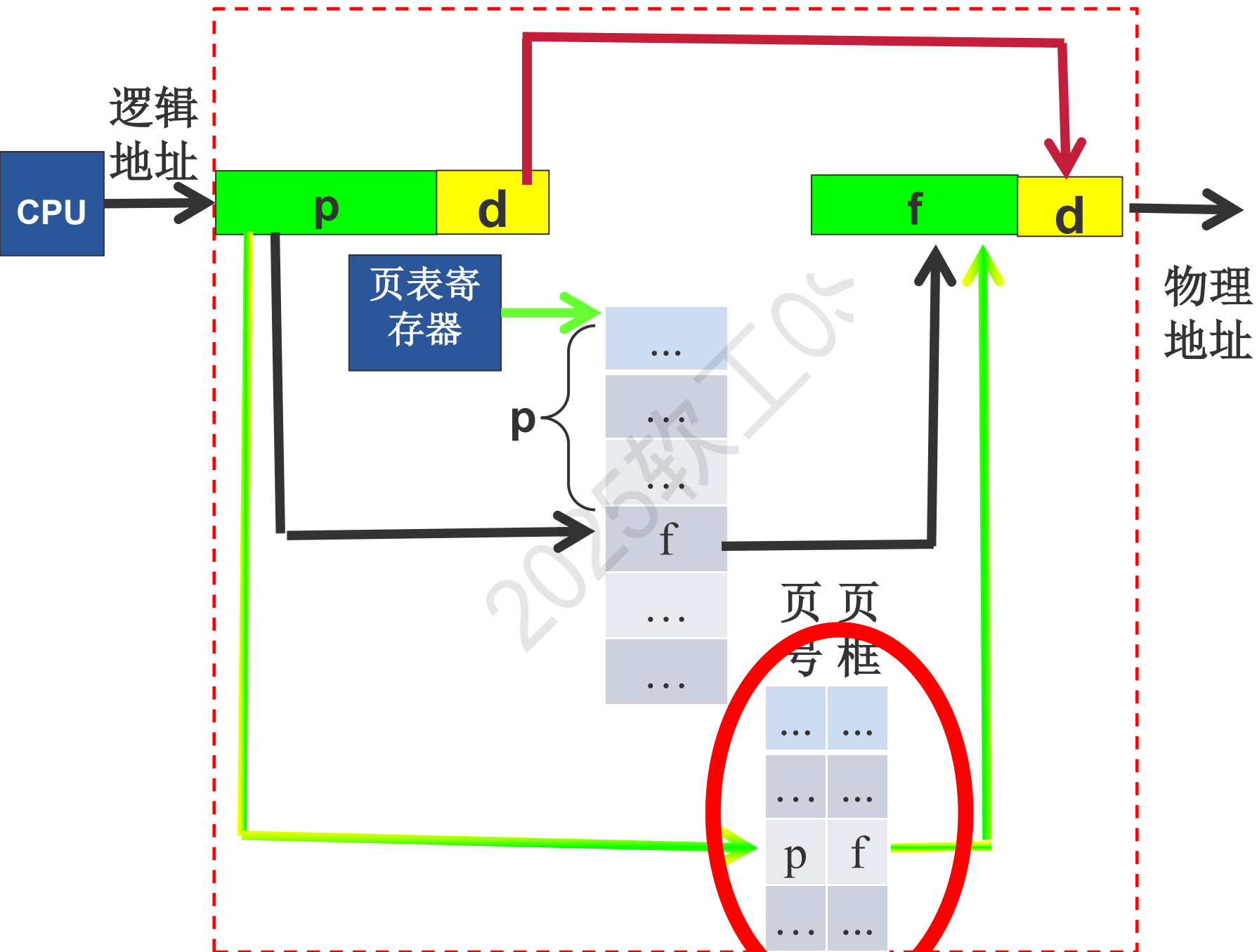
- 快表是硬件

- 别名有“联想寄存器” associative memory
- TLB, Translation Look aside Buffer

- 有了快表后，MMU的工作流程：

2025软工

# MMU



# 有了快表后的内存访问时间

## ● 假设：

- 查快表和查内存串行执行，即，快表不命中时，才去查内存
- 【实际系统中是并行执行的】
- 【建议解题时注明TLB按串行还是并行执行】

## ● 访存的平均时间是

命中率 \* (快表+访存) + (1-命中率) \* (快表+访存+访存)

## 例题（2024期末考试题）

某机器采用分页存储管理，单级页表，假设一次TLB访问10ns，一次内存访问50ns，要求达到平均65ns的内存访问时间，需要TLB命中率为多少？

设为 $x$ ，则

$$x \cdot (10 + 50) + (1 - x) \cdot (10 + 50 + 50) = 65 \quad \text{【2分】}$$

求得 $x = 0.9$  【1分】



# 页表需要多少个页框？（方法一）

## ● 页表多大？

- 页表项多少个？ $2^{20}$ （跟什么有关？页面数量、逻辑地址空间大小）
- 每个页表项多大？ $\geq 20$  bit（跟什么有关？页框号长度、物理地址空间大小）
- 页表大小  $\geq 2^{20} * 20 / 8 \text{ B} = 2.5 * 2^{20} \text{ B}$

## ● 至少需要多少页框？

- 页框多大？ $2^{12} \text{ B}$
- 最少需要多少页框？ $2.5 * 2^{20} / 2^{12} = 640$

## ● 【以上计算过程，需要理解，熟练掌握】

# 页表需要多少个页框？（方法二）

## ● 一个页框最多可以容纳多少个页表项？

- 页框多大？ $4\text{KB}=2^{12}\text{B}$
- 每个页表项多大？ $\geq 20\text{ bit}$ （跟什么有关？页框号长度、物理地址空间大小）
- 能装的页表项数  $\leq 2^{12} \times 8 / 20 = 1638$

## ● 一个页表至少需要多少个页框？

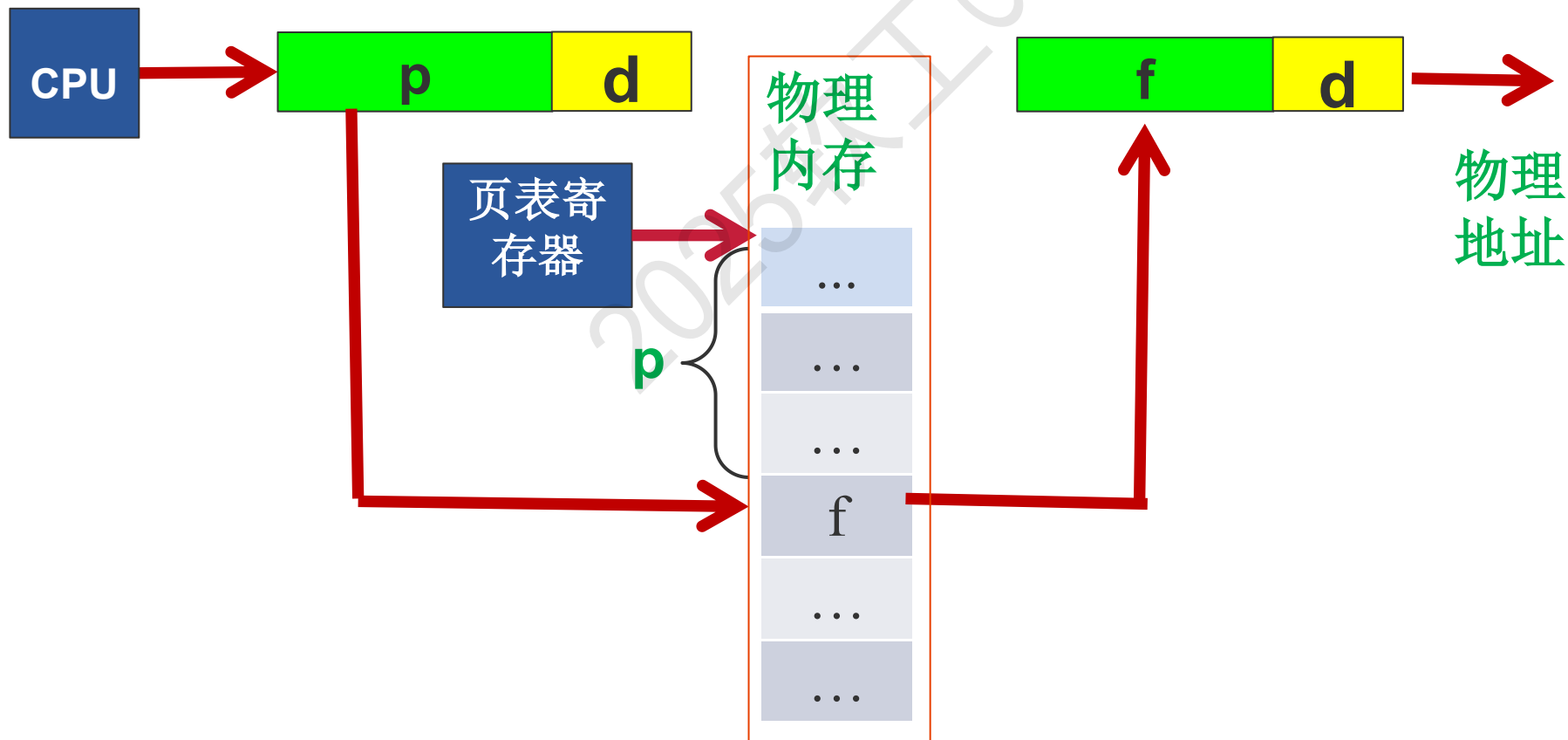
- 共有页表项数？ $2^{20}$ （跟什么有关？页面数量、逻辑地址空间大小）
- 至少需要多少页框？ $2^{20} / 1638 = 640$

## ● 【以上计算过程，需要理解，熟练掌握】

# 一个页框装不下，会有什么问题？

- 多个页框存储页表
- MMU无法进行地址转换！

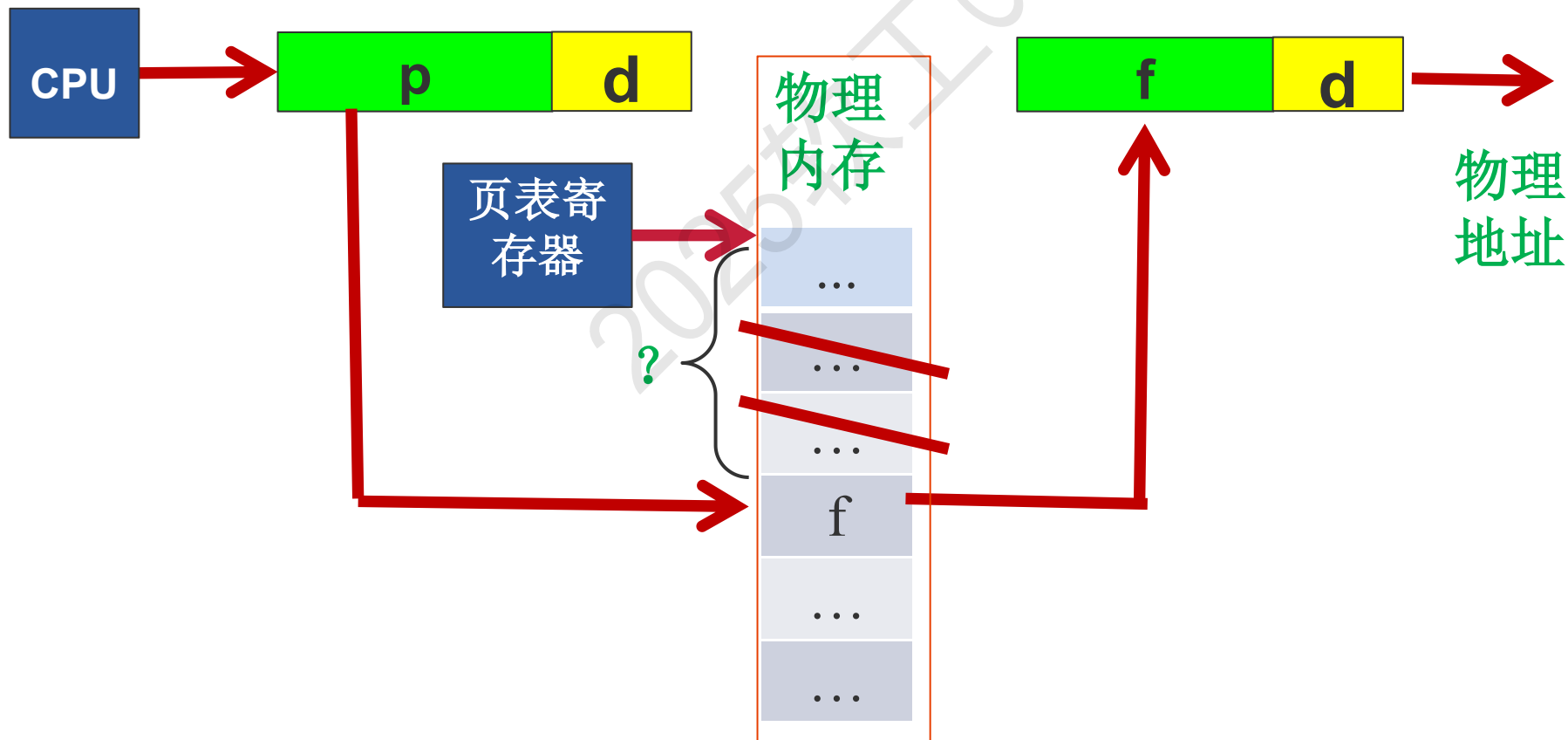
➤ 原本方法：页表基址寄存器 + (页号 \* 页表项大小)



# 一个页框装不下，会有什么问题？

- 多个页框存储页表
- MMU无法进行地址转换！

➤ 原本方法：页表基址寄存器 + (页号 \* 页表项大小)



## 例题（2024期末考试）

某x86机器采用分页存储管理，32位逻辑地址，物理内存为1GB，页面大小1KB，页表项大小为4B。如果使用单级页表，一个进程的页表最多需要多少个物理页框存储？如果将页表存储在不连续的页框中会有什么问题？

页表大小：页表项数  $2^{32}/2^{10}=2^{22}$

页表大小  $2^{22}*4B$

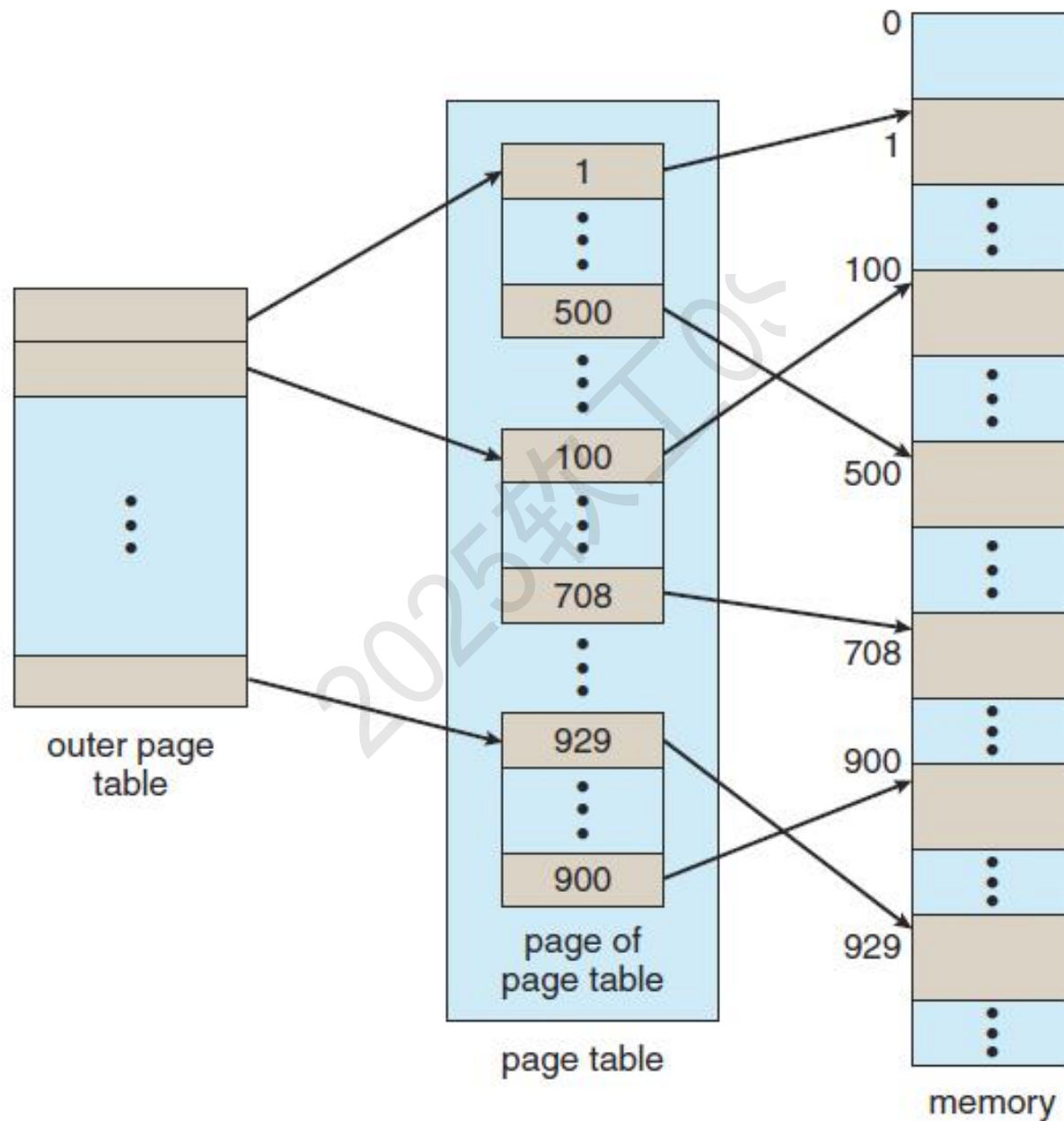
页框数  $2^{22}*4B/2^{10}B=2^{14}$  或16384 【2分】

MMU无法正常翻译地址 【2分】

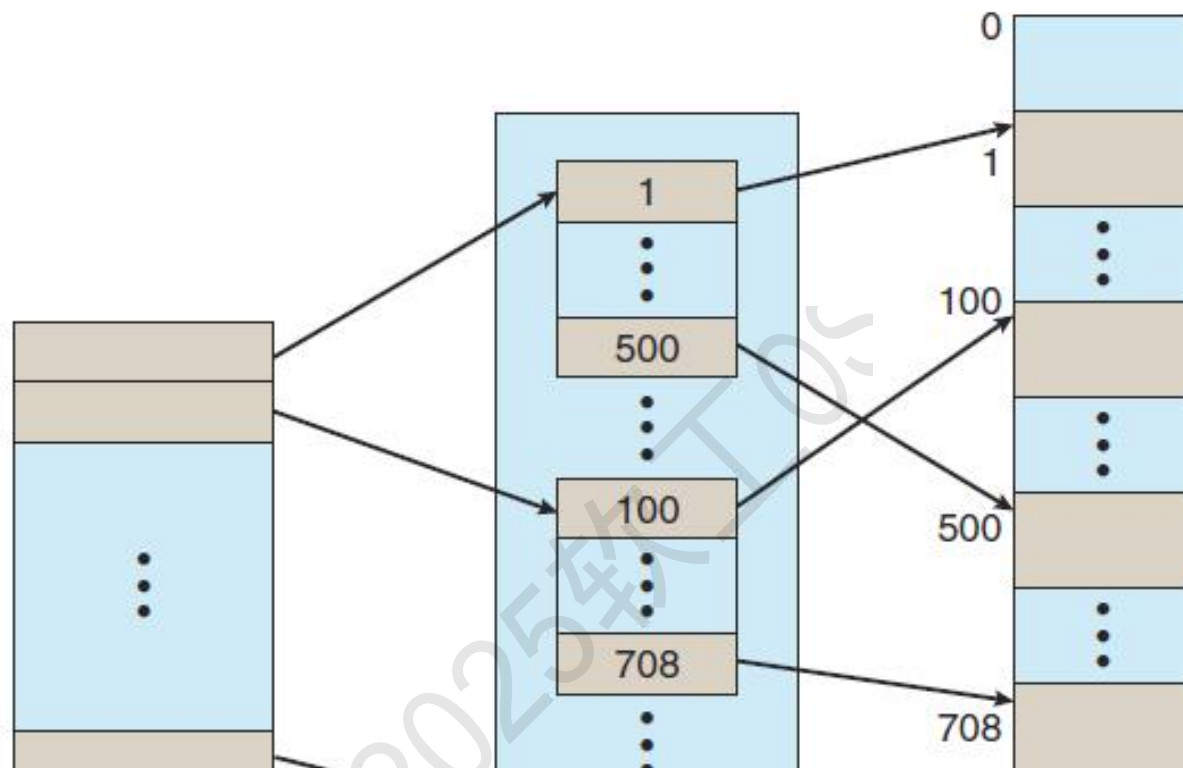
## 解决办法：

- 大页框，系统中有两类页框，一类比较大，能装下页表
- 二级页表，组原上学习的“页目录”来源于此

# 二级页表：增加页目录



# 二级页表：增加页目录

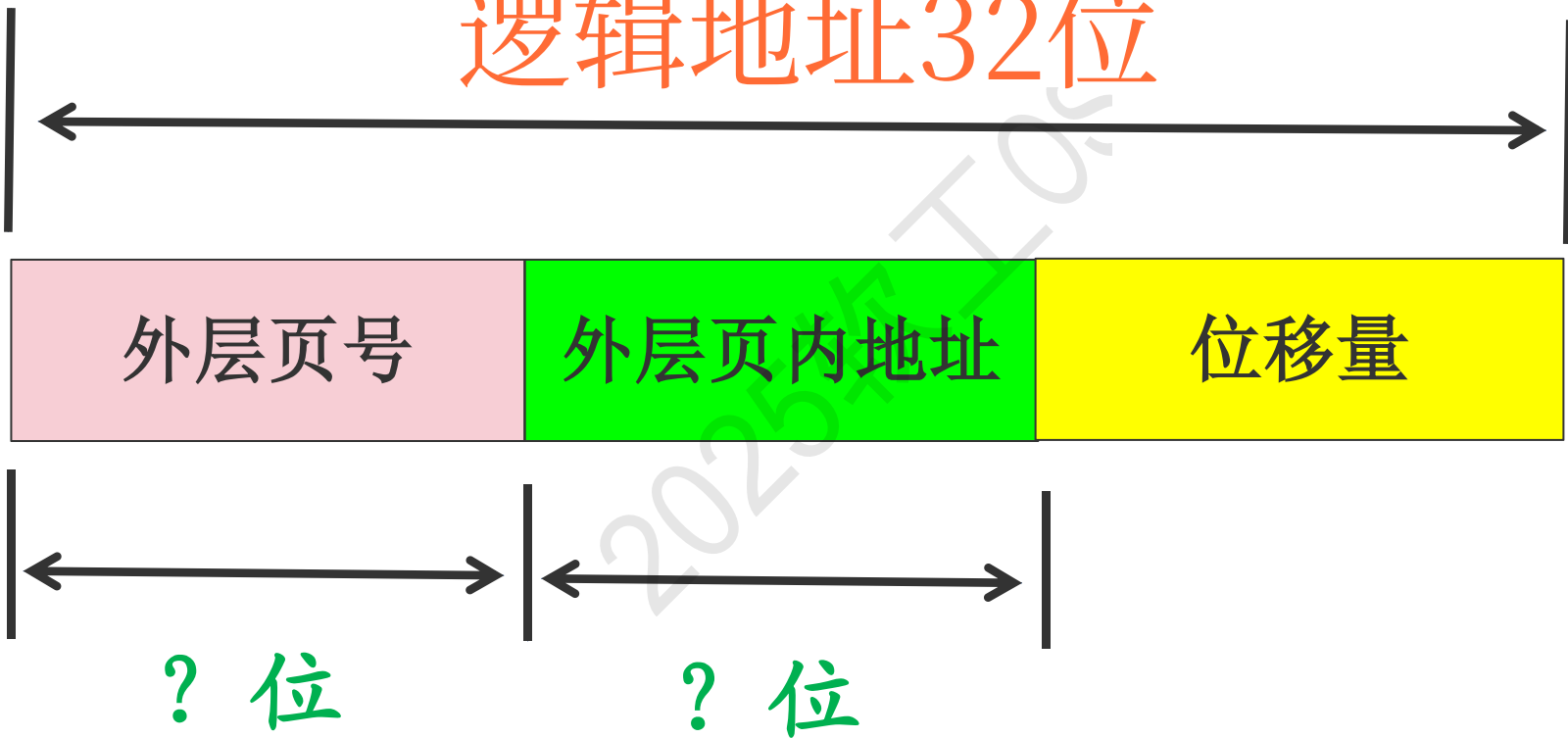


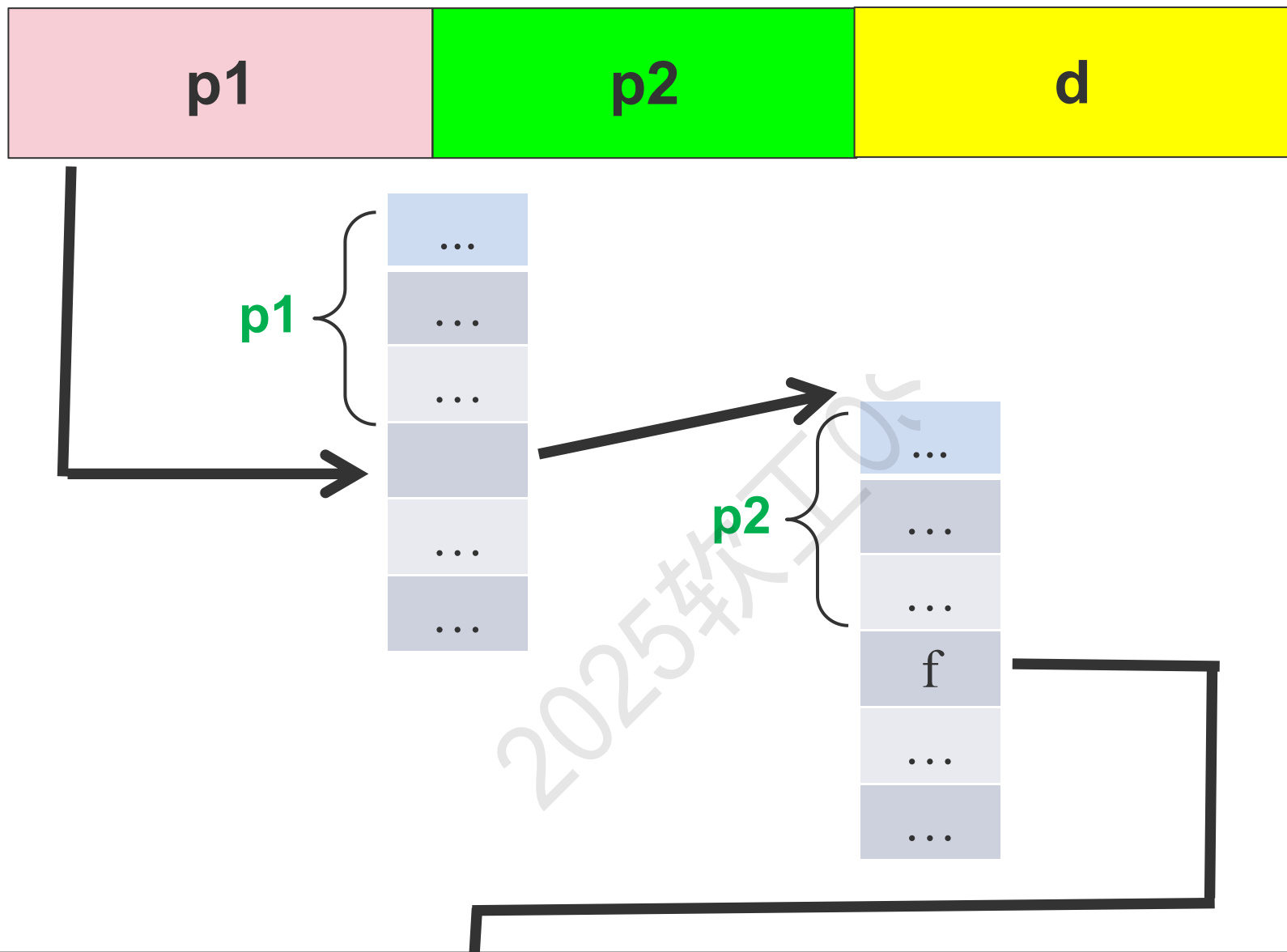
二级页表是二维数组吗？

不是！

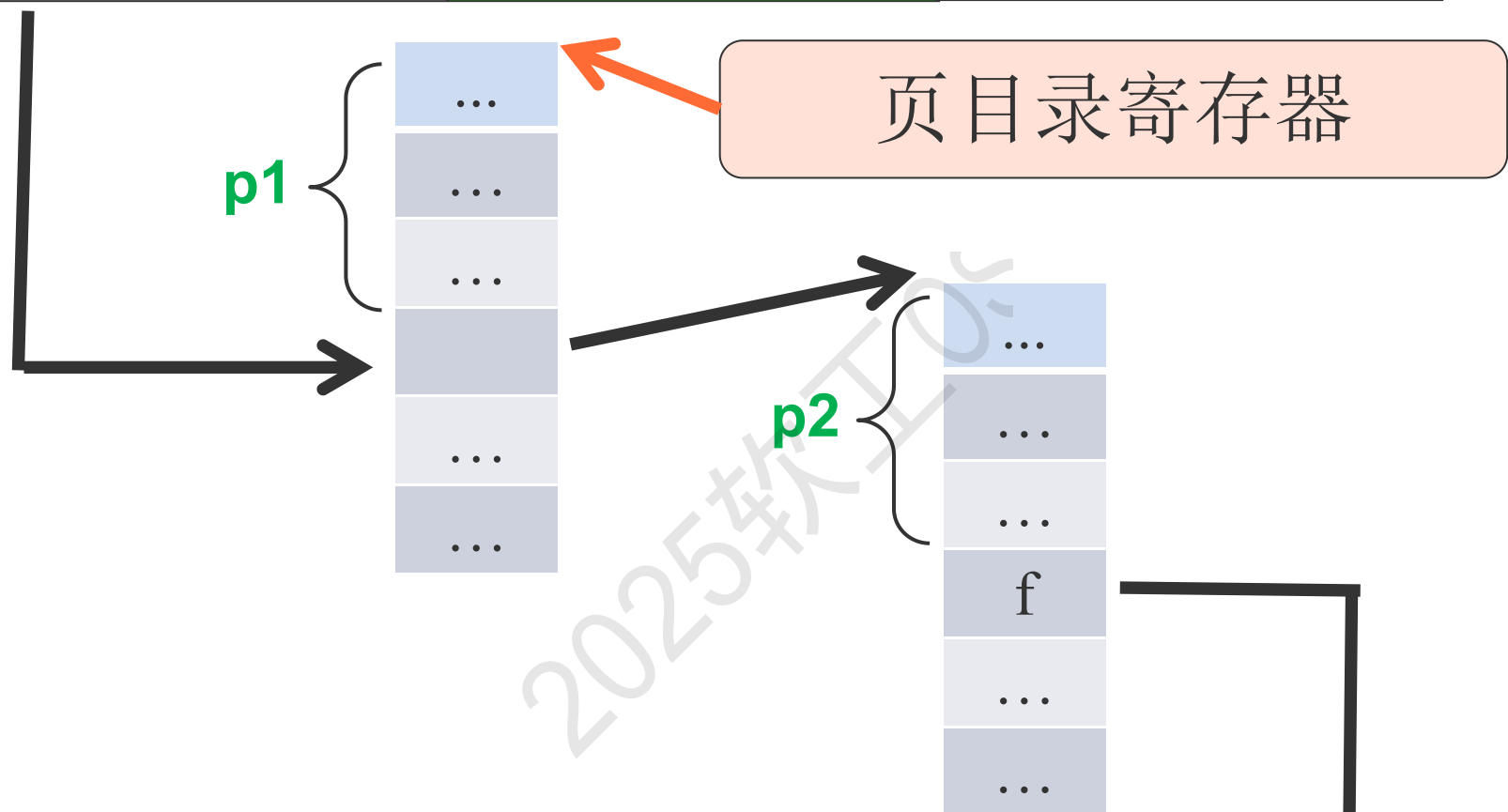
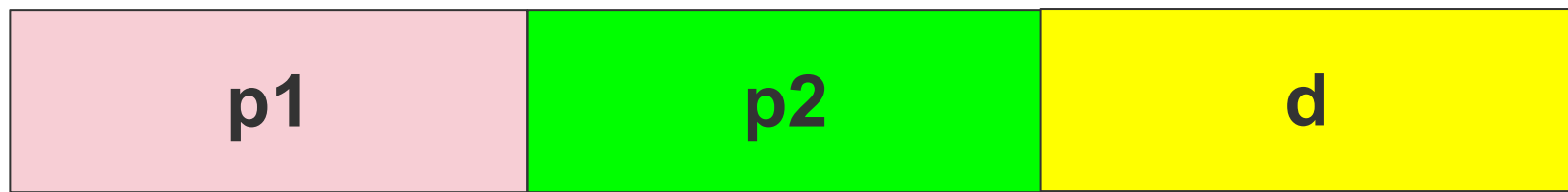


逻辑地址32位





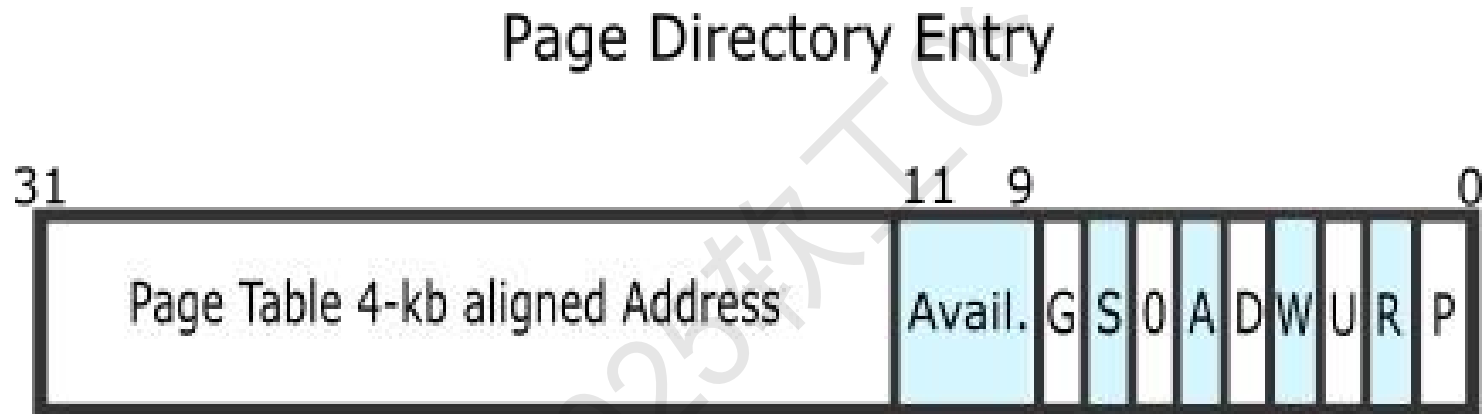
图中缺少一个寄存器，应该是什么，在哪？



# 二级页表的其他好处

## ● 节省内存 (尤其地址空间中合法地址不多时)

➤ 页目录项中指明对应页表是否存在



G - Ignored  
S - Page Size (0 for 4kb)  
A - Accessed  
D - Cache Disabled  
W - Write Through  
U - User\Supervisor  
R - Read\Write  
P - Present

页表、页目录表中的页框号均是物理地址，  
而CPU产生的每个地址均是逻辑地址。页  
目录的逻辑地址存在pcb的pgdir指针中。

假设：

- $v2p(x)$  可以将逻辑地址转换为物理地址
- $p2v(x)$  可以将物理地址转换为逻辑地址
- 每个页表项、页目录项前20位为页框号
- 页目录的逻辑地址为 `char * pgdir`

问：如果OS想将页面0对应的页框号修改为4，  
请问c语言代码怎么写？

提示：假如物理地址就是逻辑地址，怎么写？

//pgdir[0]的前20位是页表地址

➤ ptable=pgdir[0]&0xFF FF F0 00

//ptable[0]的前20位设置为4

➤ ptable[0]=(0x4<<12) | (ptable[0]&0x F FF)

哪些是物理地址，需要转换成逻辑地址？

提示：假如物理地址就是逻辑地址，怎么写？

//pgdir[0]的前20位是页表地址

➤ ptable=pgdir[0]&0xFF FF F0 00

//ptable[0]的前20位设置为4

➤ ptable[0]=(0x4<<12) | (ptable[0]&0x F FF)

哪些是物理地址，需要转换成逻辑地址？

//pgdir[0]的前20位是页表地址

➤ ptable=pgdir[0]&0xFF FF F0 00

➤ vtable=p2v(ptable)

//ptable[0]的前20位设置为4

➤ vtable[0]=(0x4<<12) | (vtable[0]&0x F FF)

# 页表大、耗内存？ 反置页表

- 虚地址空间大 $\Rightarrow$ 页面多 $\Rightarrow$ 页表项多 $\Rightarrow$ 页表大
- 页表的“逆函数”
  - 反置页表以页框作为下标，内容包括  
(进程号，页面号)
  - 物理内存少 $\Rightarrow$ 页框少 $\Rightarrow$ 反置页表项数少
- 地址如何转换？

Inverted page tables are used for example on the PowerPC,  
the UltraSPARC and the IA-64 architecture. [1]      Wikipedia



# 分页存储管理下的若干技术

- 进程间的数据共享

- 共享内存
- 如何实现？

- 进程间的代码共享

- `fork ()` 执行后父进程子进程代码一致

- 支持动态链接

- 节省代码

# 存储器管理

- 内存管理的需求
- 物理内存分配方案
  - 连续分配存储管理
  - 分页存储管理
  - 分段存储管理
- 虚拟存储器

# 分段存储管理

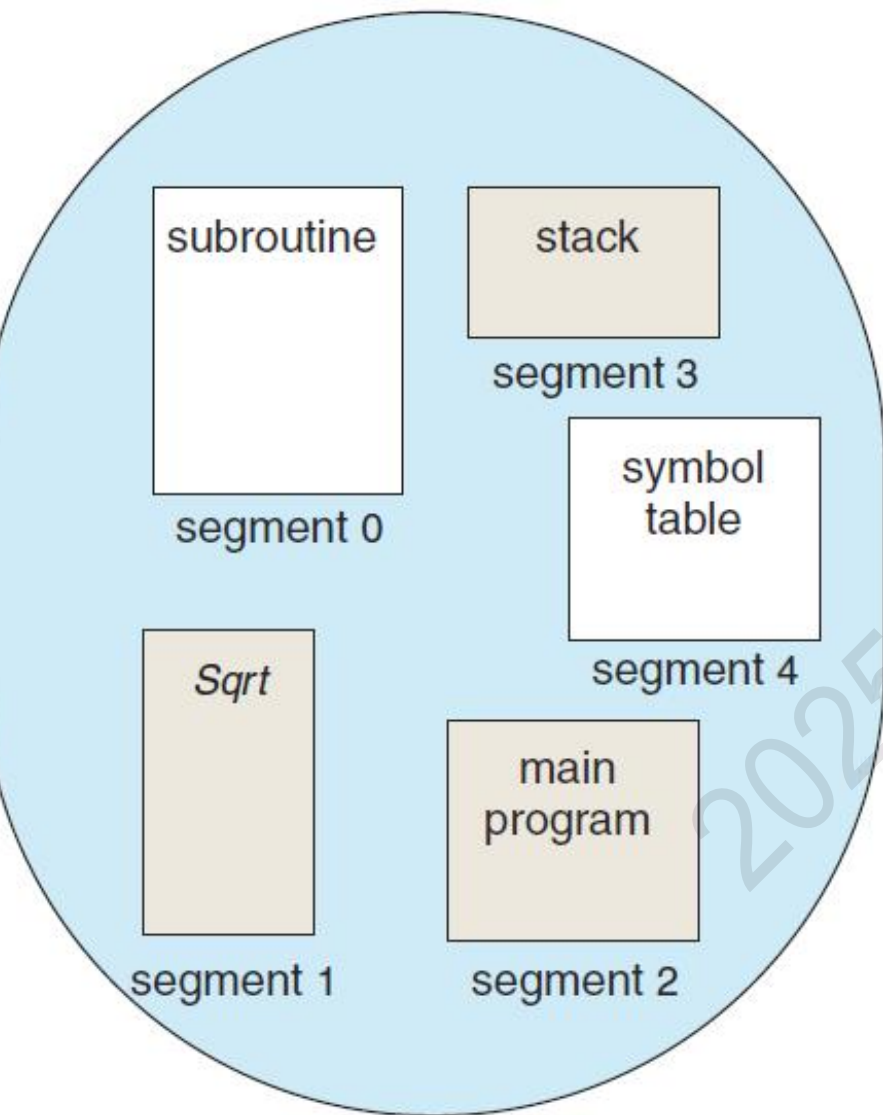
# 分段存储管理

- Segmentation

- Intel 8086 处理器为了解决16位地址线寻址空间太小的问题而引入
- 基于兼容性考虑，一些现代的CPU支持分段

- 存储管理方式从固定分区—>动态分区—>分页存储管理, 主要动力是提高内存利用率。

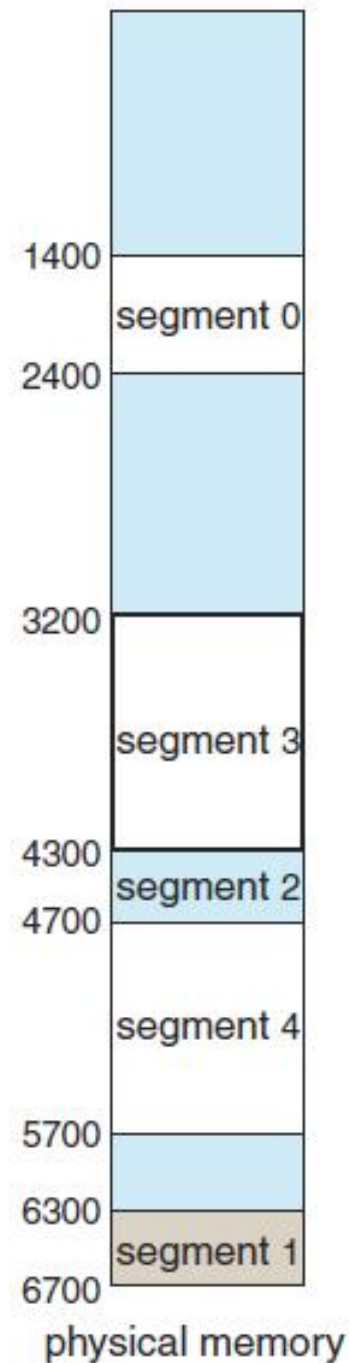
- 分段存储管理方式则主要是为了满足用户(程序员)的要求。



logical address space

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



# 段表

- 分配方式:

- 每个段分配一个连续的分区，而进程中的各个段可以离散地放入内存中不同的分区中。

- 段表:

- 为能从物理内存中找出每个逻辑段所对应的位置，系统为每个进程建立的一张段映射表。

- 作用:

- 段表常放在内存中，实现了逻辑段到物理内存区的映射。

## 若剩余物理内存小于进程所需内存

- 对换（总物理内存大于进程所需内存时）
- 虚存（任何情况都可以使用）

# 对换 (swapping)

- 磁盘（外存）上开辟一块空间，称为对换区，当内存不足时，选择一些进程换出到磁盘
- 对换区与普通磁盘文件的管理方式不同
  - 连续存储，而普通磁盘文件可以离散存储
  - Linux在安装时需要预留一块磁盘区域用作对换
- 速度慢（相对于内存）
  - 假设磁盘读取速度为50M/s，则一个100M的进程换进或换出时需要2秒
- “对换”特指将进程整体换到磁盘；现代操作系统使用的为“部分对换”



# 内存管理要求（2025考研大纲）

- 内存管理的基本概念

- 逻辑地址空间与物理地址空间，地址变换，内存共享，内存保护，内存分配与回收

- 连续分配管理方式

- 页式管理

- 段式管理

- 段页式管理