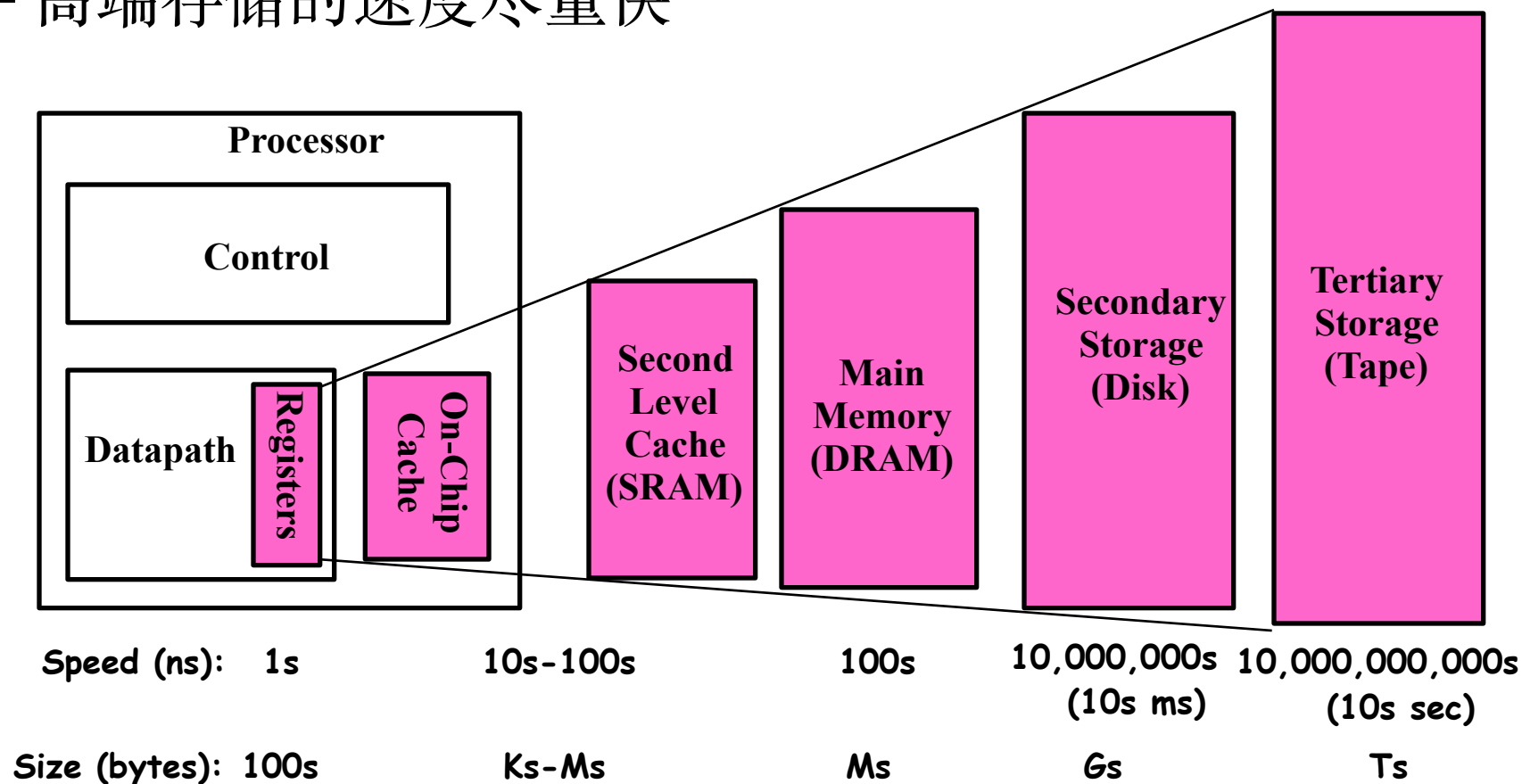


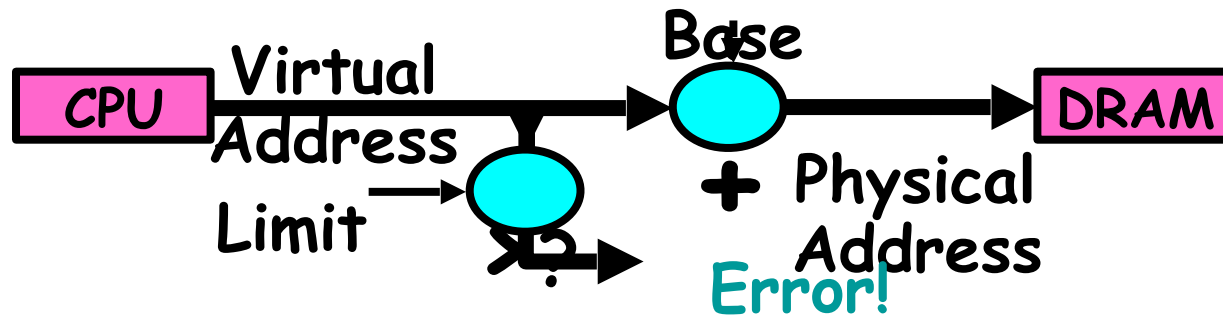
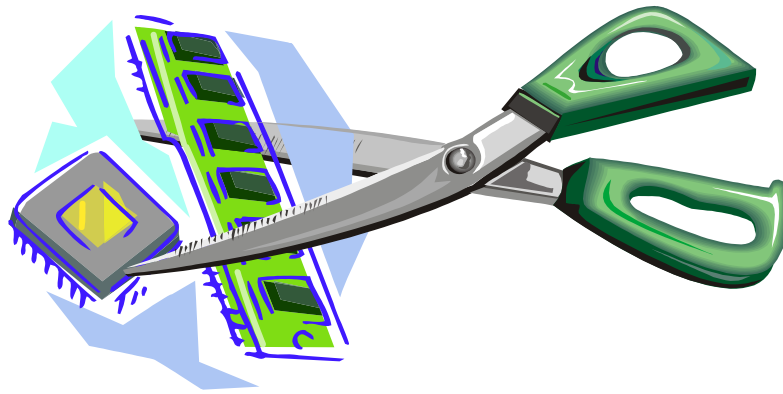
内存管理

- 背景
- 主存管理介绍
- 连续存储空间管理
- 分页式存储管理
- 分段式存储管理
- 虚拟存储管理

- 局部性原理：时间/空间局部性

- 低端存储的容量尽量大
- 高端存储的速度尽量快





- 内存：由很大一组字/字节所组成，每个字/字节都有自己的地址
 - 输入队列 – 磁盘上等待进入内存并执行的进程的集合
- 内存共享时需要考虑的问题
 - 进程和内核的工作状态取决于存储在内存/寄存器中的相关数据
 - 不同的进程/线程控制部分不能使用内存的同一部分
 - Physics: 不同部分的数据不能使用内存的同一地址
 - 有时不同线程的内存资源会private化 (protection)

存储管理的功能

- **地址变换：**将程序地址空间中使用的逻辑地址变换成主存中的地址的过程，又称地址重定位。地址变换的功能就是要建立虚实地址的对应关系。
- **主存分配：**按照一定的算法把某一空闲的主存区分配给作业或进程。
- **存储保护：**保证用户程序(或进程映象)在各自的存储区域内操作，互不干扰。
- **虚拟存储：**使用户程序的大小和结构不受主存容量和结构的限制，即使在用户程序比实际主存容量还要大的情况下，程序也能正确运行。

地址变换

- 逻辑地址（虚拟地址）- 用户编程序时所用的地址（或称程序地址、虚地址），基本单位可与内存的基本单位相同，也可以不相同
 - 逻辑地址空间- 由程序所生成的所有逻辑地址的集合；可以是一维线性空间，也可以是多维空间
- 物理地址- 把内存分成若干个大小相等的存储单元，每个单元给一个编号，这个编号称为内存地址，即内存单元所用的地址
 - 物理地址空间- 与逻辑地址相对应的内存中所有物理地址的集合，一维的线性空间

地址变换

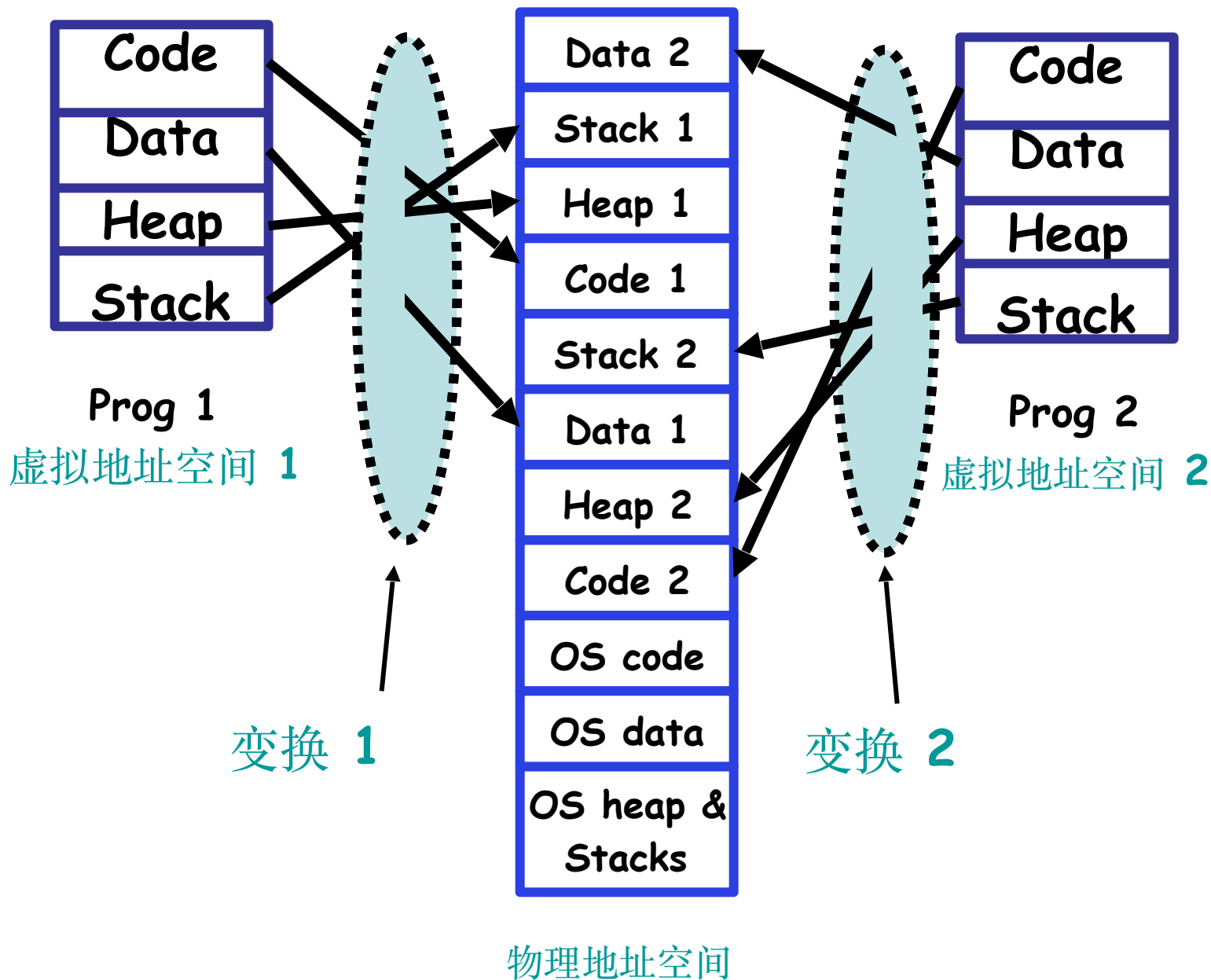
- 用户程序看不见真正的物理地址。用户只生成逻辑地址，且认为进程的地址空间为0到max。物理地址范围从R+0到R+max，R为基地址
- **地址变换**- 将程序地址空间中使用的逻辑地址变换成内存中的物理地址的过程。由内存管理单元（MMU）来完成。

内存保护

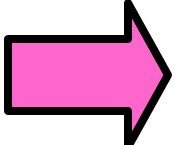
- 内存保护：保护操作系统不受用户进程影响，保护用户进程不受其它用户进程影响。措施：
 - **上下界保护**：下界寄存器存放程序装入内存后的开始地址（首址）。上界寄存器存放程序装入内存后的末地址
 - 物理地址
 - **基址、限长寄存器保护**：基址寄存器存放程序装入内存后的起始地址。限长寄存器存放程序装入内存后的最大长度
 - 逻辑地址

对于合法的访问地址，这两者的效率是相同的，对不合法的访问地址来说，上下界存储保护浪费的CPU时间相对来说要多些。

例子



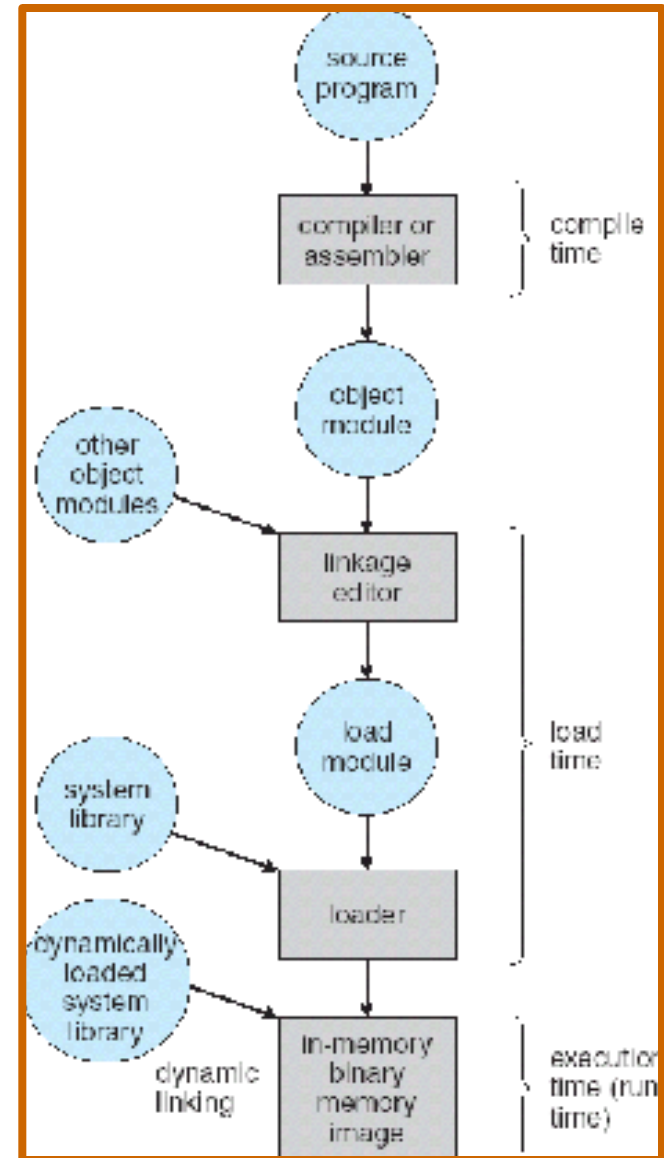
例子：指令和数据地址变换

<code>data1: dw 32</code>		<code>0x300</code>	<code>00000020</code>
<code>start: lw ... r1, 0(data1)</code>		<code>0x900</code>	<code>8C2000C0</code>
<code>jal checkit</code>		<code>0x904</code>	<code>0C000340</code>
<code>loop: addi r1, r1, -1</code>		<code>0x908</code>	<code>2021FFFF</code>
<code>bnz loop</code>		<code>0x90C</code>	<code>1420FFFF</code>
<code>checkit: ...</code>		<code>0xD00</code>	<code>...</code>

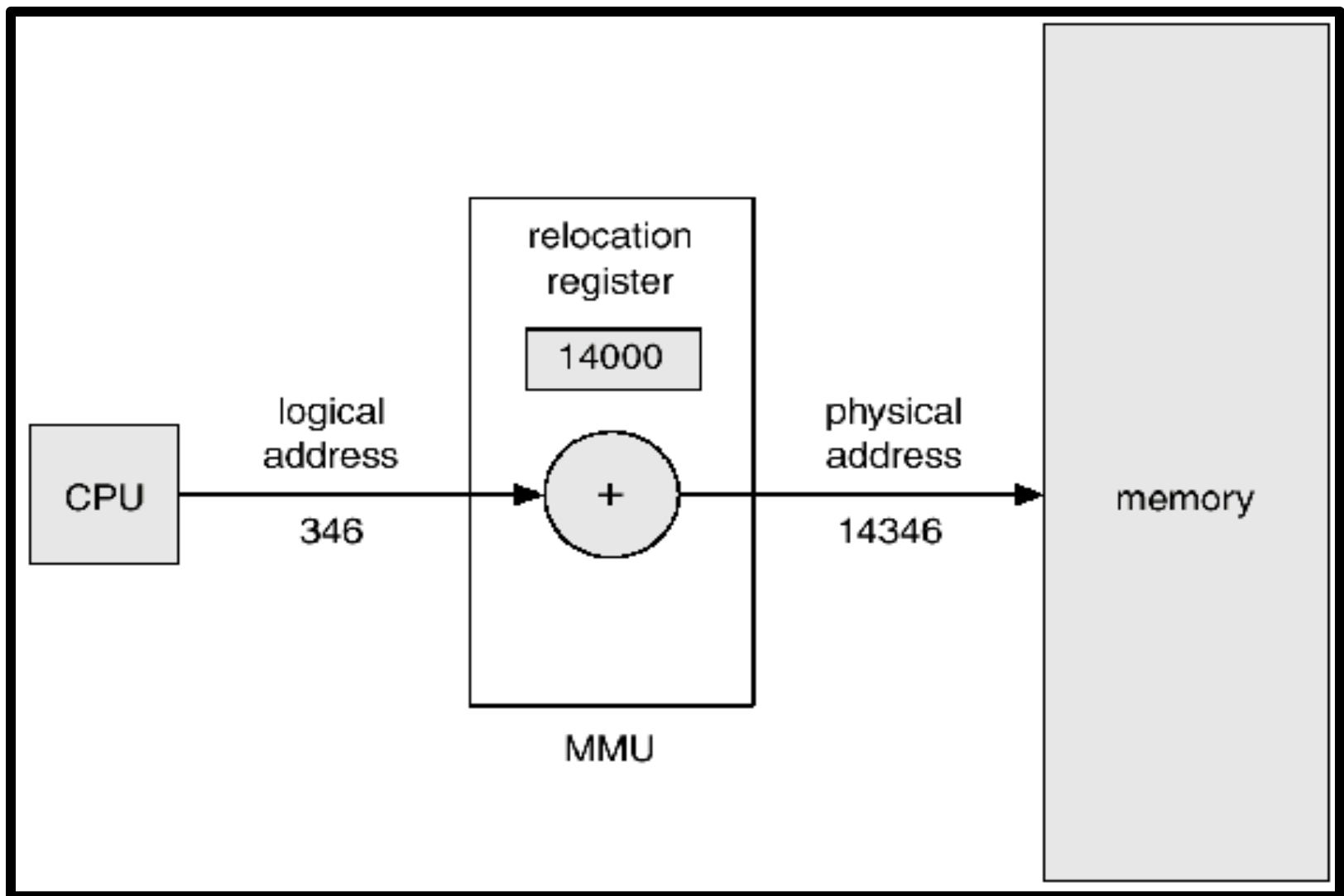
- Could we place `data1`, `start`, and/or `checkit` at different addresses?
 - Yes
 - When? Compile time/Load time/Execution time
- Related: which physical memory locations hold particular instructions or data?

进程在执行时，会访问内存中的指令和数据。将指令和数据捆绑到内存地址可以在以下步骤的任何一步中执行。

- 编译时确定地址变换关系：如果内存位置已知，可生成**绝对代码**；如果开始位置改变，需要重新编译代码。
- 加载时(静态地址变换)：如果存储位置在编译时不知道，则必须生成**可重定位代码**。
- 执行时(动态地址变换)：如果进程在执行时可以在内存中移动，则地址绑定要延迟到运行时。
 - 需要硬件对动态地址变换的支持，例如基址和限长寄存器
 - 基址寄存器这时称为重定位寄存器。用户进程所生成的地址在送交内存之前，都将加上重定位寄存器的值。



使用重定位寄存器的动态重定位



地址变换方式

- **编程或编译时确定地址变换关系：**编程时确定虚- 实地址的关系是指在用机器指令编程时，程序员直接按物理内存地址编程，这种程序在系统中是不能做任何移动的，否则就会出错。
- **静态地址变换：**静态地址变换是在程序装入内存时完成从逻辑地址到物理地址的转换的。在一些早期的系统中都有一个装入程序（加载程序），它负责将用户程序装入系统，并将用户程序中使用的访问内存的逻辑地址转换成物理地址。
 - **优点：**实现简单，不要硬件的支持。
 - **缺点：**程序一旦装入内存，移动就比较困难，有时间上的浪费。在程序装入内存时要将所有访问内存的地址转换
- **动态地址变换：**地址变换是在程序执行时由系统硬件完成从逻辑地址到物理地址的转换的。
 - 动态地址变换是由硬件执行时完成的，程序中不执行的程序就不做地址变换的工作，这样节省了CPU的时间。
 - 系统中设置了重定位寄存器。重定位寄存器的内容由操作系统用特权指令来设置，比较灵活。实现动态地址变换必须有硬件的支持，并有一定的执行时间延迟。现代计算机系统中都采用动态地址变换技术。

动态加载

- 一个子程序只有在调用时才被加载。
- 更好的内存空间利用率；不用的子程序不会被装入内存。
- 当需要大量的代码来处理不经常发生的事情时是非常有用的。
- 不需要操作系统的特别支持, 通过程序设计实现。

动态链接

- 链接被推迟到执行时期
- 二进制映像中对每个库程序的引用都有一个存根。
存根是小的代码片，用来定位合适的保留在内存中的库程序。
- 存根用子程序地址来替换自己，并开始执行子程序。
- 动态链接需要操作系统的帮助。操作系统需要检查子程序是否在进程的内存空间，或是允许多个进程访问同一内存地址。

动态地址变换

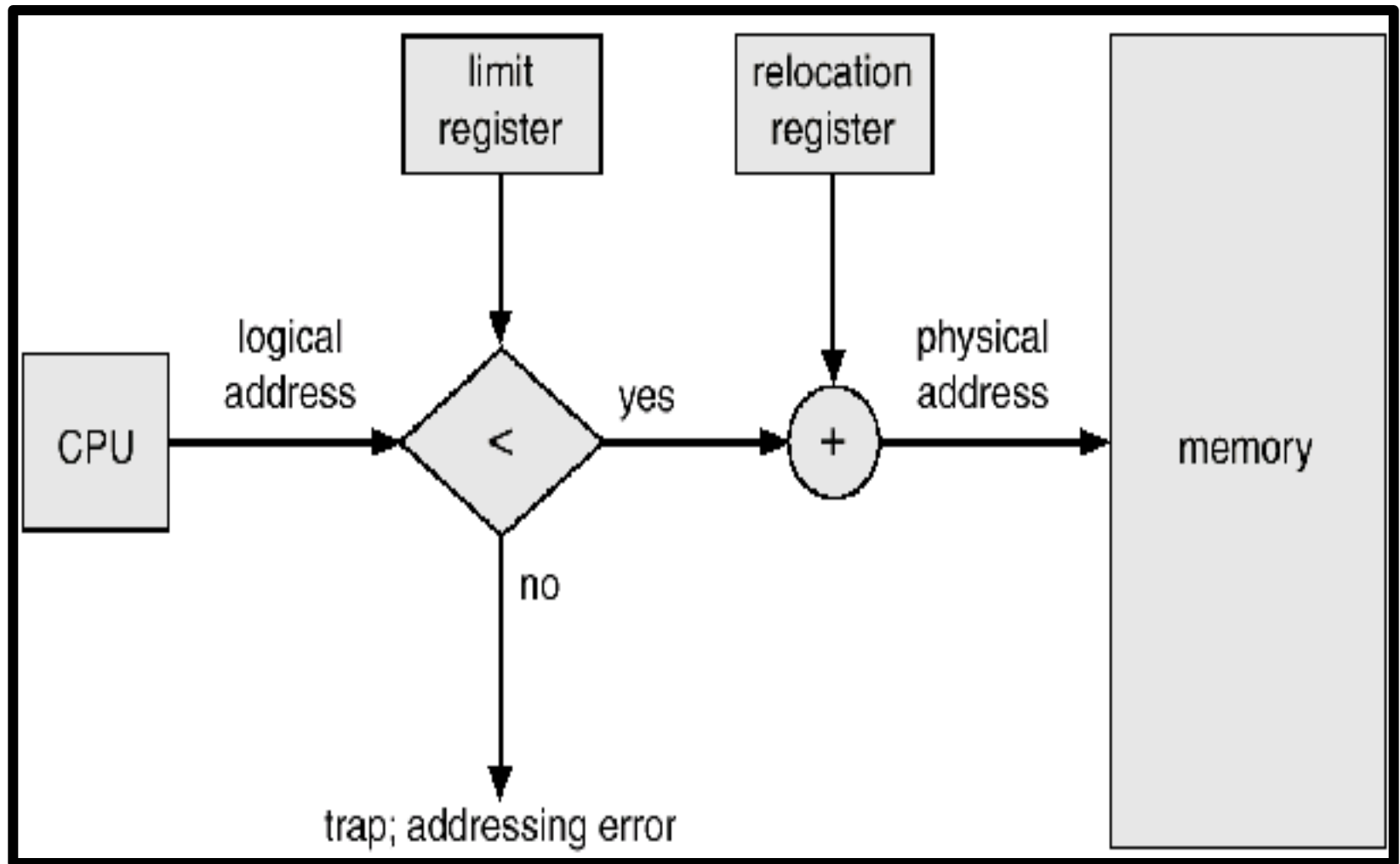
动态地址变换技术的优点：

- (1) 具有给一个用户程序任意分配内存区的能力；
- (2) 可实现虚拟存储；
- (3) 具有重新分配的能力；
- (4) 对于一个用户程序，可以分配到多个不同的存储区。

连续内存分配

- 主存通常被分为两部分
 - 用于驻留操作系统：为操作系统保留的部分，通常用中断向量保存在内存低端。
 - 用于用户进程：保存在内存高端。
- 每个程序（作业）占据主存中连续的空间，按管理方式的不同分为：
 - 单用户连续存储管理
 - 固定分区存储管理
 - 可变分区存储管理
- 采用连续内存分配时，每个进程位于一个连续的内存区域
 - 动态定位
 - 静态定位

基址和限长寄存器的硬件支持



连续存储空间管理

- 单用户连续存储管理
 - 又称单分区模式，适用于单用户情况，任何时刻主存储器中最多只有一道程序
 - 主存空间划分为系统区和用户区
 - 地址转换与存储保护：
 - 地址转换：物理地址 = 界限地址 + 逻辑地址
 - 多采用静态重定位，采用栅栏寄存器进行存储保护
 - 动态重定位，采用定位寄存器进行存储保护
 - 单用户连续存储管理的缺点：
 - 同单道程序的缺点，系统利用率低

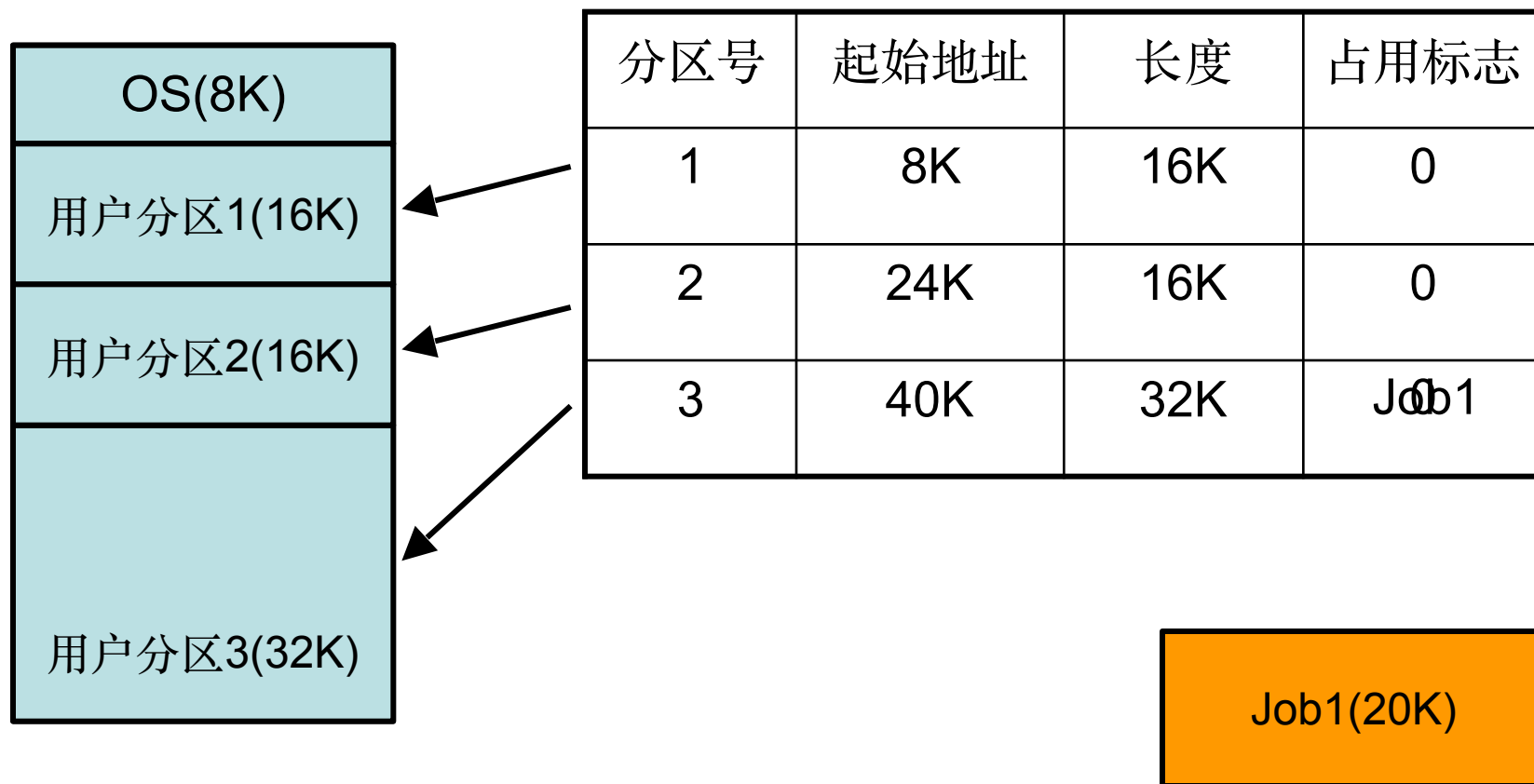
连续存储空间管理

- 固定分区存储管理
 - 又称定长分区或静态分区模式，是满足多道程序设计需要的最简单的存储管理技术
 - 基本思想：
 - 给进入主存的用户作业划分一块连续存储区域，把作业装入该连续存储区域，若有多个作业装入主存，则它们可并发执行。
 - 实现：
 - 系统启动时，系统操作员根据作业情况静态地把可分配的主存储器空间（用户空间）分割成若干个连续的区域，每个区域的位置固定，大小可相同也可不同，每个分区在任何时刻最多只装入一道程序执行

连续存储空间管理

- 固定分区存储管理示例

主存分配表



连续存储空间管理

- 固定分区存储管理

- 地址转换与存储保护

- 静态定位方式，地址转换时检查其绝对地址是否落在为其分配的用户分区？
 - 动态定位方式，专门设置一对地址寄存器（上限/下限寄存器），硬件地址转换机构对相应的地址进行比较。

- 作业调度策略

- 每个等待作业被选中时，排到一个能够装入它的最小分区的等待队列，该调度方式可能导致分区使用不均匀
 - 所有作业排成一个队列，当调度其中一个进入分区运行时，选择可容纳它的最小可用分区

连续存储空间管理

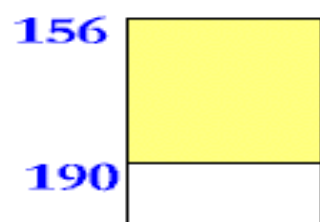
- 固定分区存储管理
 - 比较适合已知程序（作业）大小和出现频率的情形
 - 缺点：
 - 实际系统运行时，往往无法预知分区大小（太大，等同于“单用户分区模式”）
 - 主存空间利用率仍然较低
 - 无法适应动态扩充主存
 - 分区数目预先确定，限制了多道运行程序的数量

可变分区内存分配

- 根据一组空闲分区来分配大小为 n 的请求，常用方法有：
 - 首次适应(first fit)
 - 最佳适应(best fit)
 - 最坏适应(worst fit)
 - 下次适应(next fit)
 - 快速适应(quick fit)
- 在存储速度和存储资源的利用上，首次适应和最佳适应要比最差适应好。
- 可变分区存储管理
 - 又称变长分区模式
 - 基本思想：按作业的大小划分分区，但划分的时间、大小和位置均动态确定，系统在作业装入主存执行之前并不建立分区。
- 通用数据结构
 - (1) 已分配区表
 - (2) 未分配区表。

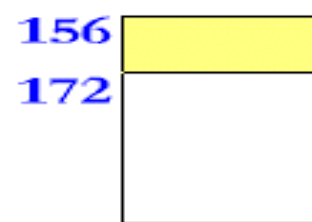
首次适应算法：

1. 首次适应算法是按空闲区首址升序的（即空闲区表是按空闲区首址从小到大）方法执行的。分配时从表首开始，以请求内存的大小逐个与空闲区进行比较，找到第一个满足要求的空闲为止；若大于请求区，就将该空闲区的一部分分配给请求者，然后，修改空闲区的大小和首址。
2. 修改空闲区有两种方法：从空闲区头开始；从空闲区尾开始。例如，空闲区大小50KB，首址156KB，申请34KB。
3. 这种算法的实质是尽可能地利用低地址部分的空闲区，而尽量地保证高地址部分的大空闲区，使其不被切削成小的区，其目的是保证在已有大的作业的到来有足够大的空闲区来满足请求者。
4. 回收时，首先考察释放区是否与系统中的某个空闲区相邻，若相邻则合并成一个空闲区，否则，将释放区作为一个空闲区按首址升序的规则插入到空闲区表适当的位置。



(190, 16)

从空闲区头开始



(172, 16)

从空闲区尾开始

最佳适应算法：

1. 最佳适应算法是将申请者放入与其大小最接近的空闲区中。切割后的空闲区最小，若系统中有与申请区大小相等的空闲区，这种算法肯定能将这种空闲区分配给申请者。（首次适应法则不一定）。
2. 最佳适应算法的空闲区表按空闲区大小升序方法组织。
3. 分配时，按申请的大小逐个与空闲区大小进行比较，找到一个满足要求的空闲区，就说明它是最适合的（即最佳的）。
4. 这种算法最大的缺点是分割后的空闲区将会很小，直至无法使用，而造成浪费。

最坏适应算法：

1. 为了克服最佳适应算法把空闲区切割得大小的缺点，人们提出了一种最坏适应算法，即每次分配时，总是将最大的空闲区切去一部分分配给请求者，其依据是当一个很大的空闲区被切割了一部分后可能仍是一个较大的空闲区。避免了空闲区越分越小的问题。
2. 最坏适应算法的空闲区表是按空闲区大小降序的方法组织的（从大到小的顺序）。
3. 分配时总是取表中的第一个表目，若不能满足申请者的要求，则表示系统中无满足要求的空闲区，分配失败；否则，将从该空闲区中分配给申请者，然后修改空闲区的大小，并将它插入到空闲区表的适当位置。

连续存储空间管理

Processor contains two *control registers*

- Memory base
- Memory limit

Each memory access checks

`If $V < \text{limit}$`

`$P = \text{base} + V;$`

`Else`

`ERROR /* what do we call this error? */`

During context switch...

- Save/load user-visible registers
- Also load process's base, limit registers

连续存储空间管理

1. How do we *grow* a process?

- Must increase “limit” value
- Cannot expand into another process's memory!
- Must move entire address spaces around
 - Very expensive

2. Fragmentation

- New processes may not fit into unused memory “holes”

3. Partial memory residence

- Must *entire* program be in memory at same time?

连续存储空间管理

- 固定分区存储管理示例



连续存储空间管理

- 可变分区存储管理实现
 - 地址转换和存储保护
 - 设置两个专门的寄存器
 - 基址寄存器，存放分区的起始地址
 - 限长寄存器，存放分区的长度
 - 移动技术
 - 将分散的空闲区汇集成一个较大的空闲区，以利于大作业的执行
- 连续分区管理方式存在的问题
 - 碎片问题：
 - 每个程序总是要求占用连续的存储空间，经过一段时间的运行将会产生许多碎片（不连续的容量较小的分区），为接纳新的作业往往需要通过移动已有的主存内容来产生容量较大的分区。
 - 移动技术实现复杂，并不可避免地导致管理开销增大

碎片问题

碎片：在采用分区存储管理的系统中，会形成一些非常小的分区，最终这些非常小的分区不能被系统中的任何用户（程序）利用而浪费。造成这样问题的主要原因是用户程序装入内存时是整体装入的

- **外部碎片**—当整个内存空间可以满足一个请求，但它不是连续的。
- **内部碎片**—分配的内存可能比申请的内存大一点，这两者之间的数字之差。
- 通过移动技术来减少外部碎片
 - 移动内存内容，把一些小的空闲内存结合成一个大的块。
 - 只有在执行时期进行动态重定位(relocation)，才有可能进行移动

连续存储空间管理-外部碎片

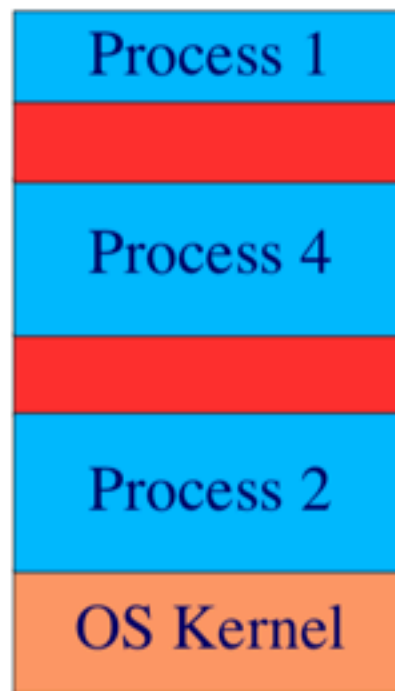
**Free memory is small
chunks**

Doesn't fit large objects

**Can “disable” lots of
memory**

Can fix

- Costly “compaction”
 - aka “Stop & copy”



连续存储空间管理-内部碎片

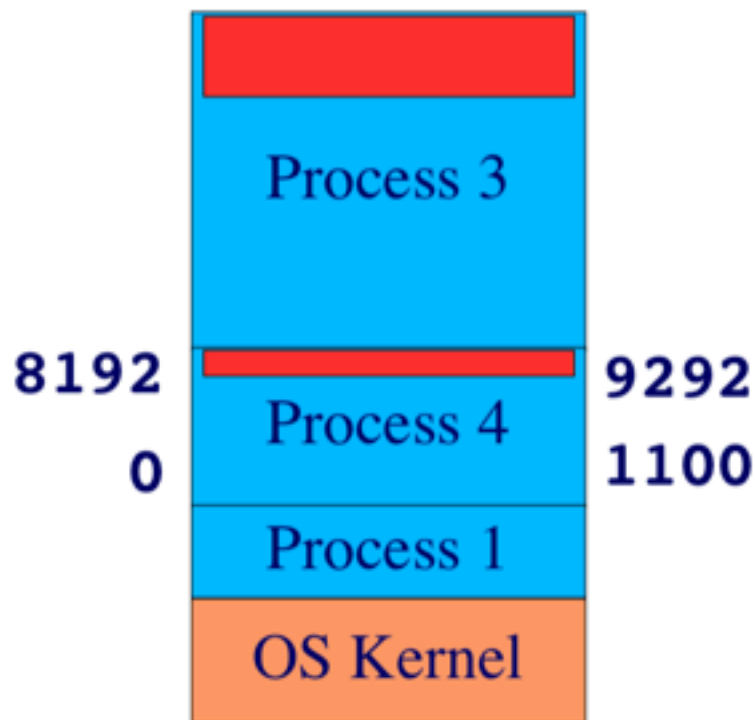
Allocators often round up

- 8K boundary (*some* power of 2!)

Some memory is wasted *inside* each segment

Can't fix via compaction

Effects often non-fatal



连续存储空间管理-交换技术(Swapping)

Multiple user processes

- Sum of memory demands > system memory
- Goal: Allow *each process* 100% of system memory

Take turns

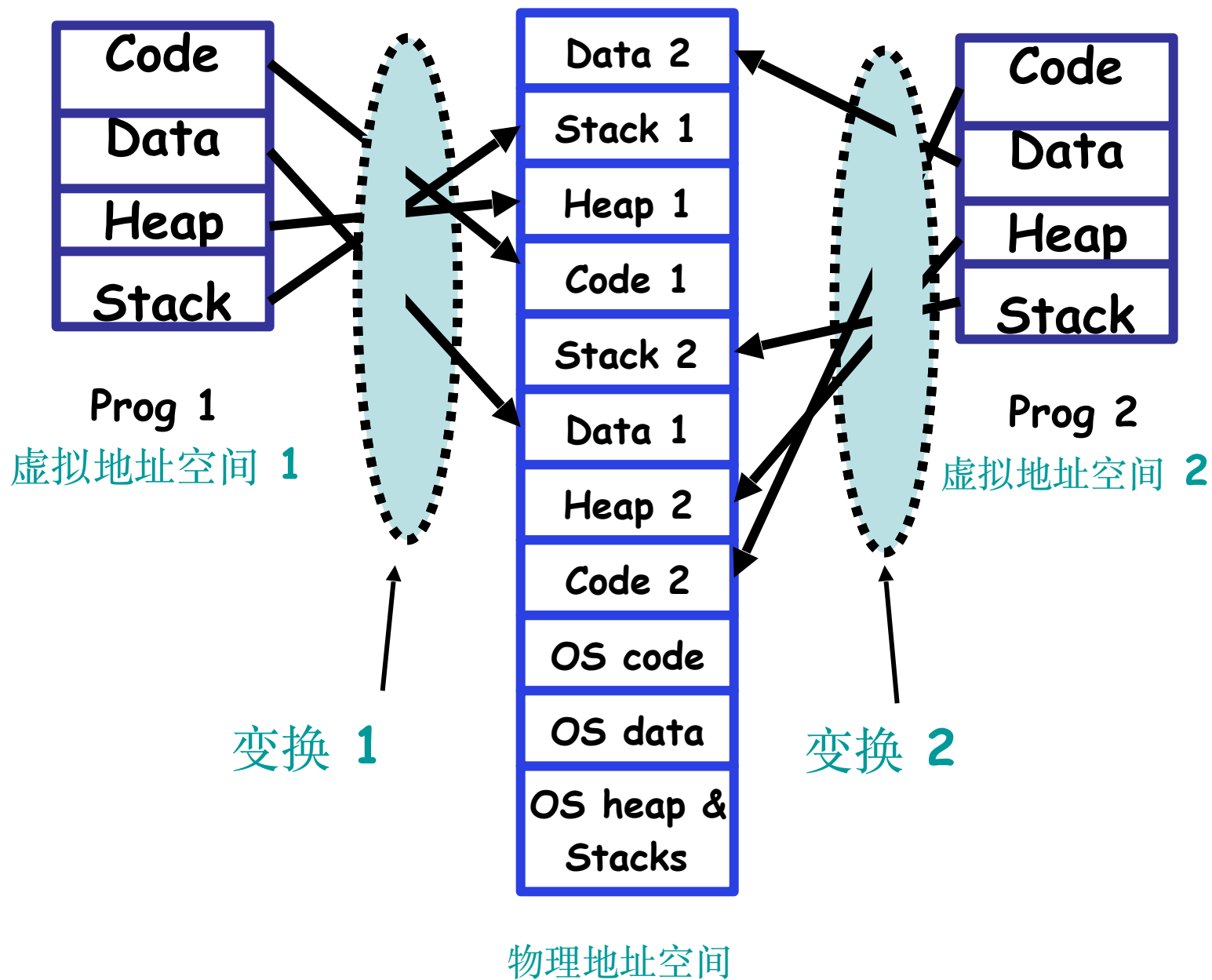
- Temporarily evict process(es) to disk
 - Not runnable
 - Blocked on *implicit* I/O request (e.g., “swapread”)
- “Swap daemon” shuffles process in & out
- Can take *seconds* per process
 - Modern analogue: laptop suspend-to-disk
- Maybe we need a better plan?

传统内存管理技术

- 分区技术
- 交换技术
 - 把暂时不用的某个程序及数据的一部分或全部从内存移到外存中去，或把指定的程序或数据从外存读到相应的内存中，并将控制权转给该程序的一种内存扩充技术。
- 覆盖技术
 - 将大程序划分为一系列的覆盖，每个覆盖是一个相对独立的程序单位，把程序执行时不需要同时装入内存的覆盖构成一组，称为覆盖段。一个覆盖段内的覆盖共享同一存储区域，该区域成为覆盖区，其大小由对应的覆盖段内最大的覆盖决定。

Problems to solve?

- **Process Growth Problem**
- **Fragmentation (compaction) Problem**
- **Long delay to swap**



Key ideas

- **Divide memory more finely**
 - **Page: small region of virtual mem**
 - **Frame: small region of physical mem**
- **Any page can map to any frame**

Problems solved

- **Process Growth Problem**
 - any process can use any free frame for any purpose
- **Fragmentation (compaction) Problem**
 - process doesn't need to be contiguous
 - no compaction needed
- **Long delay to swap**
 - swap part of the process

分页式存储管理

- 基本原理：
 - 允许作业存放在若干个不相邻的分区中，
 - 既可免去移动内存信息而产生的工作量，又可充分利用主存空间，尽量减少主存碎片。
 - 程序地址空间分成大小相等的页面，同时把内存也分成与页面大小相等的块
 - 当一个用户程序装入内存时，以页面为单位进行分配。页面的大小是为 2^n 。

分页式存储管理

- 基本概念：

- **帧**：主存空间按物理地址分成多个大小相等区，每个区称为块(帧)（又称frame）
- **页**：程序（作业）按逻辑地址分成多个大小相等的区，每个区称为一个页面(page)，大小与帧大小相等
- 虚地址（逻辑地址）： 页号和页偏移

页码	页偏移
p	d

分页式存储管理

Contiguous allocation

- Each process was described by (base,limit)

Paging

- Each *page* described by (base,limit)?
 - Pages typically one size for whole system
- Ok, each *page* described by (base address)
- Arbitrary page \Rightarrow frame mapping requires some work
 - Abstract data structure: “map”
 - Implemented as...

Page-Frame Mapping: How?

Linked list

- $O(n)$, so $V \Rightarrow P$ time gets longer for large addresses!

Array

- Constant time access
- Requires (large) contiguous memory for table

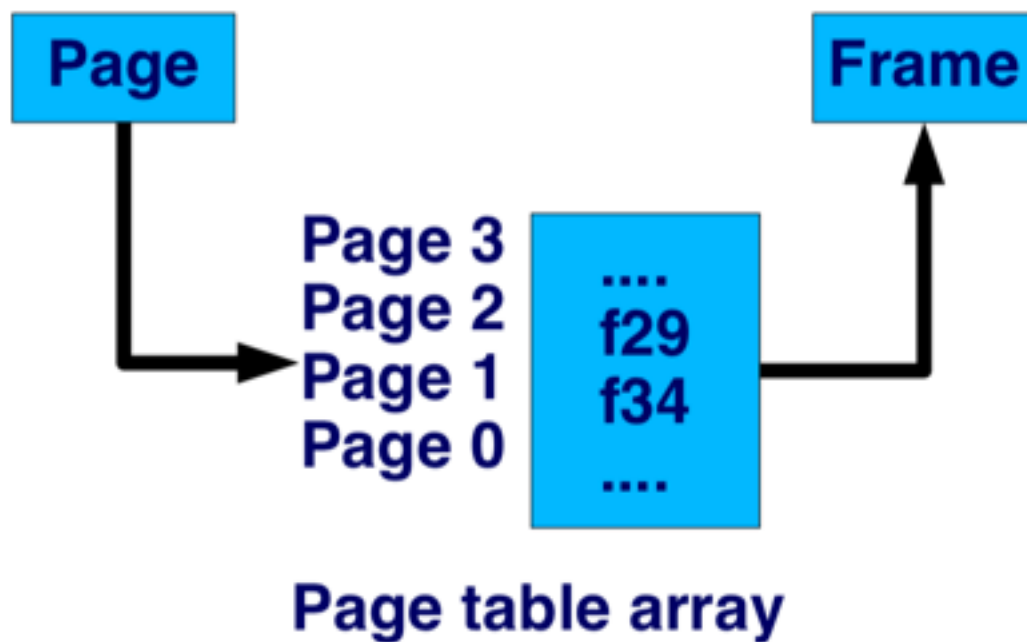
Hash table

- Vaguely-constant-time access
- Not really bounded though

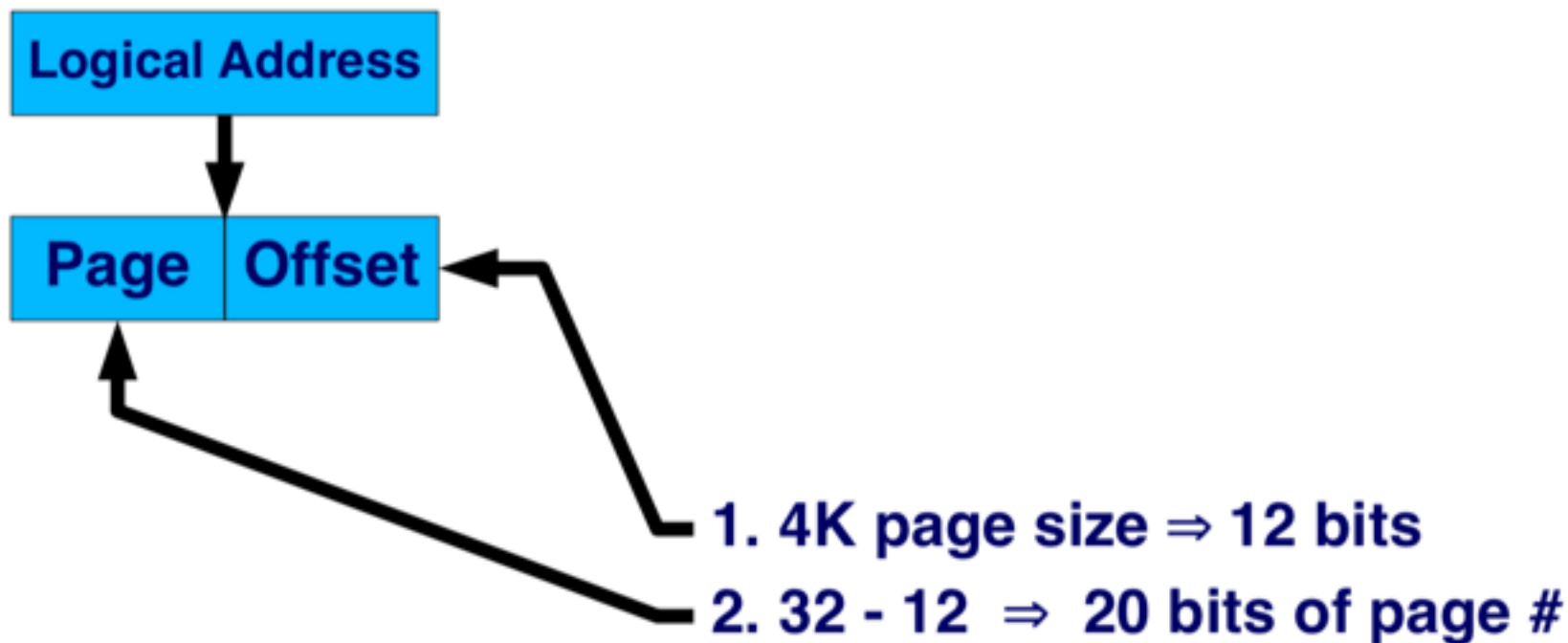
Splay tree

- Excellent amortized expected time
- *Lots* of memory reads & writes possible for one mapping
- Not yet demonstrated in hardware

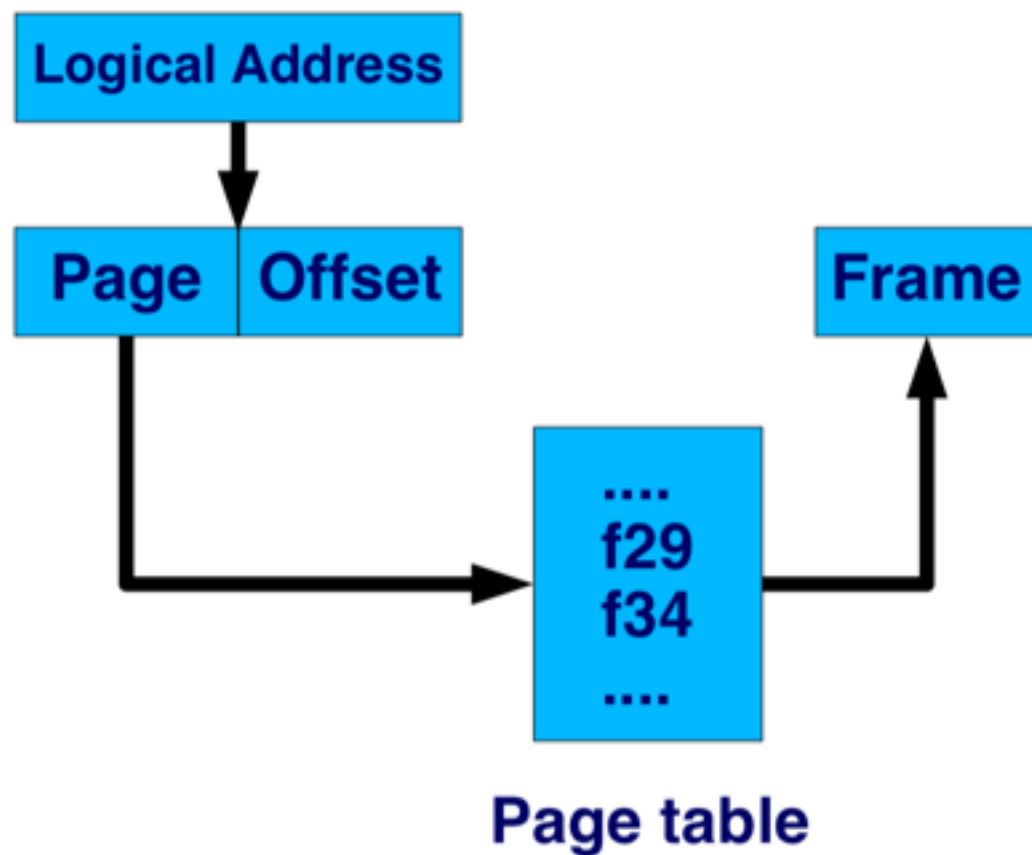
分页式存储管理



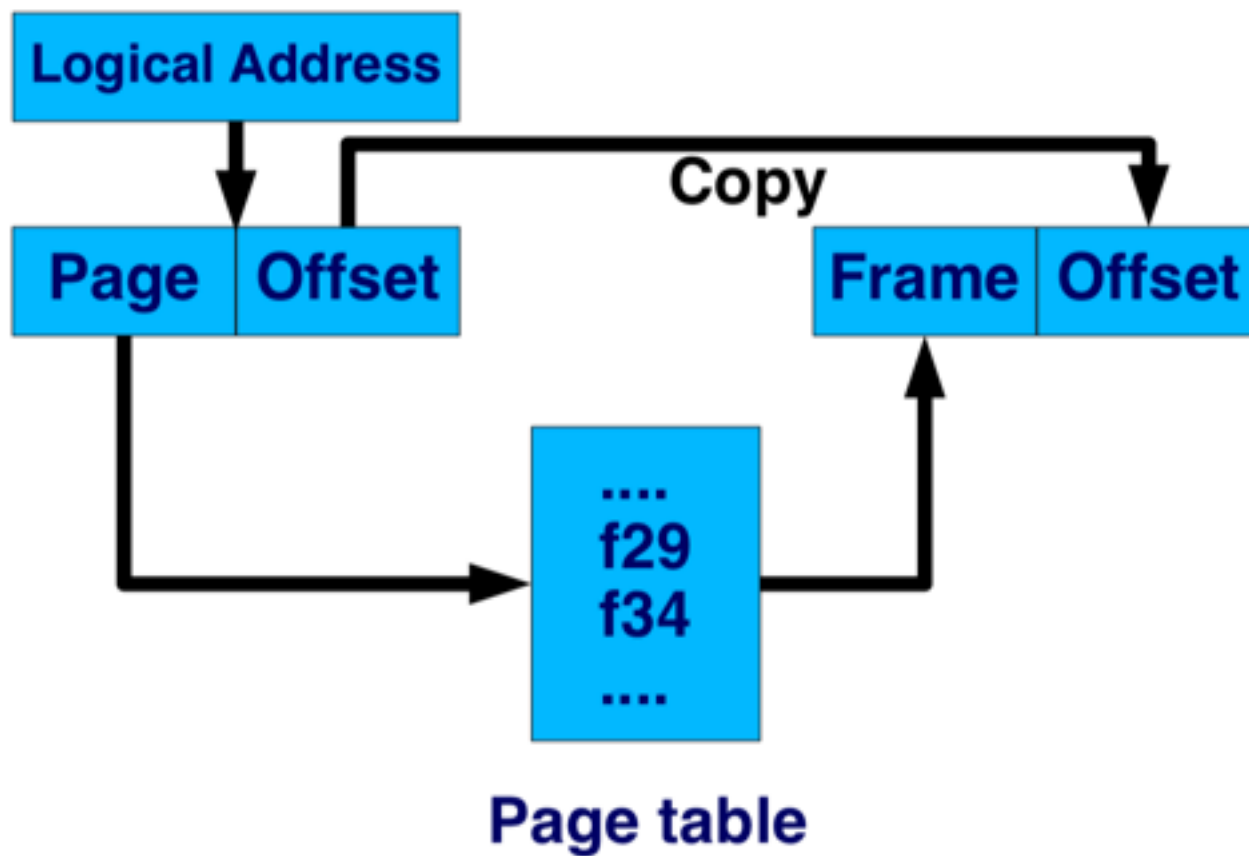
分页式存储管理



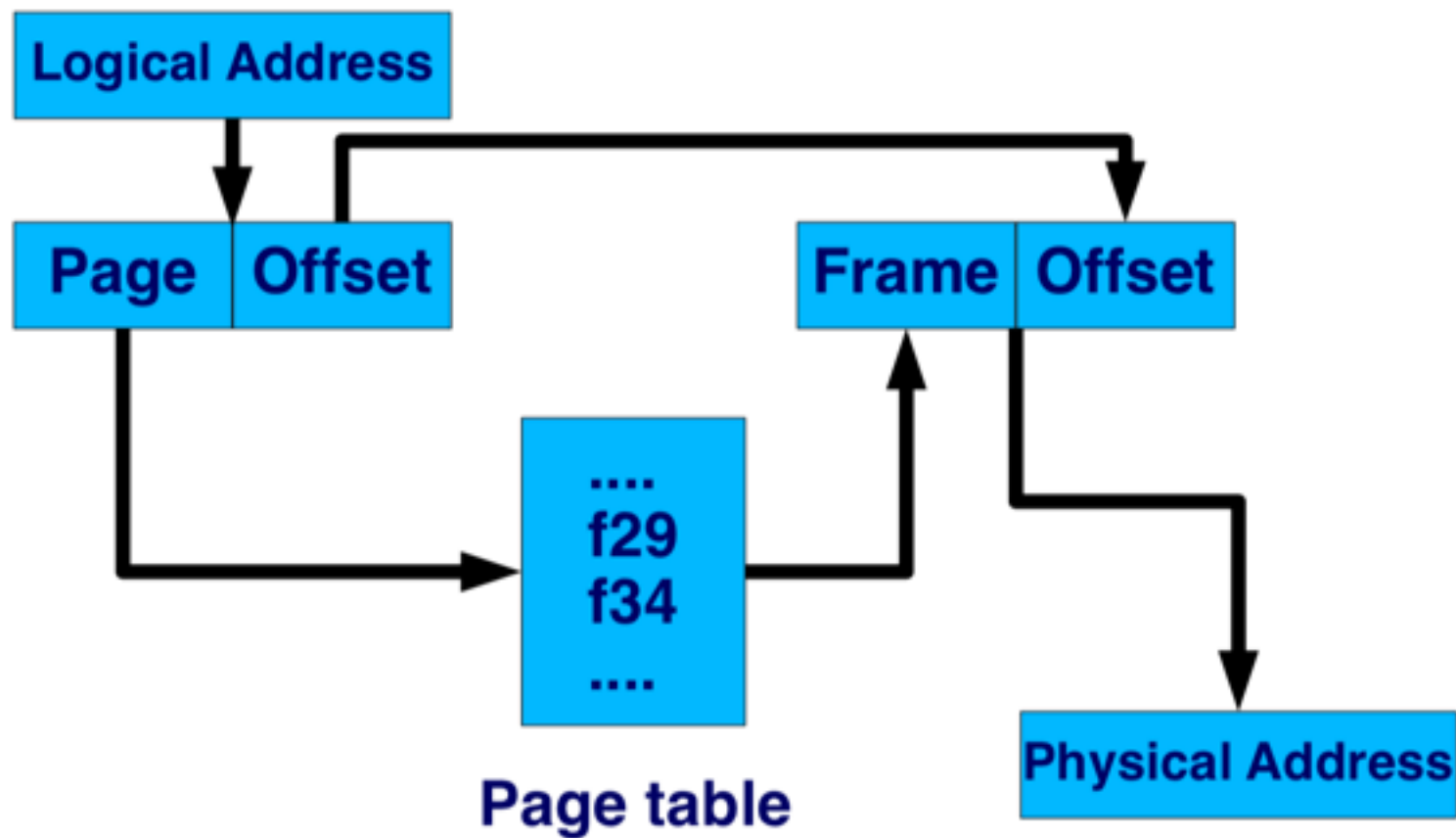
分页式存储管理



分页式存储管理



分页式存储管理



分页式存储管理

User view

- Memory is a linear array

OS view

- Each process requires N frames, located anywhere

Fragmentation?

- *Zero* external fragmentation
- Internal fragmentation: average $\frac{1}{2}$ page per region

分页式存储管理 - Problem?

Processor makes *two* memory accesses!

- Splits address in %esi into page number, intra-page offset
- Adds page number to page table base register
- *Fetches page table entry (PTE) from memory*
- Concatenates frame address with intra-page offset
- *Fetches program's data from memory into %eax*

分页式存储管理

- 基本概念

- 页表、作业表和地址转换

页表

第0页	块号1 (20)
第1页	块号2(21)
第2页	块号3(51)
...	...

作业表

作业名	页表始址	页表长度
Job1	0	3
...

物理地址 = 帧号 (块号) * 块长 + 单元号

查找页表

逻辑地址 = 页号 * 页长 + 单元号

分页式存储管理

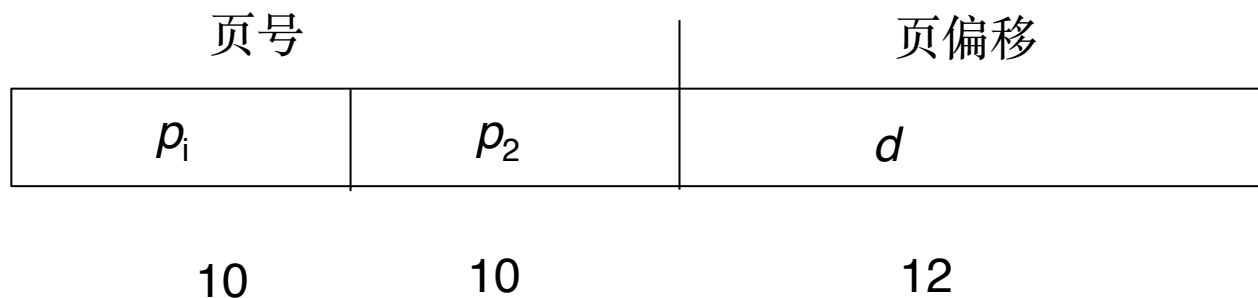
- 允许进程的物理地址空间可以是不连续的
- 物理内存分成大小固定的块，称为**帧(Frame)**
- 逻辑内存也分位同样固定大小的块，叫做**页(Page)**
- 页、帧大小由硬件来决定，通常为**2**的幂
- 建立一个页表，把逻辑地址转换为物理地址
 - **页表是页式存储管理的数据结构**：它包括用户程序空间的页面与内存块的对应关系、页面的存储保护和存取控制方面的信息。在实际的系统中，为了节省存储空间，在页表中可以省去页号这个表目。
 - **动态重定位**：让程序的指令执行时动态地进行地址变换，给每个页面设立重定位寄存器，重定位寄存器的集合便称页表。页表是操作系统为每个用户作业建立的，用来记录程序页面和主存对应页框的对照表。

分页式存储管理

- 为解决页表规模过大占用内存空间的问题
 - 多级页表
 - 页表可以部分存放在内存中
 - 例，二级页表系统中，一次按逻辑地址的主存访问需要访问三次主存：一次访问页目录、一次访问页表、一次访问具体的数据
 - 现代的大多数计算机系统，都支持非常大的逻辑地址空间($2^{32} \sim 2^{64}$)。在这样的环境下，页表就变得非常大，要占用相当大的内存空间。

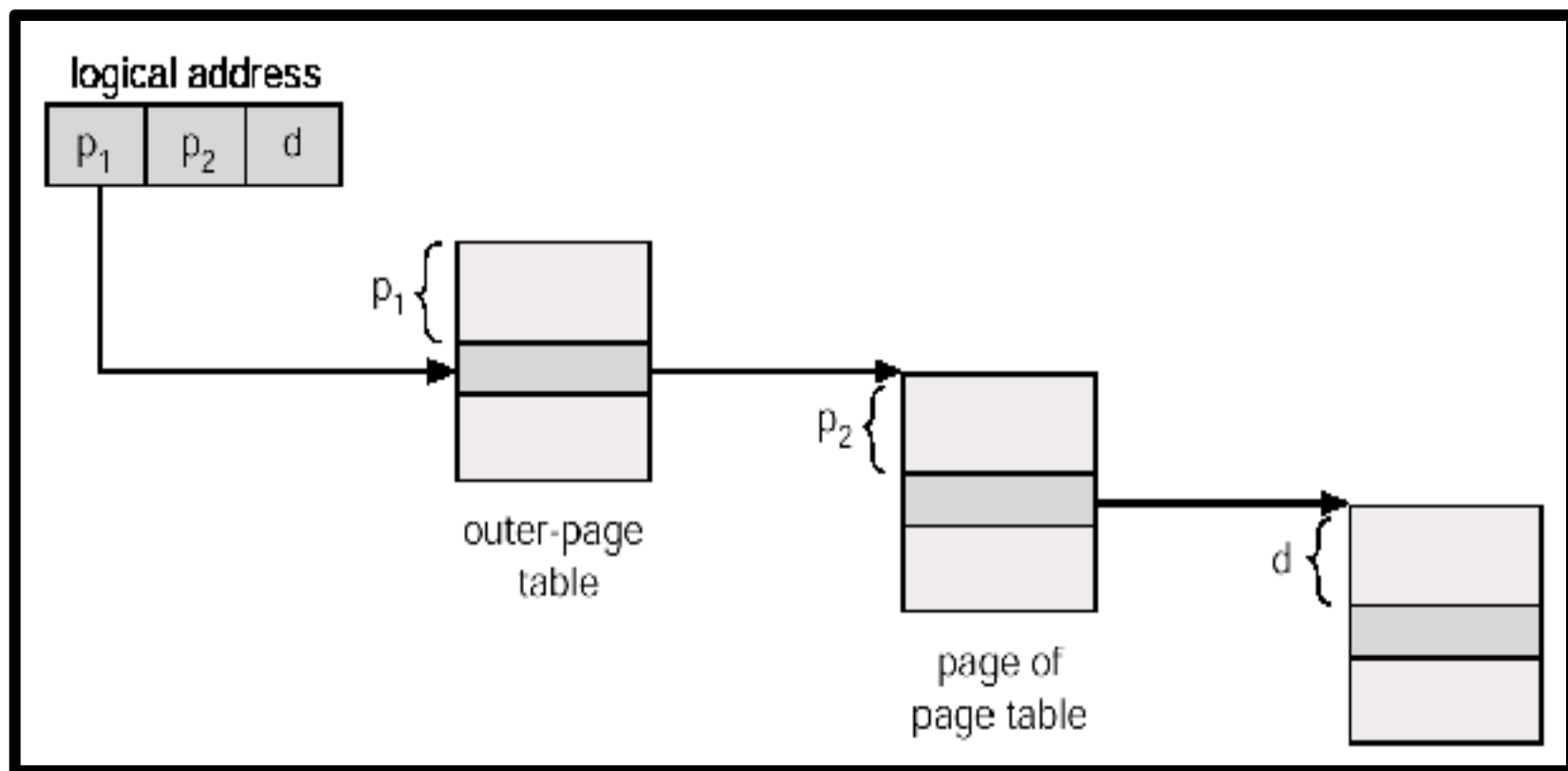
两级页表

- 一个逻辑地址(32位系统, 页大小 4K) 被分为:
 - 一个20位的页号
 - 一个12位的偏移
- 对页表进行再分页, 页号分解为:
 - 一个10位的页号
 - 一个10位的偏移
- 因此, 一个逻辑地址表示如下:



- p_1 是用来访问外部页表的索引, p_2 是外部页表的页偏移

两级32位分页结构的地址转换机制



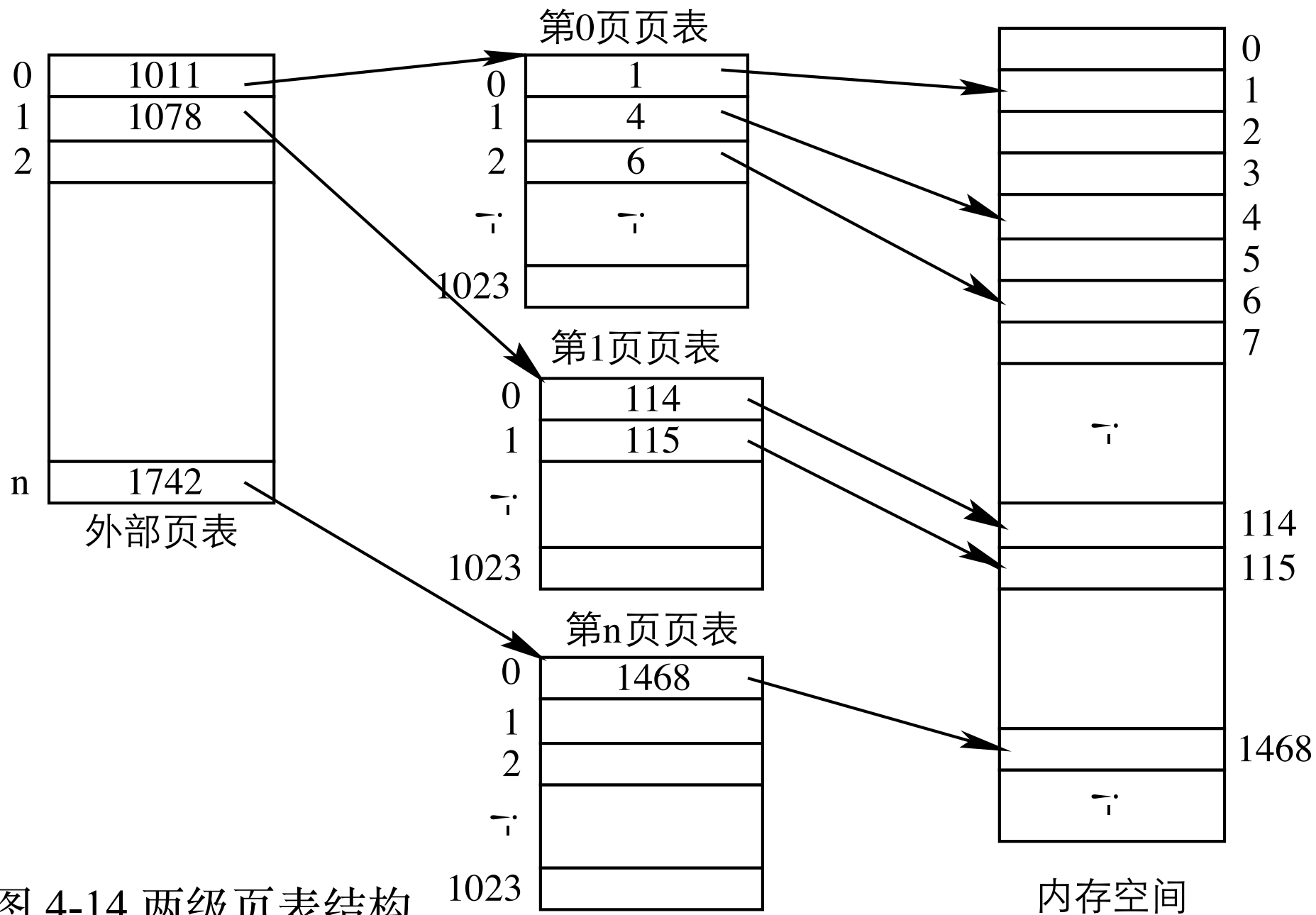


图 4-14 两级页表结构

虚地址变换

1. 虚地址（逻辑地址、程序地址）以十六进制、八进制、二进制的形式给出

- 将虚地址转换成二进制的数；
- 按页的大小分离出页号和位移量（低位部分是位移量，高位部分是页号）；
- 将位移量直接复制到内存地址寄存器的低位部分；
- 以页号查页表，得到对应页装入内存的块号，并将块号转换成二进制数填入地址寄存器的高位部分，从而形成内存地址。
内存地址 = 块号 × 页大小 + 位移量。

2、虚地址以十进制数给出

- 页号 = 虚地址 % 页大小
- 位移量 = 虚地址 mod 页大小
- 以页号查页表，得到对应页装入内存的块号
- 内存地址 = 块号 × 页大小 + 位移量

例 1：有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、A、5块，试将逻辑地址0AFEH，1ADDH转换成物理地址。

页号	块号
0	7
1	9
2	A
3	5

逻辑地址0AFEH

0000 1010 1111 1110

P = 1 d = 010 1111 1110

物理地址

= 0100 1010 1111 1110

= 4AFEH

虚地址1ADDH

0001 1010 1101 1101

P = 3

d = 010 1101 1101

物理地址

= 0010 1010 1101 1101

= 2ADDH

页号 块号

0	7
1	9
2	A
3	5

- 例2：有一系统采用页式存储管理，有一作业大小是8KB，页大小为2KB，依次装入内存的第7、9、10、5块，试将逻辑地址7145，3412转换成物理地址。

逻辑地址 3412

$$P = 3412 \% 2048 = 1$$

$$\begin{aligned} d &= 3412 \bmod 2048 \\ &= 1364 \end{aligned}$$

$$MR = 9 * 2048 + 1364 = 19796$$

逻辑地址3412的物理地址
是：19796

解：页表：

页号	块号
0	7
1	9
2	10
3	5

逻辑地址 7145

$$P = 7145 \% 2048 = 3$$

$$d = 7145 \bmod 2048 \\ = 1001$$

$$MR = 5 * 2048 + 1001 = 11241$$

逻辑地址7145的物理地址
是： 11241

解： 页表：

页号 块号

0	7
1	9
2	10
3	5

分页中的碎片

- 没有外部碎片
- 有内部碎片
 - 最坏情况下，一个需要 n 页再加1B的进程，需要 $n+1$ 个帧，几乎有一个帧的碎片

页表的实现

- 页表放在放在内存中，用页表基址寄存器指向页表
(*Page-table base register* , PTBR)
- 页表限长寄存器表明页表的长度 (*Page-table length register* , PRLR)
- 每一次的数据/指令存取需要两次内存访问，一次是访问页表，一次是访问数据)
 - 通过一个相联存储器 **associative memory** 或 **translation look-aside buffers (TLBs)**，可以解决两次存取的问题，加速页表查找

分页式存储管理

- 相联存储器和快表

- 通常页表存放在主存中，因此按逻辑地址访问某个主存地址内容时，需要涉及二次主存访问，效率较低
- 相联存储器，一个专用的高速缓冲存储器，用于存放最近被访问的部分页表，是分页式存储管理的重要组成部分。
- 快表，存放在相联存储器中的部分页表内容

页号	块号（帧号）	特征位	...
----	--------	-----	-----

相联存储器-TLB

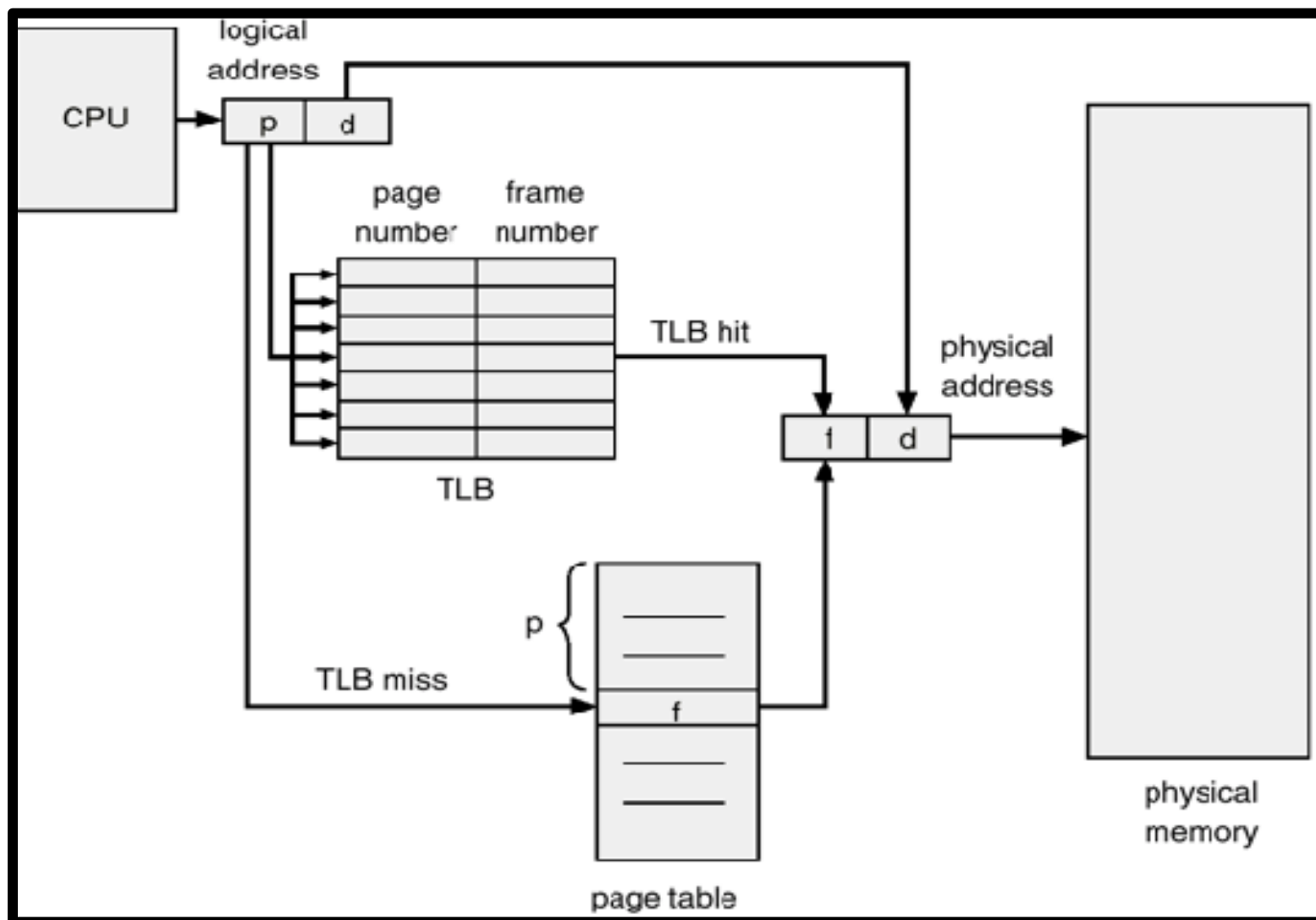
- 相联存储器- 并行搜索

Page #	Frame #

Address translation (A' , A'')

- 如果 页号 A' 在相联寄存器内, 得到对应的帧号
- 否则查找内存中的页表来得到帧号

带TLB的分页硬件



分页式存储管理 - Problem?

Processor makes *two* memory accesses!

- Splits address in %esi into page number, intra-page offset
- Adds page number to page table base register
- *Fetches page table entry (PTE) from memory*
- Concatenates frame address with intra-page offset
- *Fetches program's data from memory into %eax*

相联存储器-TLB

Problem

- *Cannot afford* double/triple/... memory latency

Observation - “locality of reference”

- Program often accesses “nearby” memory
 - Next instruction often on same page as current instruction
 - Next byte of string often on same page as current byte
 - (“Array good, linked list bad”)

Solution

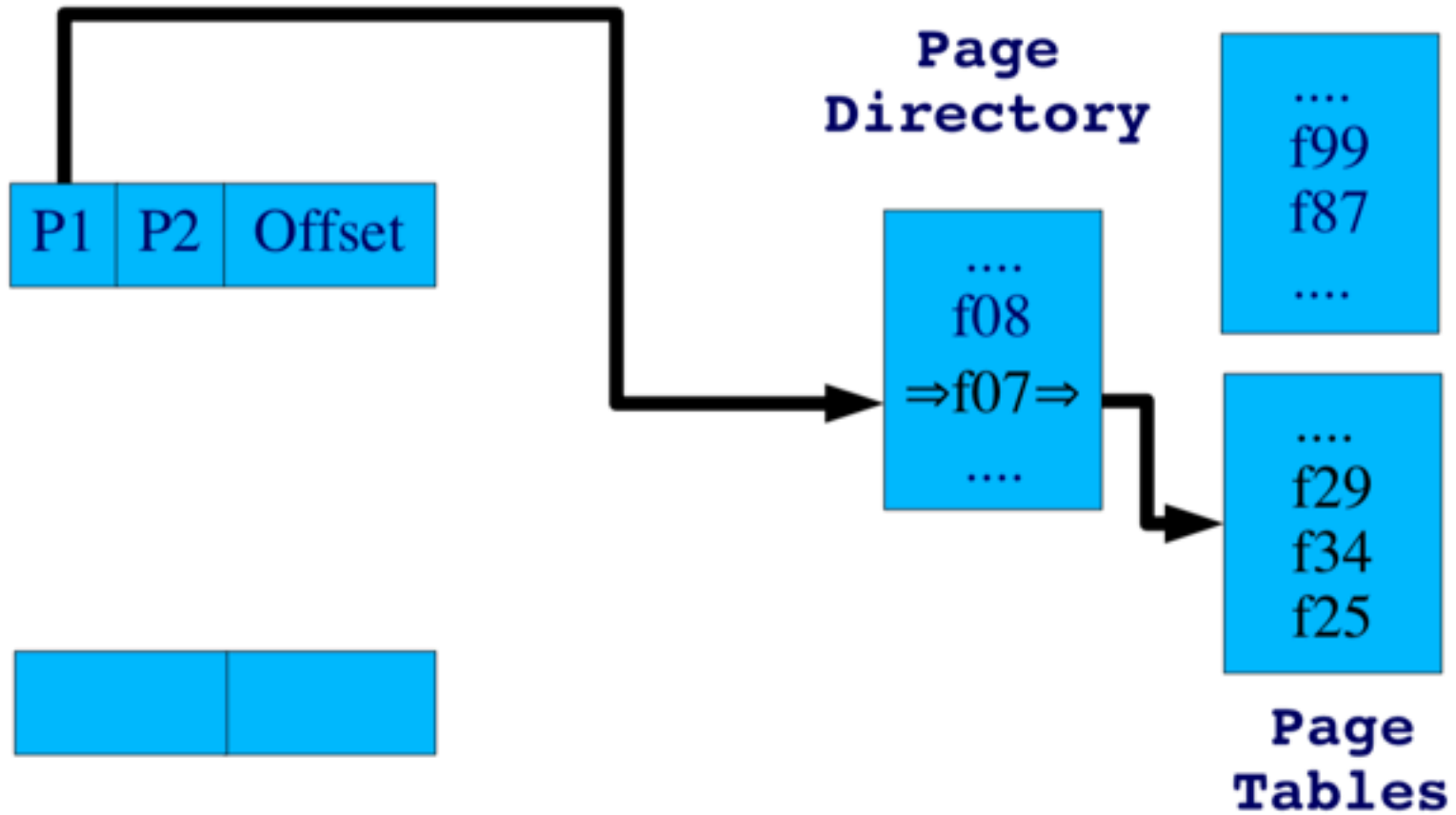
- Page-map hardware caches virtual-to-physical *mappings*
 - Small, fast on-chip memory
 - “Free” in comparison to slow off-chip memory

相联存储器-TLB

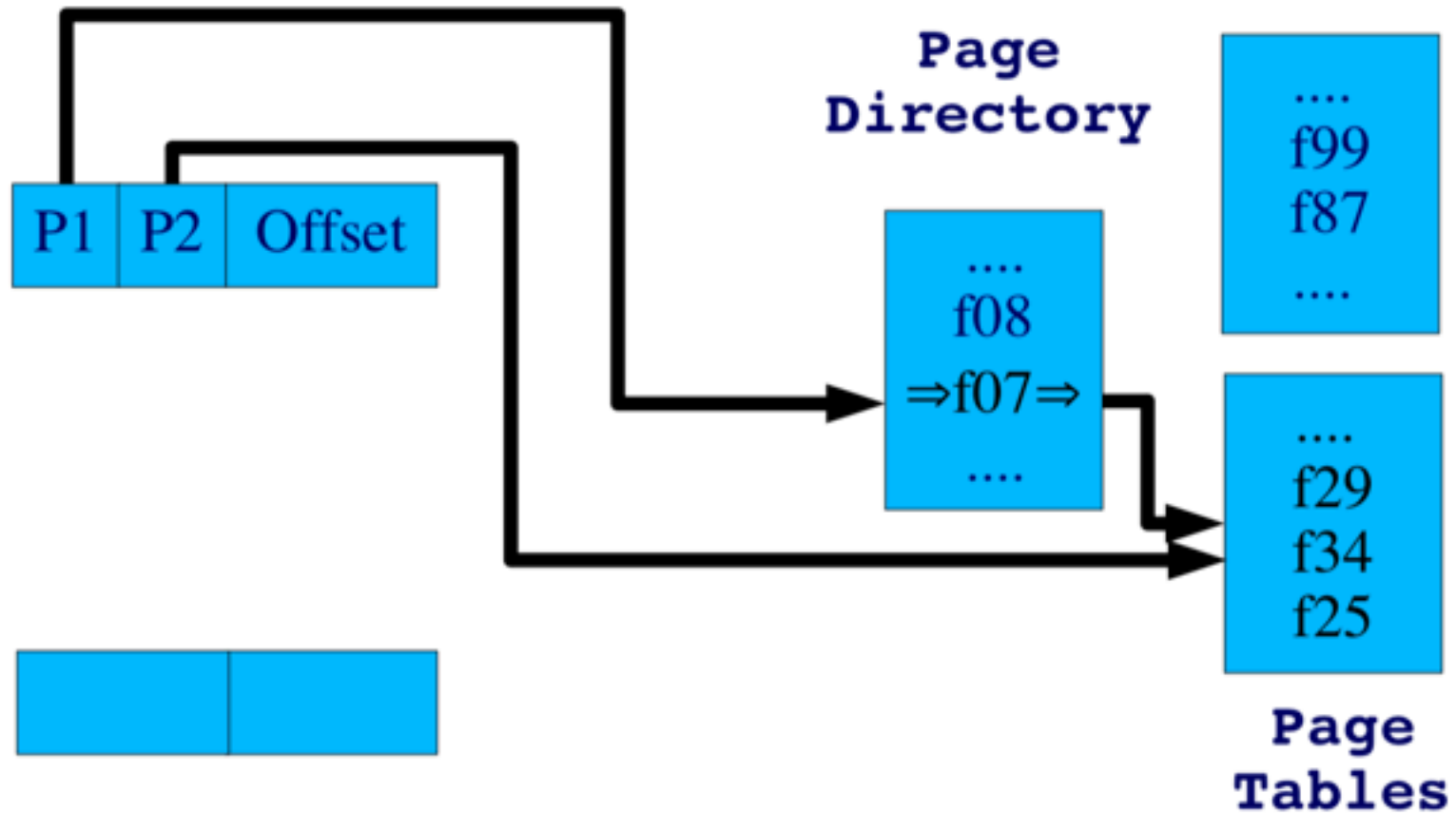
Approach

- Remember the most-recent virtual-to-physical translation
 - (obtained from, e.g., Page Directory + Page Table)
- See if next memory access is to same page
 - If so, skip PD/PT memory traffic; use same frame
 - 3X speedup, cost is two 20-bit registers
 - » “Great work if you can get it”

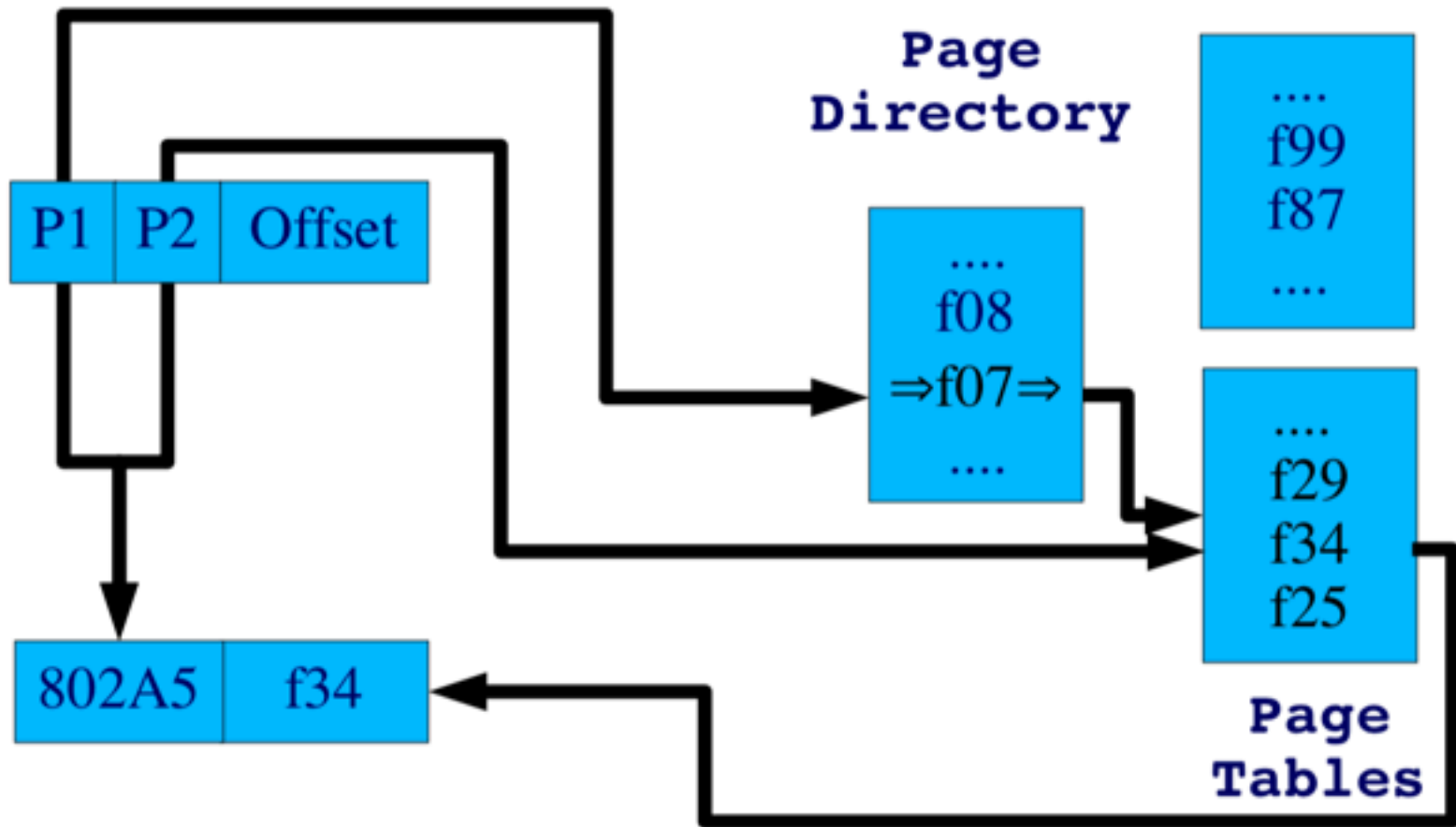
相联存储器-TLB



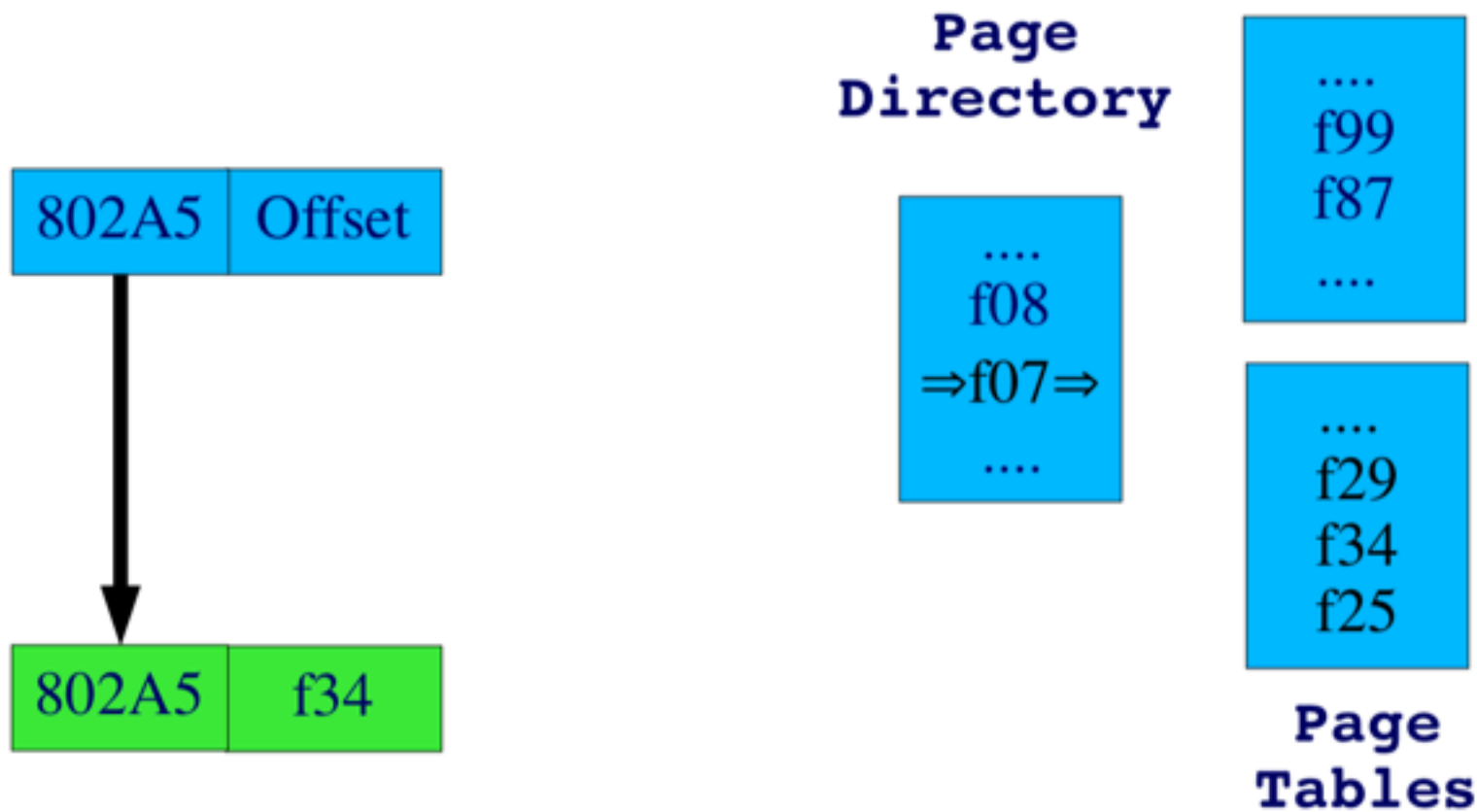
相联存储器-TLB



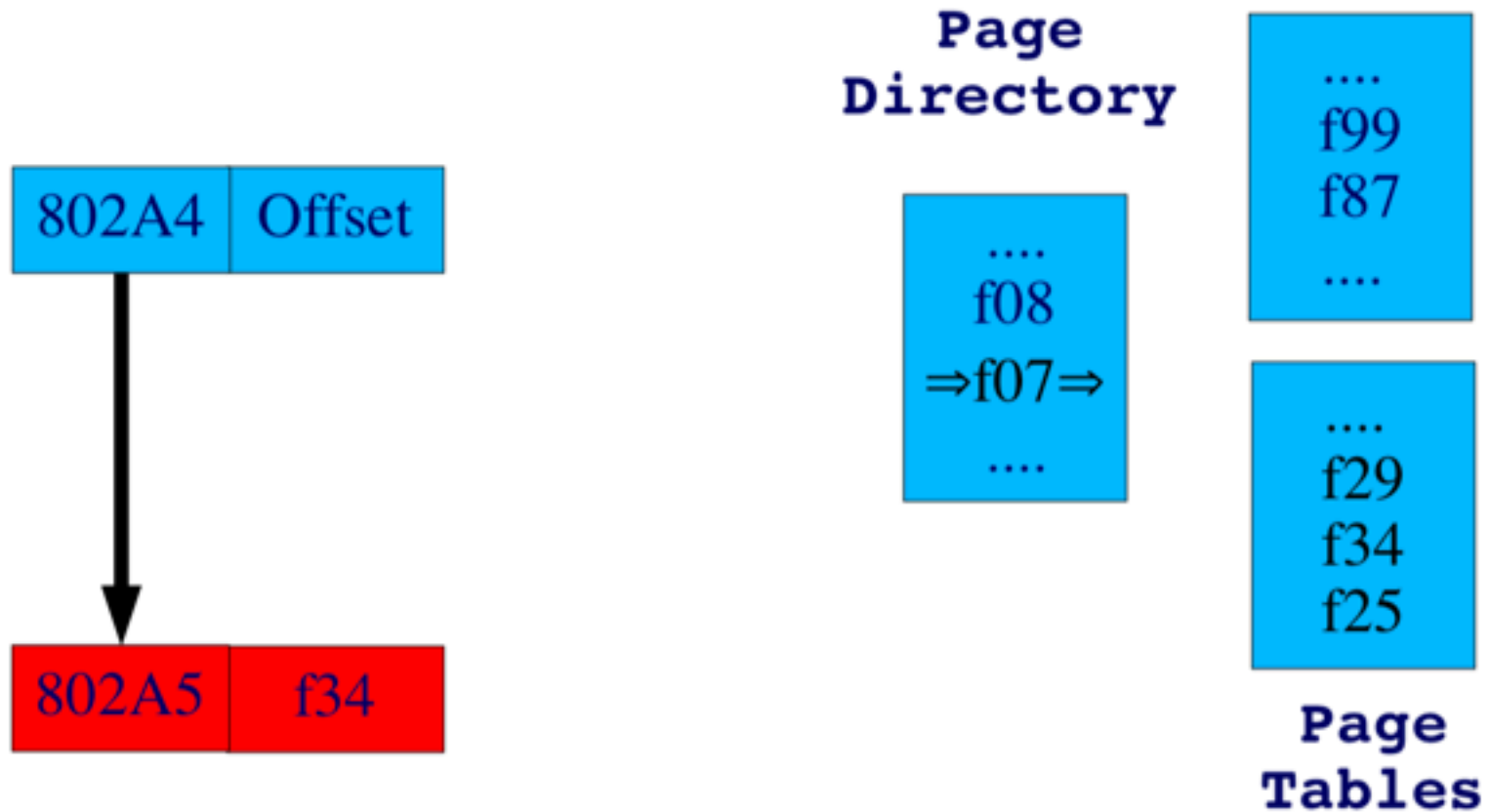
相联存储器-TLB



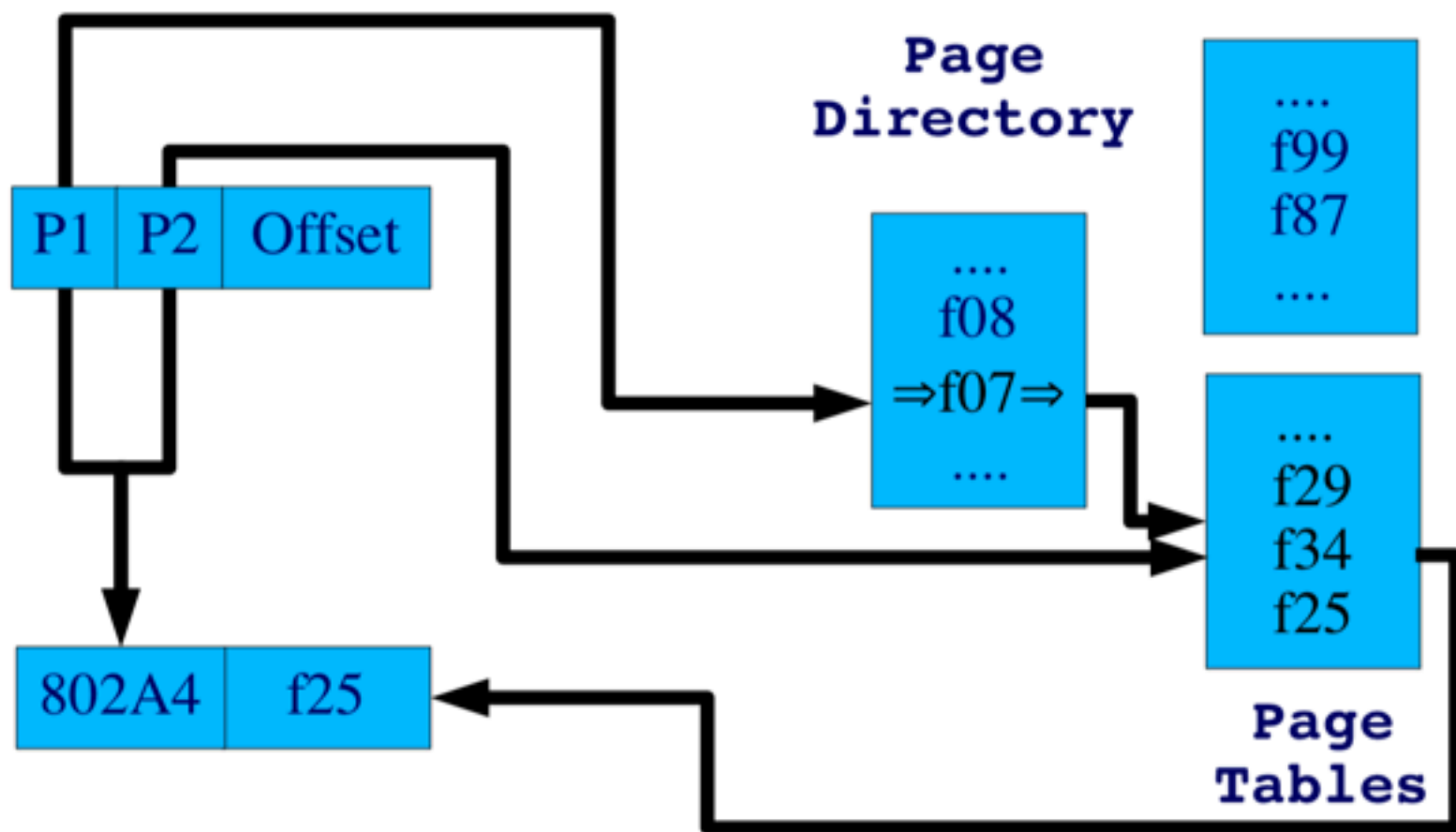
相联存储器-TLB



相联存储器-TLB



相联存储器-TLB



基于TLB的有效内存访问时间

- 相联存储器的查找需要时间： ε
- 访问内存一次需要时间： t
- 命中率 – 在相联存储器中找到页号的概率，概率与相联存储器的大小有关。
- 命中率 = α
- 有效访问时间（Effective Access Time , EAT）

$$EAT = (t + \varepsilon) \alpha + (2t + \varepsilon)(1 - \alpha)$$

Page Table

PTE conceptual job

- Specify a frame number

PTE flags

- Valid bit
 - Not-set means access should generate an exception
- Protection
 - Read/Write/Execute bits
- Reference bit, “dirty” bit
 - Set if page was read/written “recently”
 - Used when paging to disk (later lecture)
- Specified by OS for each page/frame
 - Inspected/updated by hardware

存储保护

- 内存保护由与每个帧相关联的保护位来实现的可定义读写权限
- 有效-无效位附在页表的每个表项中：
 - “有效”表明相关的页属于进程的逻辑地址空间，并且是一个valid的页。
 - “无效”表明页不在进程的逻辑地址空间中，或valid但在外存中。

Page Table

Problem

- Assume 4 KByte pages, 4-Byte PTEs
- Ratio: 1024:1
 - 4 GByte virtual address (32 bits) \Rightarrow _____ page table

Page Table

Problem

- Assume 4 KByte pages, 4-Byte PTEs
- Ratio: 1024:1
 - 4 GByte virtual address (32 bits) \Rightarrow 4 MByte page table
 - *For each process!*

Page Table

Problem

- Assume 4 KByte pages, 4-Byte PTEs
- Ratio: 1024:1
 - 4 GByte virtual address (32 bits) \Rightarrow 4 MByte page table
 - *For each process!*

One Approach: Page Table Length Register (PTLR)

- (names vary)
- Many programs don't use entire virtual space
- Restrict a process to use entries 0...N of page table
- On-chip register detects out-of-bounds reference ($>N$)
- Allows small PTs for small processes
 - (as long as stack isn't far from data)

Page Table

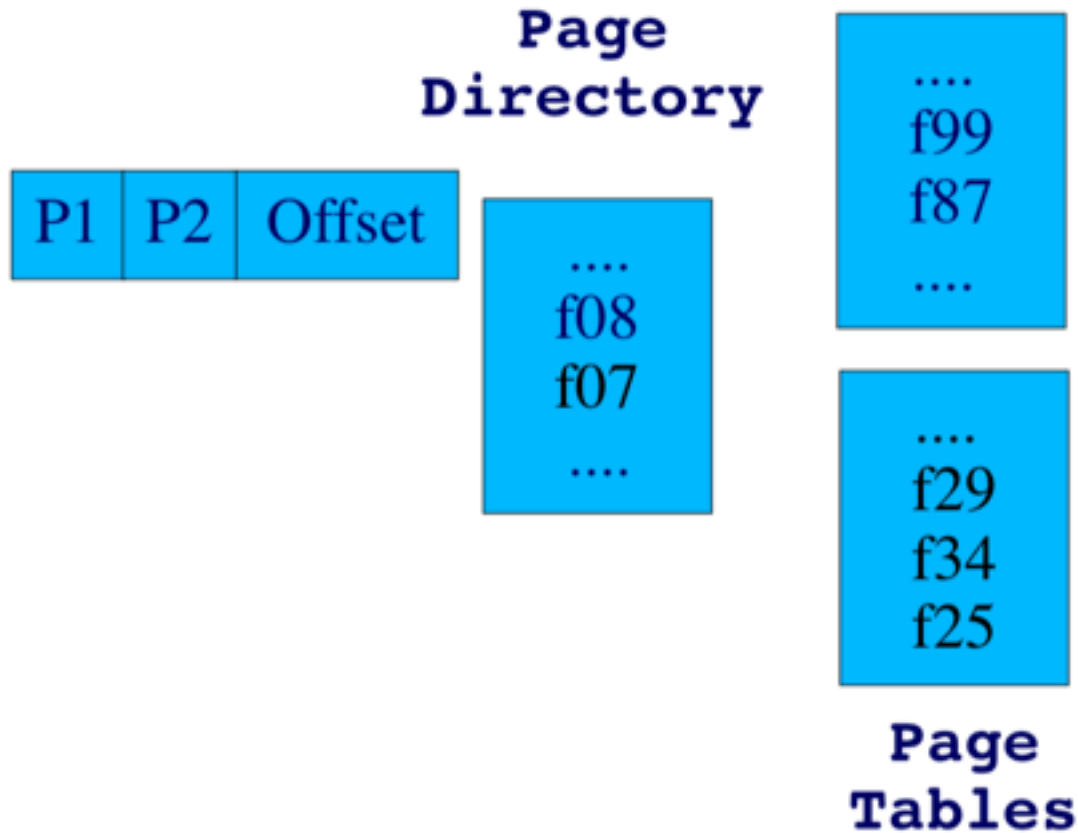
Key observation

- Each process page table is a *sparse mapping*
- Many pages are not backed by frames
 - Address space is sparsely used
 - » Enormous “hole” between bottom of stack, top of heap
 - » Often occupies 99% of address space!
 - Some pages are on disk instead of in memory

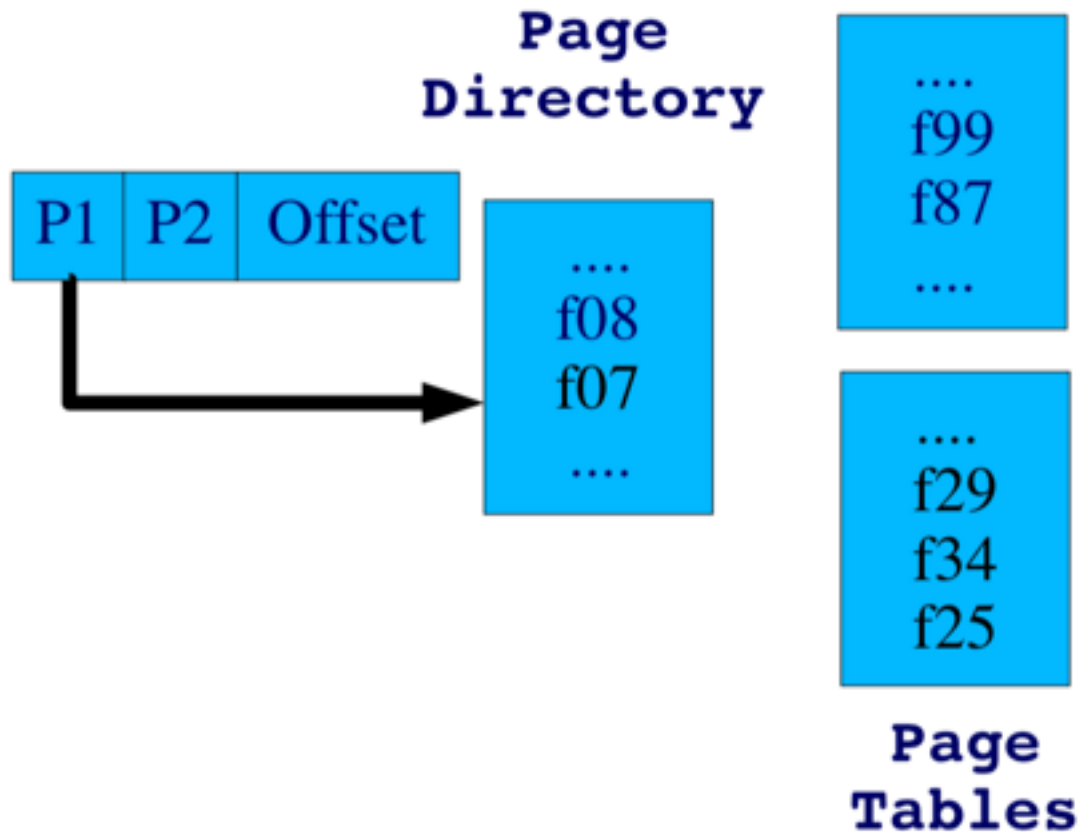
Refining our observation

- Page tables are not randomly sparse
 - Occupied by *sequential memory regions*
 - Text, rodata, data+bss, stack
- “Sparse list of dense lists”

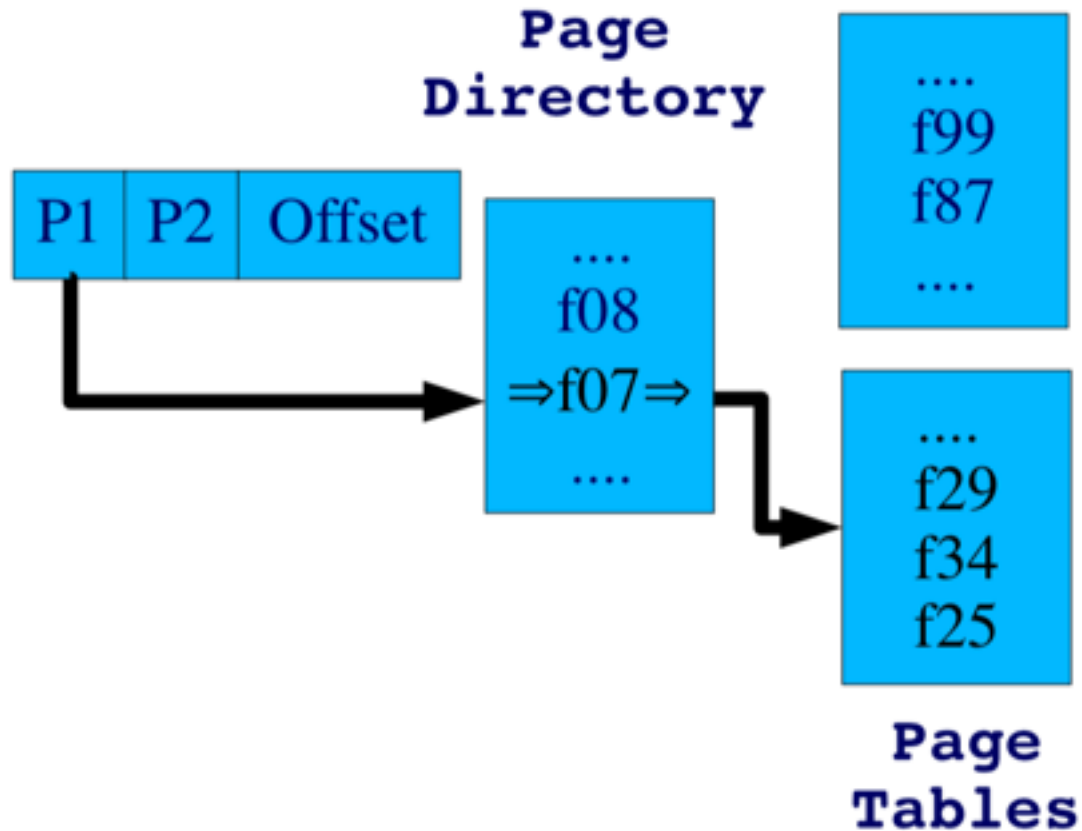
Multi-level Page Table



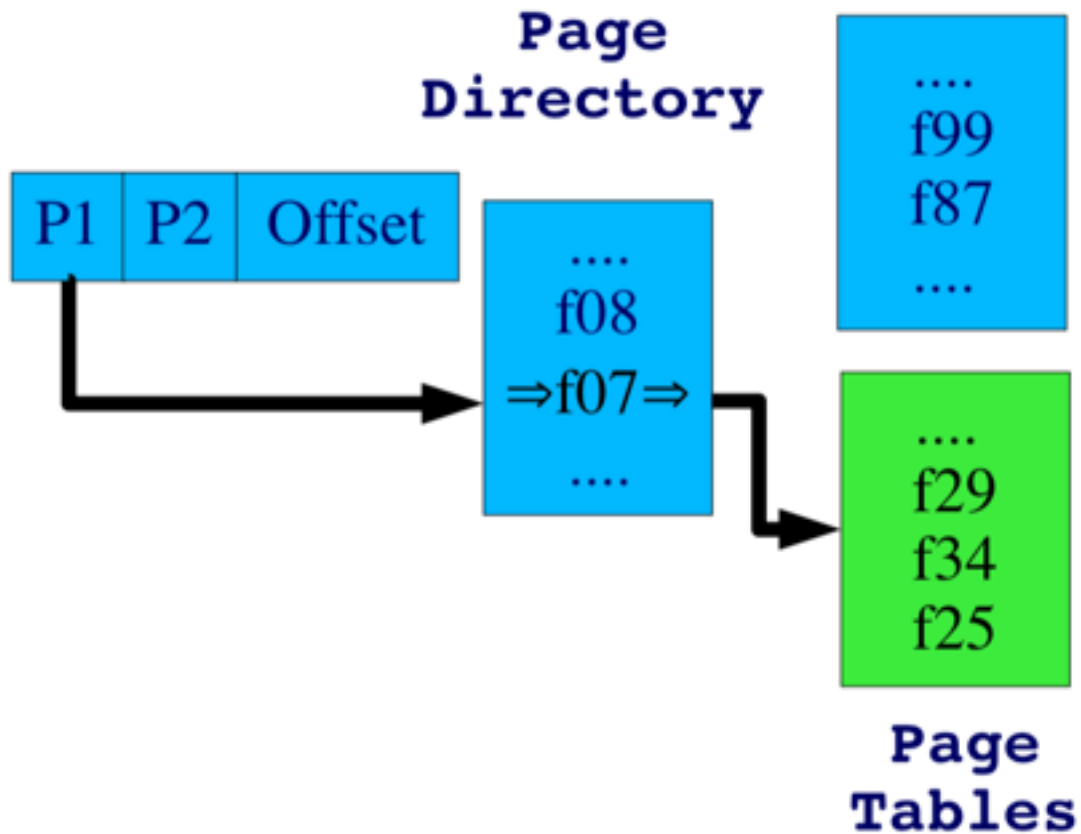
Multi-level Page Table



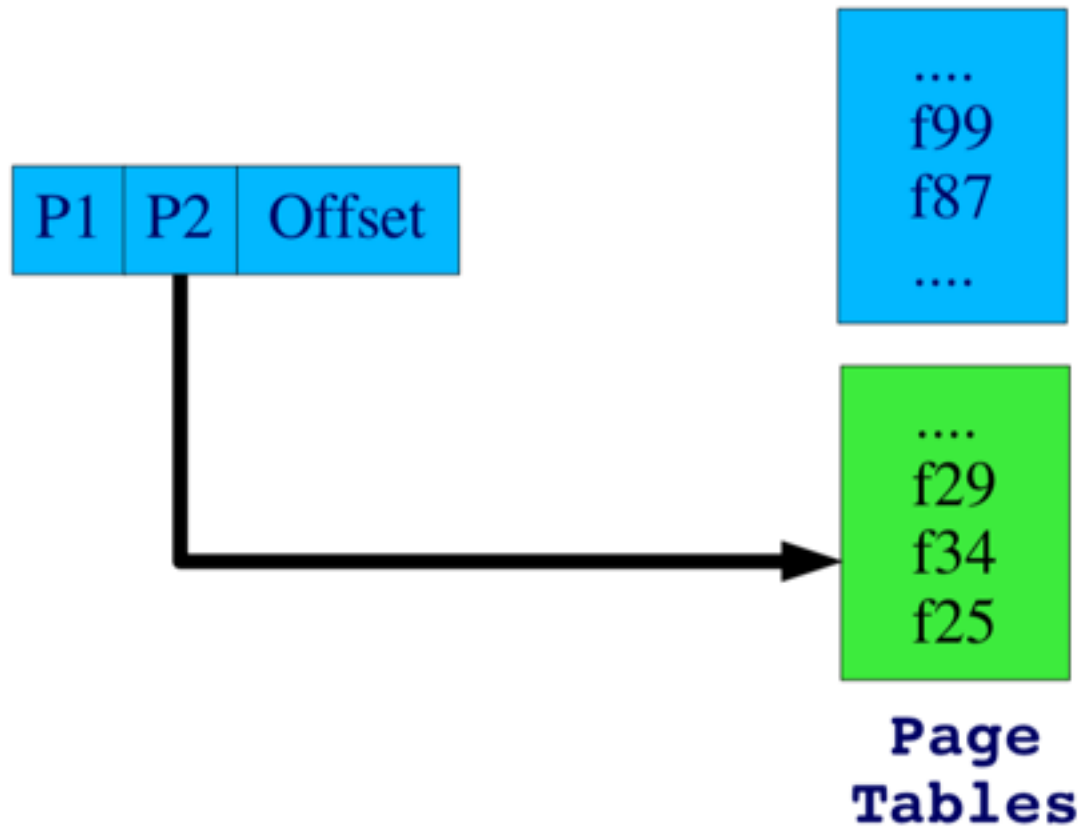
Multi-level Page Table



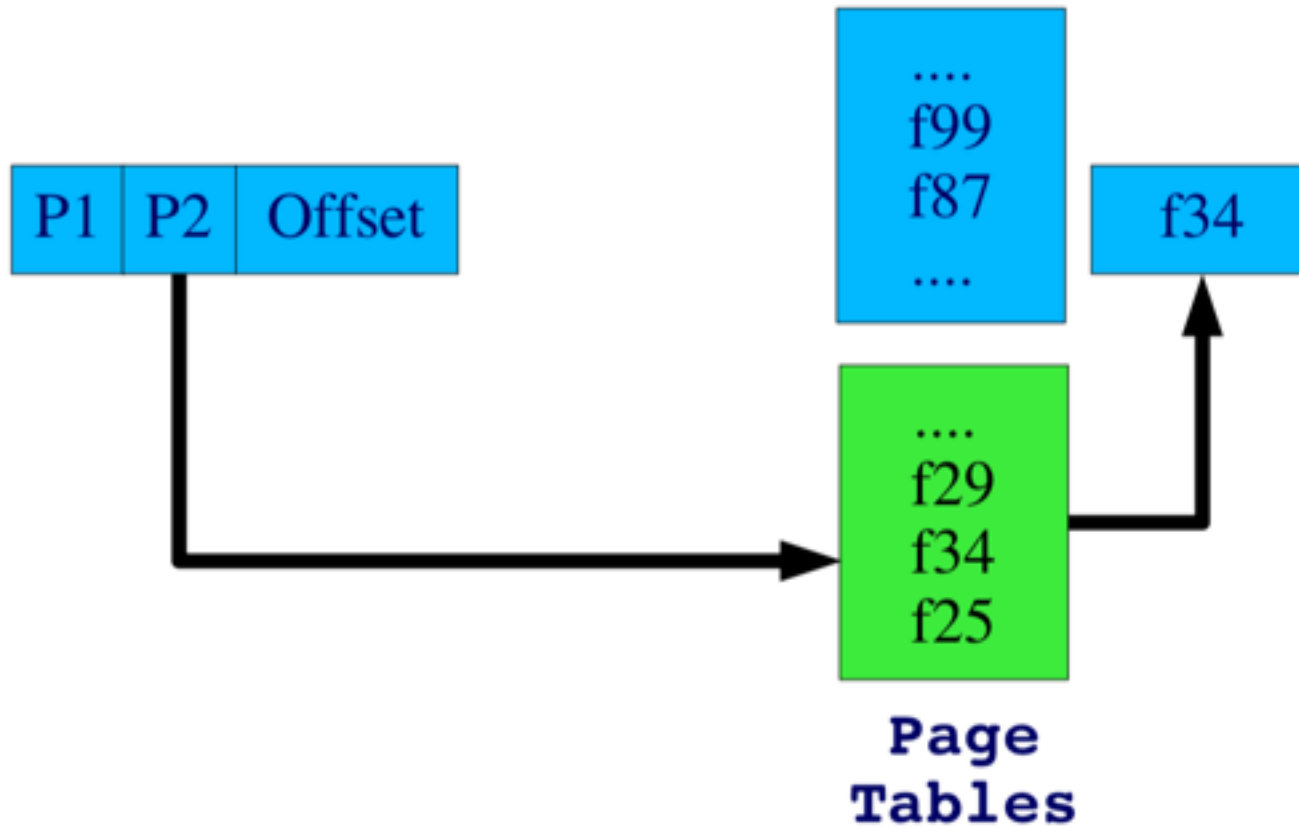
Multi-level Page Table



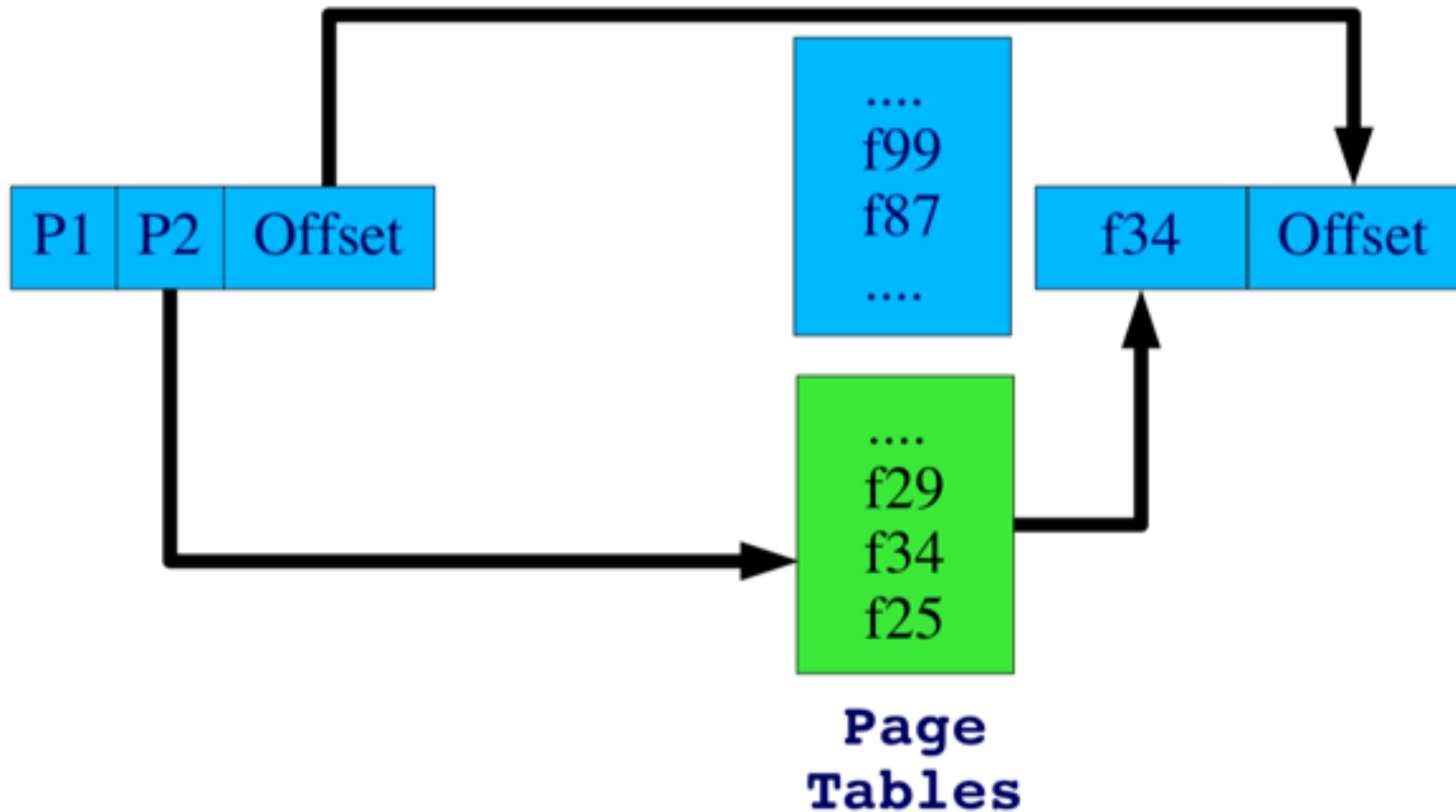
Multi-level Page Table



Multi-level Page Table



Multi-level Page Table



Multi-level Page Table

Assume 4 KByte pages, 4-byte PTEs

- Ratio: 1024:1
 - 4 GByte virtual address (32 bits) \Rightarrow 4 MByte page table

Now assume page *directory* with 4-byte PDEs

- 4-megabyte page table becomes 1024 4K page tables
- Plus one 1024-entry page directory to point to them
- Result: _____

Multi-level Page Table

Assume 4 KByte pages, 4-byte PTEs

- Ratio: 1024:1
 - 4 GByte virtual address (32 bits) \Rightarrow 4 MByte page table

Now assume page *directory* with 4-byte PDEs

- 4-megabyte page table becomes 1024 4K page tables
- Plus one 1024-entry page directory to point to them
- Result: 4 Mbyte + 4Kbyte

Multi-level Page Table

“Sparse” page directory

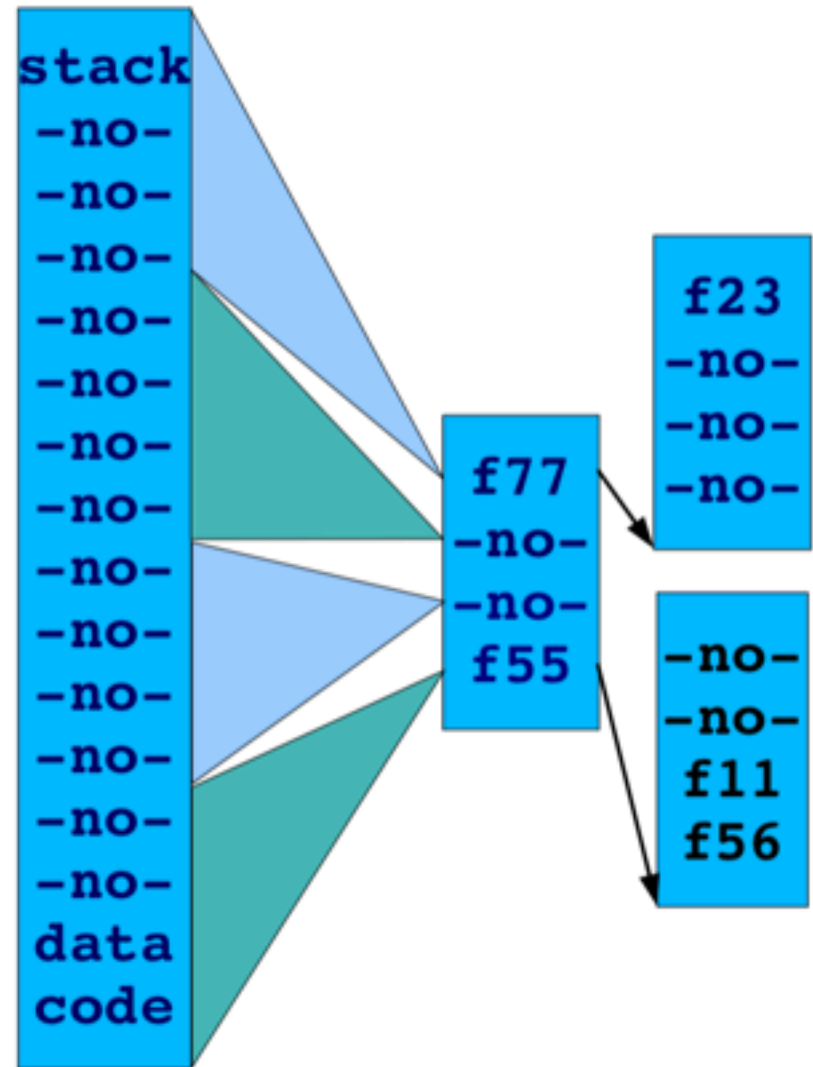
- Pointers to non-empty PT's
- “Null” instead of empty PT

Common case

- Need 2 or 3 page tables
 - One or two map code & data
 - One maps stack
- Page directory has 1024 slots
 - 2-3 point to PT's
 - Remainder are “not present”

Result

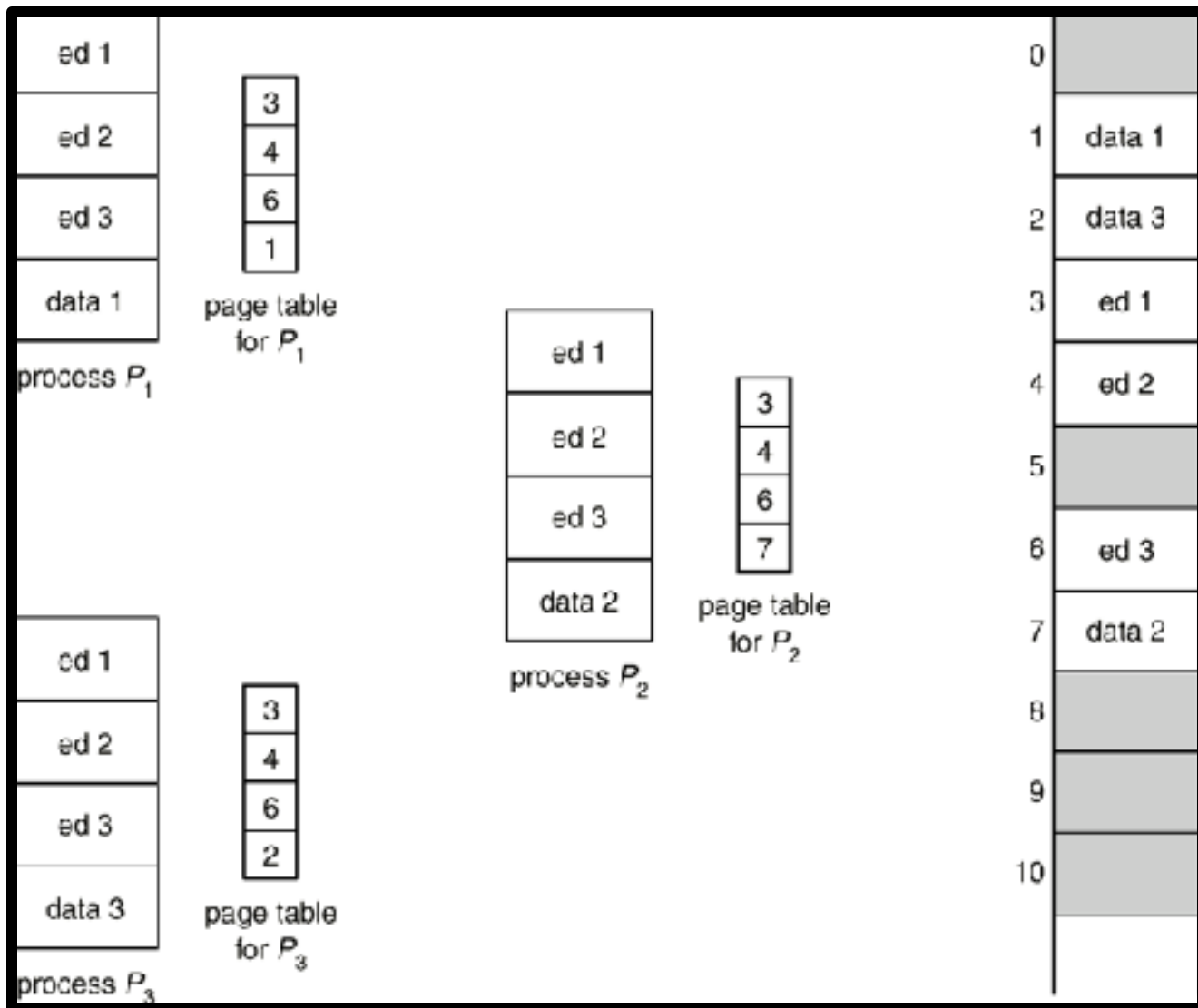
- 2-3 PT's, 1 PD
- Map entire address space with 12-16Kbyte, not 4Mbyte



共享页

- 共享代码
 - 一段只读（可重入）代码副本可由进程共享。
 - 共享代码出现在进程的逻辑地址空间的相同位置。
 - 如：文本编辑器，窗口系统等
- 私有代码和数据
 - 每个进程保留代码和数据的私有副本。
 - 私有代码和数据的页可以出现在逻辑地址空间的任何地方。

在分页环境下的代码共享



❓ 代码:
150KB

❓ 数据: **50KB**

❓ 进程数:
40

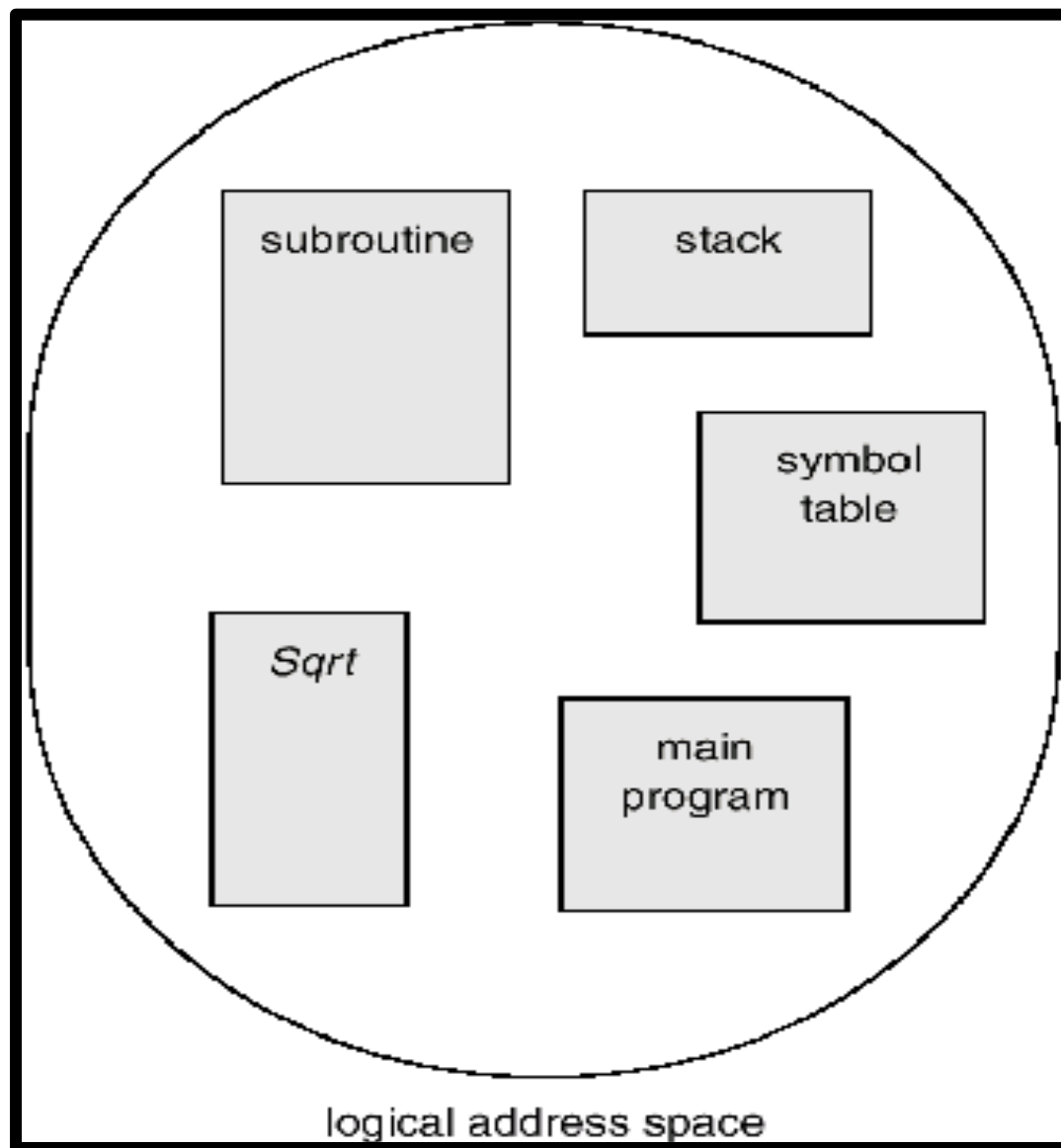
❓ 非共享:
 **40×200
 $= 8000$**

❓ 共享:
 **$40 \times 50 + 150$
 $= 2150$**

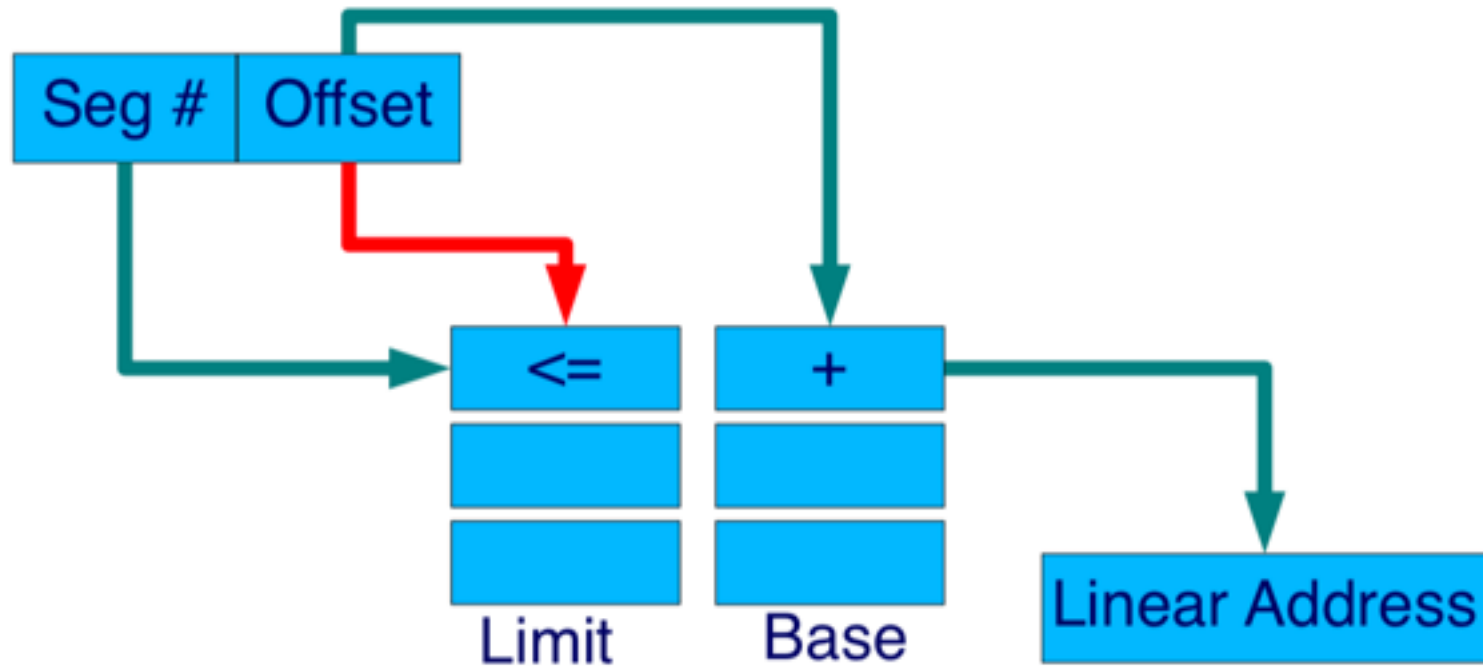
分段式存储管理

- 为提高主存空间的利用率，存储管理方式
 - 固定分区→动态分区→分页方式
- 为满足程序设计和开发的要求（为模块为单位的装配、共享和保护），出现了分段式存储管理
 - 一个程序是一些段的集合，一个段是一个逻辑单位，如：
 - 主函数
 - 过程，函数，方法，对象
 - 局部变量，全局变量
 - 堆栈
 - 符号表，数组

用户对一个程序的认知



Segment



分段式存储管理

- 基本概念
 - 逻辑地址：段号和段内地址
 - 段表、作业表

段表

第0段

第1段

始址	长度
100k	8k
256k	16k
...	...

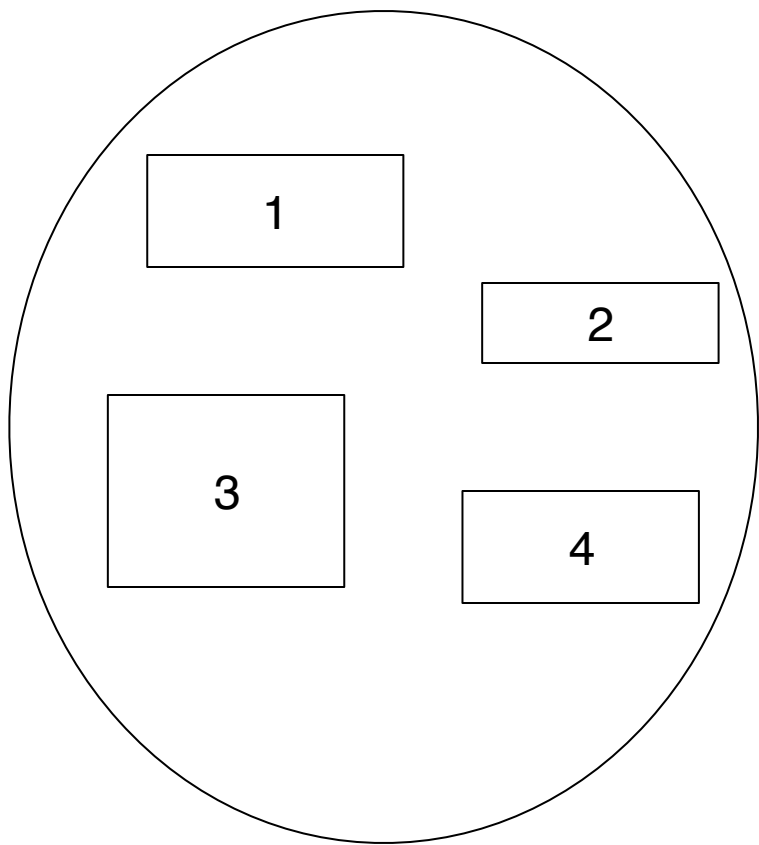
作业表

作业名	段表始址	段表长度
Job1	0	2
...

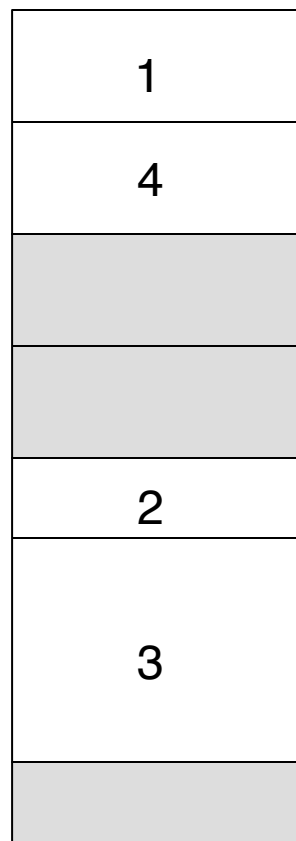
分段式存储管理

- 实现：
 - 可基于可变分区存储管理的原理，以段为单位进行主存分配
- 段共享：
 - 通过不同作业段表中的项指向同一个段基址来实现。

分段的例子



逻辑地址空间

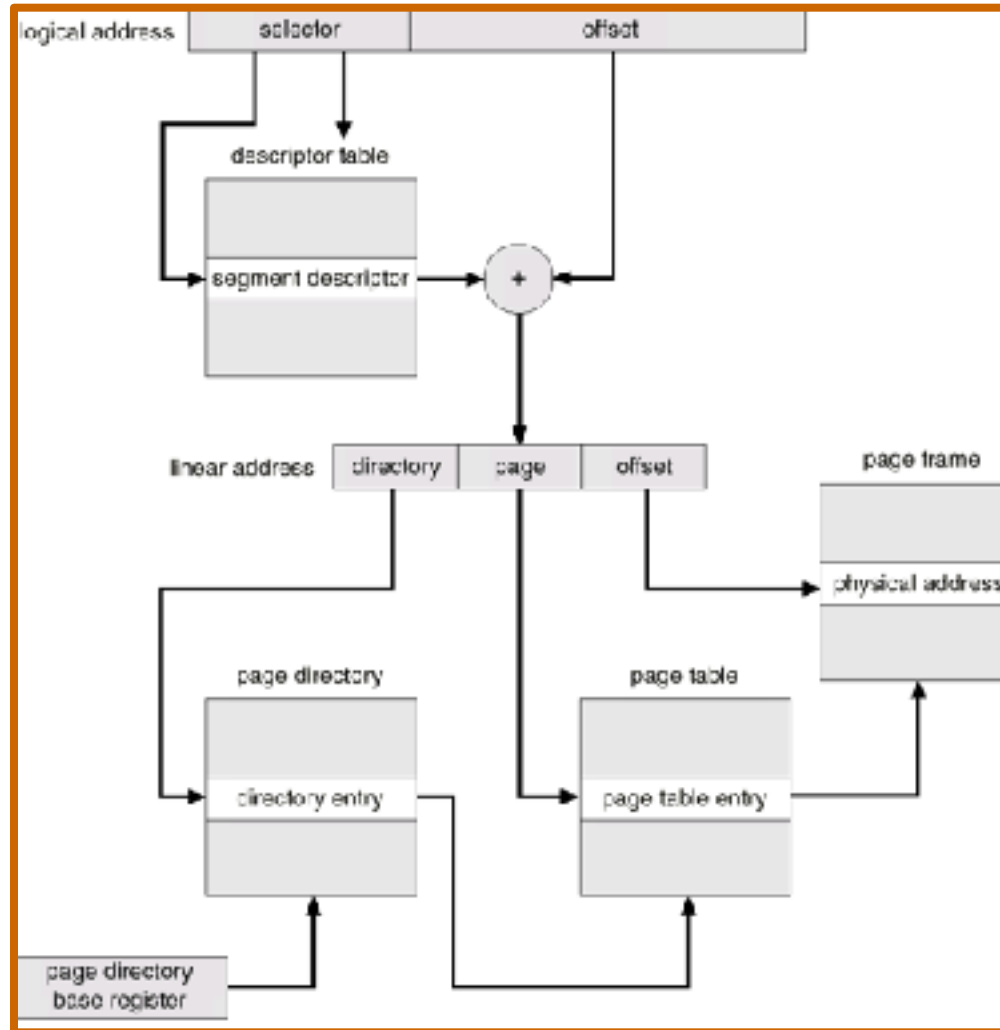


物理内存

分页和分段的主要区别❓

- (1) 页是信息的物理单位，分页是由于系统管理的需要，减少外部碎片，提高内存利用率。段则是信息的逻辑单位，它含有一组其意义相对完整的信息，是为了能更好地满足用户的需要。
- (2) 页的大小固定且由系统决定，由系统把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而在系统中只能有一种大小的页面；而段的长度却不固定，决定于用户所编写的程序，通常由编译程序在对源程序进行编译时，根据信息的性质来划分。❓
- (3) 分页的作业地址空间是一维的，即单一的线性地址空间；而分段的作业地址空间则是二维的，程序员在标识

Intel 30386 Address Translation



Linux on Intel 80x86

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 segments:
 - Kernel code
 - Kernel data
 - User code (shared by all user processes, using logical addresses)
 - User data (likewise shared)
 - Task-state (per-process hardware context)
 - LDT
- Uses 2 protection levels:
 - Kernel mode
 - User mode

小结

比较不同内存管理策略（连续分配、分页、分段）时，考虑以下方面

- 硬件支持
 - 寄存器, 页表, 段表
- 性能
 - 快速寄存器、页表、**TLB**
- 碎片
 - 固定分区，内部碎片
 - 多个分区、分段，外部碎片

小结

- 重定位
 - 紧缩，在内存中移动程序
 - 要求在执行时逻辑地址能动态重定位
- 共享
 - 要求分页或分段
- 保护
 - 页、段表中的保护位
 - 基址、限长寄存器

虚拟内存

背景

- 虚拟内存：
 - 事实
 - 数组、链表和表通常分配了比实际所需要更多的内存。
 - 程序的某些选项或特点可能很少使用。即使需要完整程序，也并不是在某时刻同时需要
 - 优点
 - 保存部分程序在内存中，可运行一个比物理内存大的多的程序
 - 逻辑地址空间能够比物理地址空间大
 - 可以有更多程序同时运行
 - 允许若干个进程共享地址空间
 - 进程创建高效
 - 实现
 - 请求分页存储管理(Demand paging)
 - 请求分段存储管理(Demand segmentation)
 - 请求段页式存储管理

局部性原理：

(1) 时间局部性。如果程序中的某条指令一旦执行，则不久以后该指令可能再次执行；如果某数据被访问过，则不久以后该数据可能再次被访问。产生时间局限性的典型原因，是由于在程序中存在着大量的循环操作。

(2) 空间局部性。一旦程序访问了某个存储单元，在不久之后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定的范围之内，其典型情况便是程序的顺序执行。

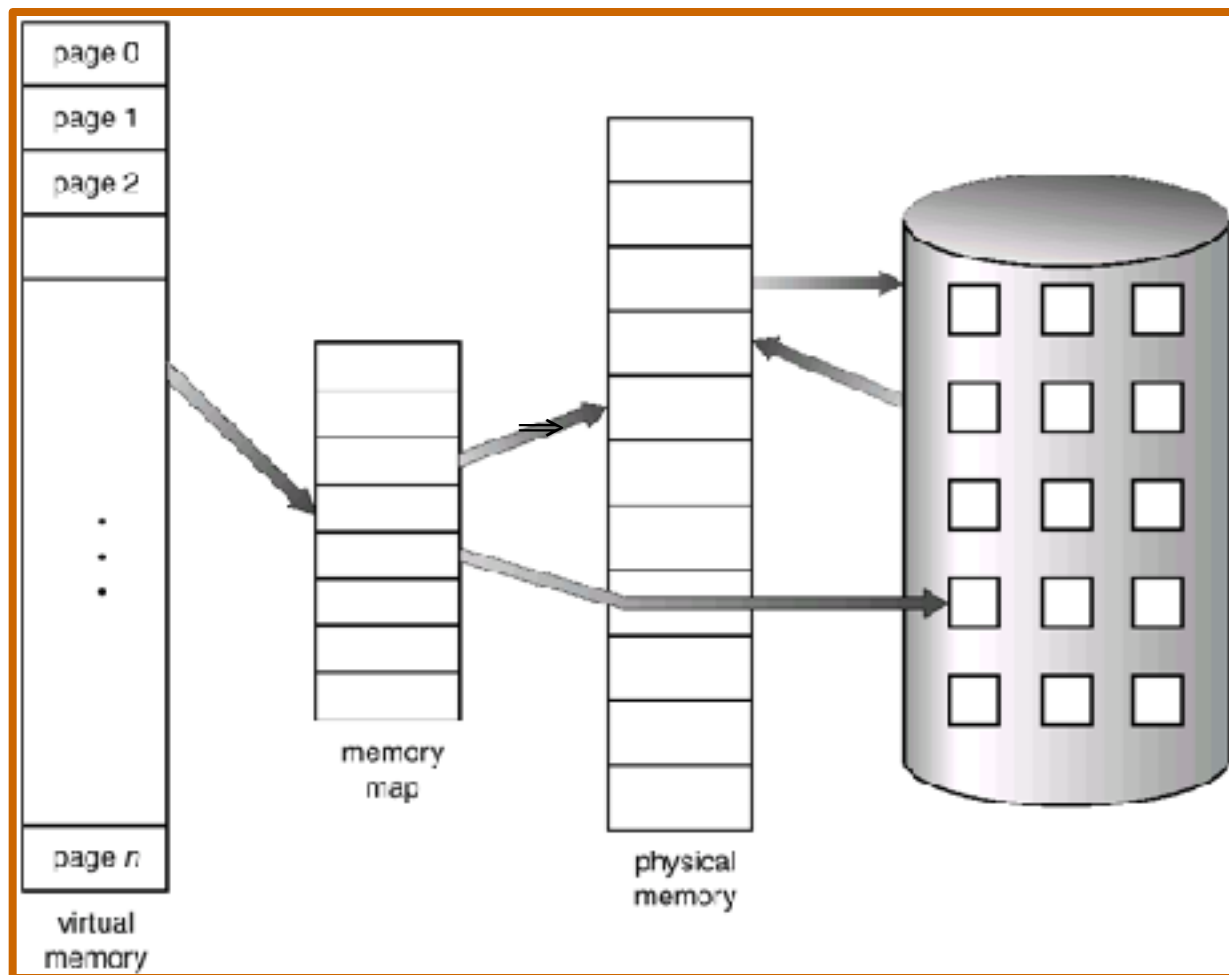
虚拟存储管理

- 虚拟存储器实现的理论基础
 - 作业为什么能够部分装入和部分对换？
 - 作业信息（程序）的局部性，使得在作业信息不完全装入主存的情况下能够保证其正确运行
 - 空间局部性，一段时间内，仅访问程序代码和数据的一小部分
 - 时间局部性，最近访问过的程序代码和数据，很快又被访问的可能性很大

虚拟存储管理

- 虚拟存储管理与对换技术的区别
 - 虚拟存储管理
 - 以页或段为单位处理
 - 进程所需主存容量大于当前系统空闲量时仍能运行
 - 对换技术（中级调度，挂起和解除挂起）
 - 以进程为单位处理
 - 进程所需主存容量大于当前系统空闲量时，无法解除挂起

示意图



虚拟存储管理

- 请求分页式存储管理
 - 分页式存储管理技术的扩展，是一种常用的分页式虚拟存储管理技术
 - 基本原理：
 - 将作业信息被分为多个页面，其副本存放在辅助存储器中。当作业被调度运行时，仅装入需要立即访问和使用的页面，在执行过程中如果需要访问的页面不在主存中，则将其动态装入。

虚拟存储管理

- 请求分页式存储管理
– 页表的扩展

1表示在主存中，
0表示不在主存中



页号	驻留标志	页框号	辅存地址	其他标志
...	1
...	0
...

其他标志：

缺页标志、脏页标志、访问标志、锁定标志、淘汰标志等

虚拟存储管理

- 请求分页式存储管理

- 硬件支撑

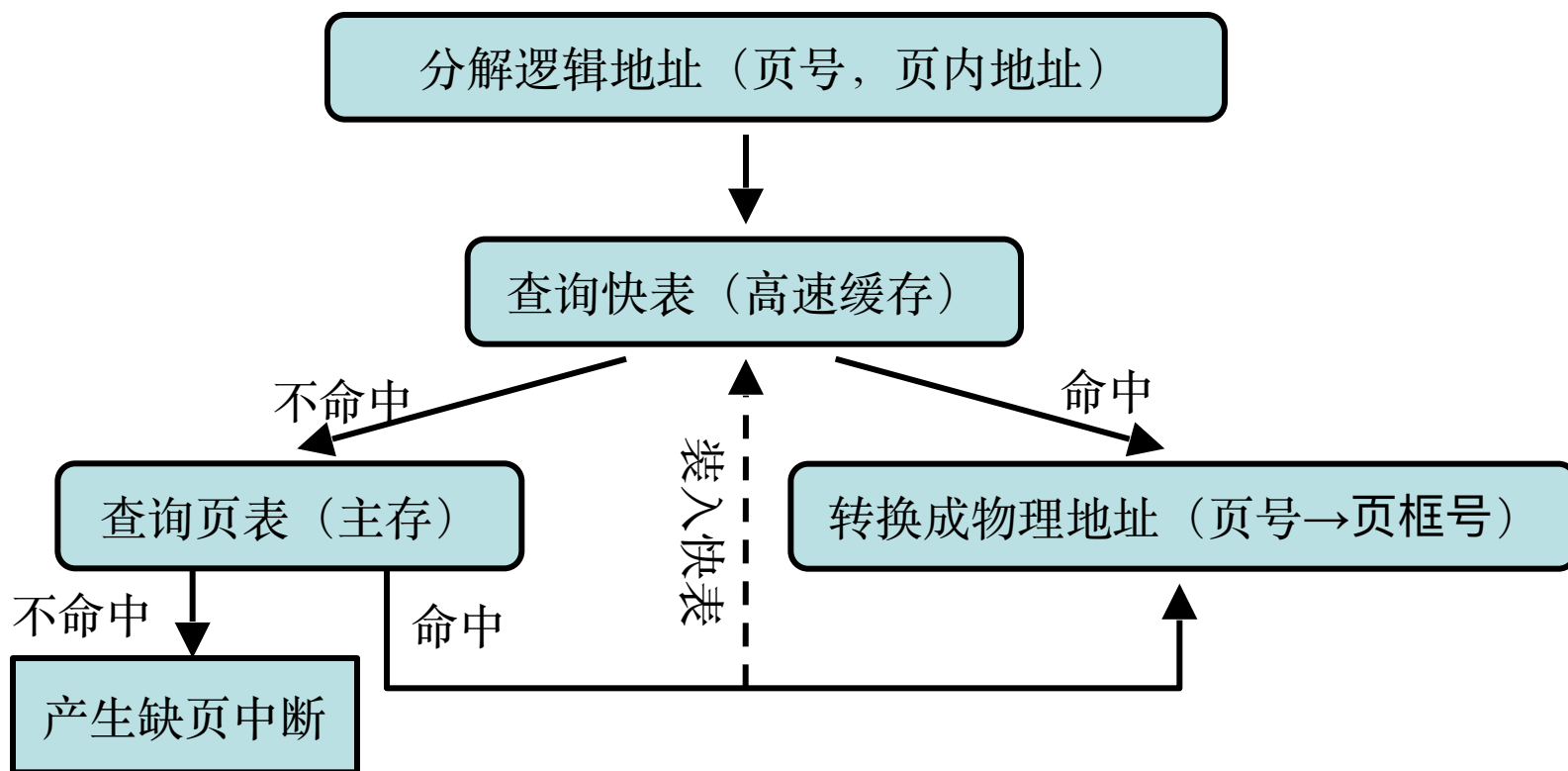
- 操作系统的存储管理需要依靠低层硬件的支撑来完成，该硬件称为主存管理单元MMU。
 - MMU的主要功能，完成逻辑地址到物理地址的转换，并在转换过程中产生相应的硬件中断（缺页中断、越界中断）
 - MMU的主要组成：
 - 页表基址寄存器
 - 快表TLB

请求分页式存储管理

- 需要调页的时候才把它换入内存.
 - 需要很少的I/O
 - 需要很少的内存
 - 响应快
 - 多用户
- 需要页面调度时⇒查阅此页的引用
 - 无效引用⇒中止
 - 不在内存⇒将其调入内存
 - 缺页中断
 - 找一个空闲帧
 - 将需要的页调入空闲帧
 - 重置页表，有效位为1
 - 重启指令

虚拟存储管理

- 请求分页式存储管理
 - 硬件支撑
 - MMU的工作流程



Page Replacement 基本步骤

1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim** frame
3. Read the desired page into the (newly) free frame.
Update the page and frame tables.
4. Restart [continue] the process

缺页

- 当需要调页但没有空闲帧时?
- **Swapping**: 把内存中暂时不能运行的进程或者暂时不用的程序和数据, 调出到外存, 腾出足够的内存空间, 再把已具备运行条件的进程或进程所需要的程序和数据, 调入内存。
 - 数据结构: 其形式与内存的动态分区分配方式中所用数据结构相似, 空闲分区表或空闲分区链。
 - 在空闲分区表中的每个表目中应包含两项, 即对换区的首址及其大小, 它们的单位是盘块号和盘块数。
- Swapping可有效提高内存利用率。

虚拟存储管理

- 虚拟存储管理需要解决的主要问题
 - 主存和辅存的统一管理问题
 - 逻辑地址到物理地址的转换问题
 - 部分装入和部分对换问题

虚拟存储管理

- 请求分页式存储管理

- 页面装入策略，何时将一个页面装入主存？

- 请求式调入，缺页中断驱动，一次调入一页
 - 预调式调入，按某种预测算法动态预测并调入若干页面）

- 消除策略，何时将修改过的页面写回辅存？

- 请求式清除，仅当一页被选中进行替换时，该页内容已修改则写回辅存。（清除与替换成对）
 - 预约式清除，内容被修改页面成批写回辅存，写回操作在该页面被替换前，而非替换时。

例子：请求分页性能

- 缺页概率 $0 \leq p \leq 1.0$
 - $p = 0$ 无缺页
 - $p = 1$, 全部缺页

- 有效访问时间 (Effective Access Time, EAT)

$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{内存访问时间} \\ & + p (\text{页错误开销} \\ & + \text{换出开销} \\ & + \text{换入开销} \\ & + \text{重启开销}) \end{aligned}$$

“换出开销” = “换入开销” x “probability it has been changed”

例子：请求分页性能

- 内存访问时间= 1 usec
- 内存中50% 被选为参与调页的页已被修改，调页时需换出
- 页替换时间 = 10,000 usec

$$\begin{aligned} \text{EAT} &= (1 - p) \times 1\text{usec} + p (1 + 0.50) 10000 \text{ usec} \\ &= 1 + 14999 \times p \quad (\text{in usec}) \end{aligned}$$

如果缺页率为 $p = 0.1\% (0.001)$, then

$$\text{EAT} = 16 \text{ usec} \quad (16 \text{ 倍于内存访问时间})$$

[Why will adding more memory speed up your PC?]

Demand Paging Example

- **有效访问时间** = Hit Rate x Hit Time + Miss Rate x Miss Time
- 示例
 - 内存访问时间 = 200 nanoseconds
 - 缺页时的页替换时间 = 8 milliseconds
 - 缺页率 p
 - 有效访问时间 = $(1 - p) \times 200\text{ns} + p \times 8\text{ ms}$
 $= (1 - p) \times 200\text{ns} + p \times 8,000,000\text{ns}$
 $= 200\text{ns} + p \times 7,999,800\text{ns}$
- 如果缺页率为0.1%,有效访问时间 = 8.2 μs :
 - 访问速度比内存访问速度慢40倍!
- 如果要求访问速度比内存慢10%?
 - $200\text{ns} \times 1.1 < \text{EAT}$
 - $\Rightarrow p < 2.5 \times 10^{-6}$
 - 即缺页率为 1/400000!

Demand Paging: Extended

Effective access time of memory word

- $(1 - p_{\text{miss}}) * T_{\text{memory}} + p_{\text{miss}} * T_{\text{disk}}$

Textbook example (a little dated)

- T_{memory} 100 ns
- T_{disk} 25 ms
- $p_{\text{miss}} = 1/1,000$ slows down by factor of 250
- slowdown of 10% needs $p_{\text{miss}} < 1/2,500,000!!!$

Demand Paging: Extended

fork() produces two *very*-similar processes

- Same code, data, stack

Expensive to copy pages

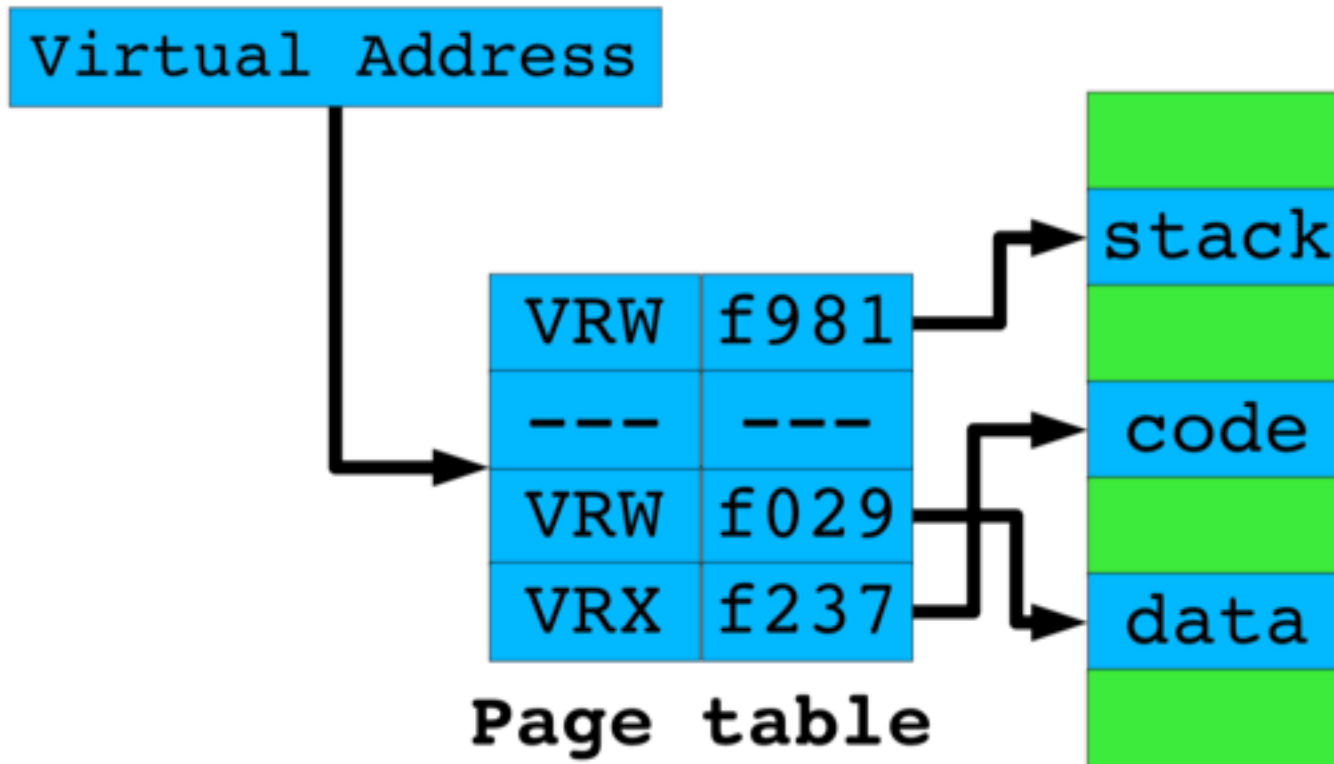
- Many will never be modified by new process
 - Especially in fork(), exec() case

***Share* physical frames instead of copying?**

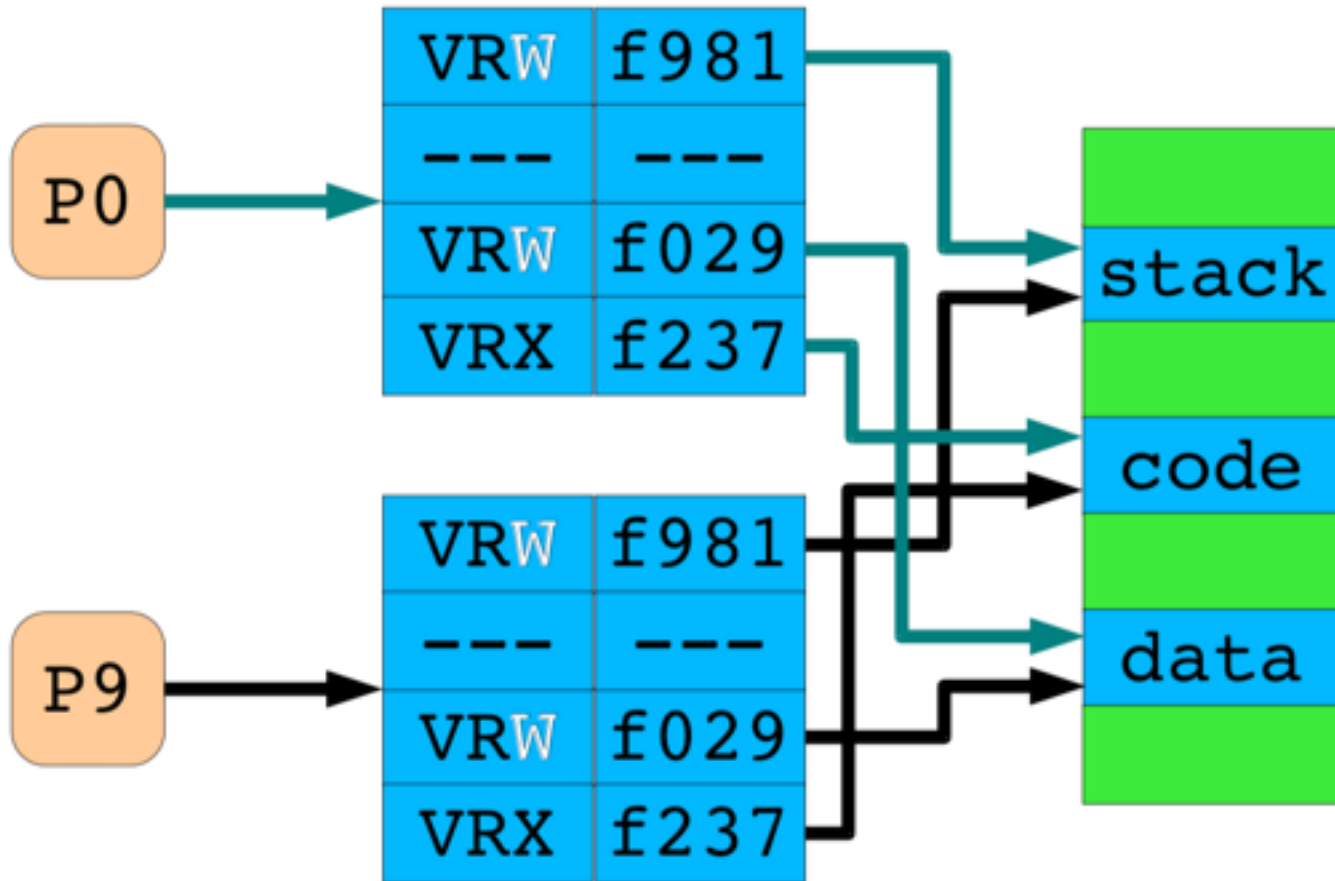
- Easy: code pages – read-only
- Dangerous: stack pages!

How to speedup?

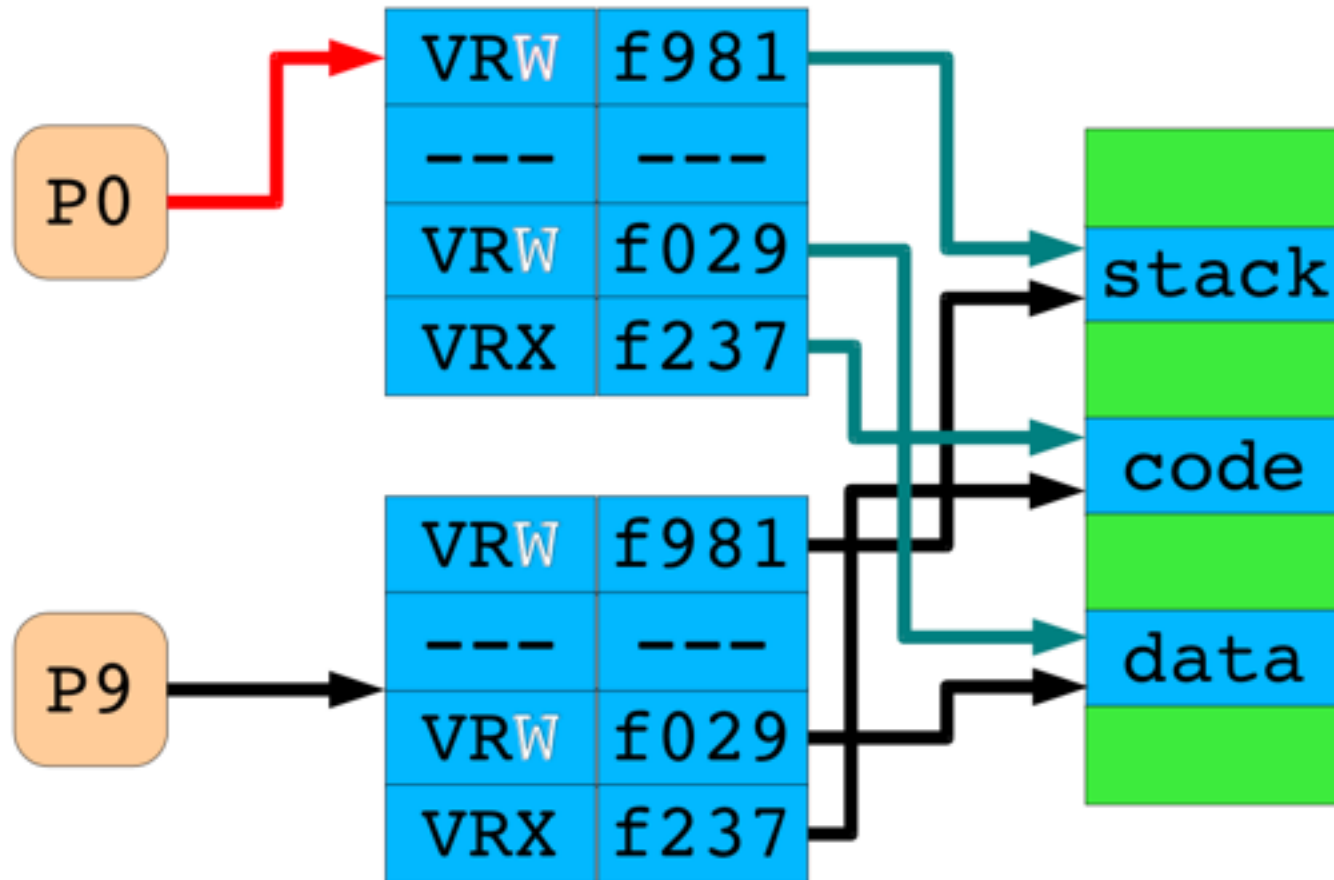
🐮: Copy on Write (COW)



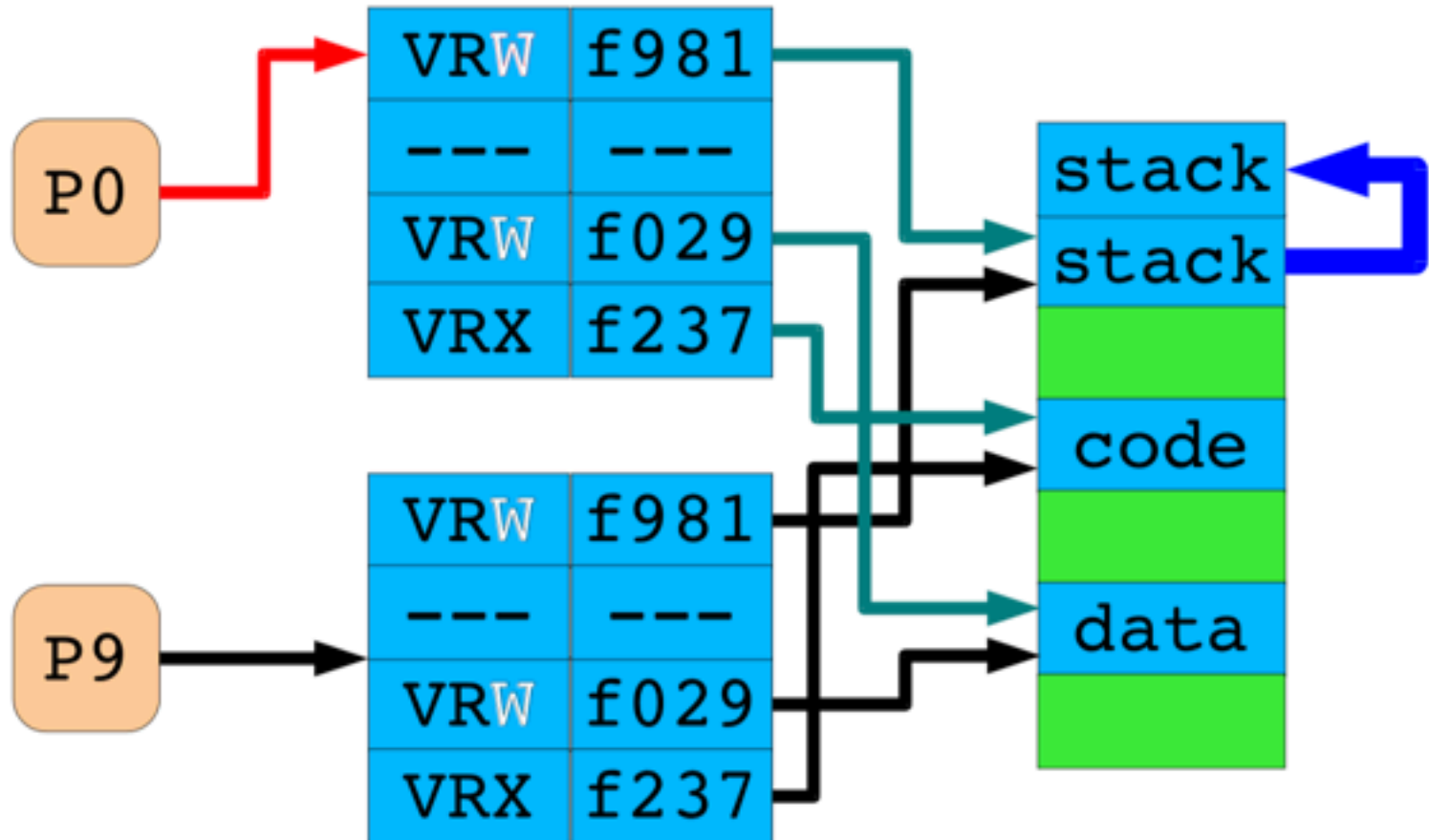
🐮: Copy on Write (COW)



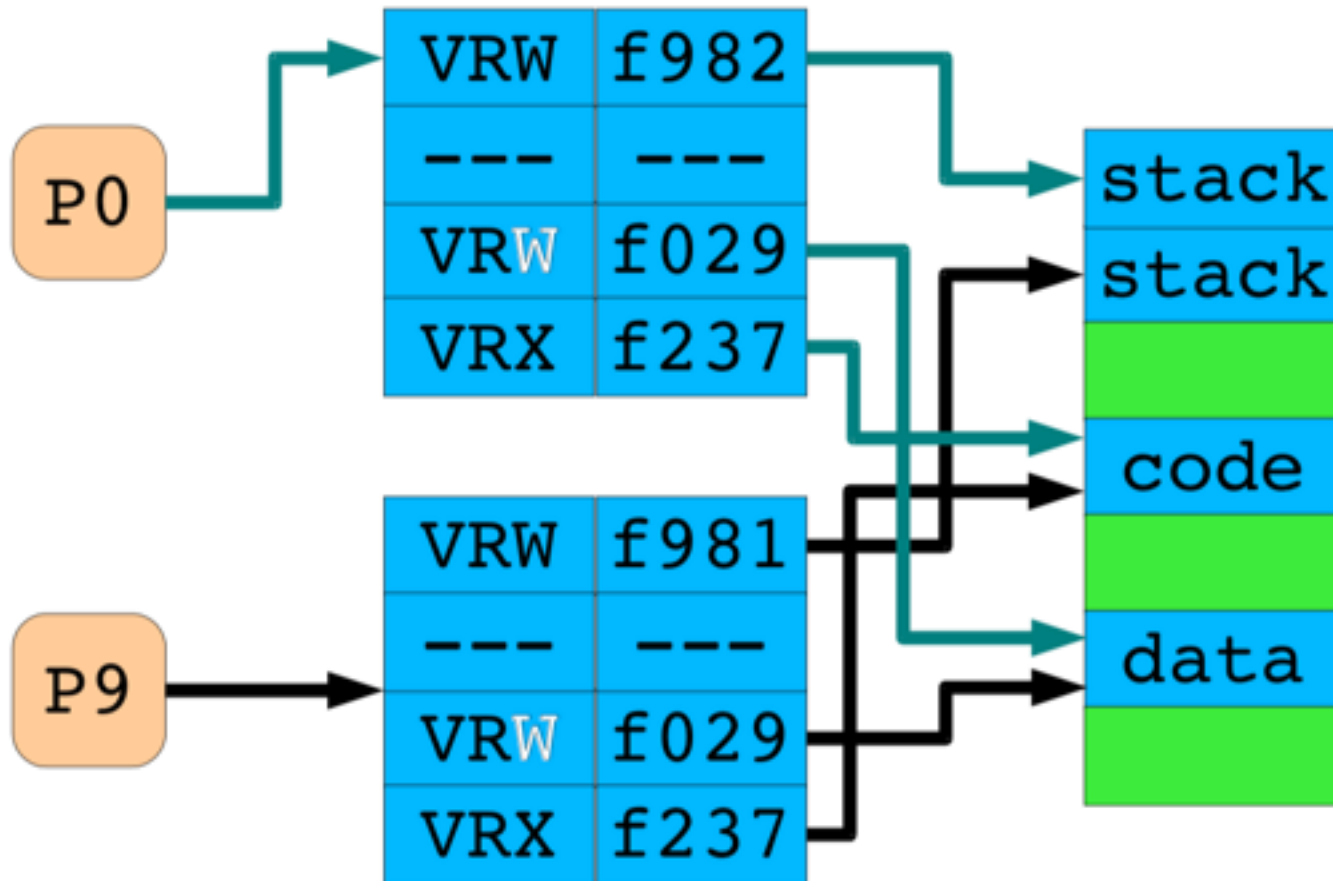
🐮: Copy on Write (COW)



🐮: Copy on Write (COW)



🐮: Copy on Write (COW)



请求分页式存储管理

- 为实现请求分页式存储管理须解决两个问题
 - 帧分配算法: 给每个进程分配多少帧
 - 页替换算法: 怎样选择要替换的帧
- 主要包括: 主存和辅存的统一管理, 逻辑地址到物理地址的转换, 部分装入和部分对换问题
- 磁盘 I/O 非常费时, 为提高系统性能- >降低缺页率

虚拟存储管理

- 请求分页式存储管理
 - 影响缺页中断率的因素：
 - 页面替换算法
 - 主存页框数
 - 页面大小
 - 程序特性

虚拟存储管理

- 请求分页式存储管理

- 帧分配策略

- 固定分配，在一个进程的生命周期中，分配给它的帧数固定。
 - 平均分配、按比例分配、优先权分配
 - 可变分配，在一个进程的生命周期中，当进程缺页次数较多时，分配给它较多的帧，反之，则分配给较少的帧。

- 页面替换策略

- 局部替换策略，通常与固定分配结合使用
 - 全局替换策略，通常与可变分配结合使用

帧分配算法

平均分配算法

这是将系统中所有可供分配的帧，平均分配给各个进程。例如，当系统中有100个帧，有5个进程在运行时，每个进程可分得20个帧。这种方式貌似公平，但实际上是不公平的，因为它未考虑到各进程本身的大小。如有一个进程其大小为200页，只分配给它20个帧，这样，它必然会有很高的缺页率；而另一个进程只有10页，却有10个帧闲置未用。

按比例分配算法

这是根据进程的大小按比例分配帧的算法。如果系统中共有 n 个进程，每个进程的页数为 S_i ，则系统中各进程页数的总和为：

$$S = \sum_{i=1}^n S_i$$

又假定系统中可用的帧总数为 m ，则每个进程所能分到的帧数为 b_i ，将有：

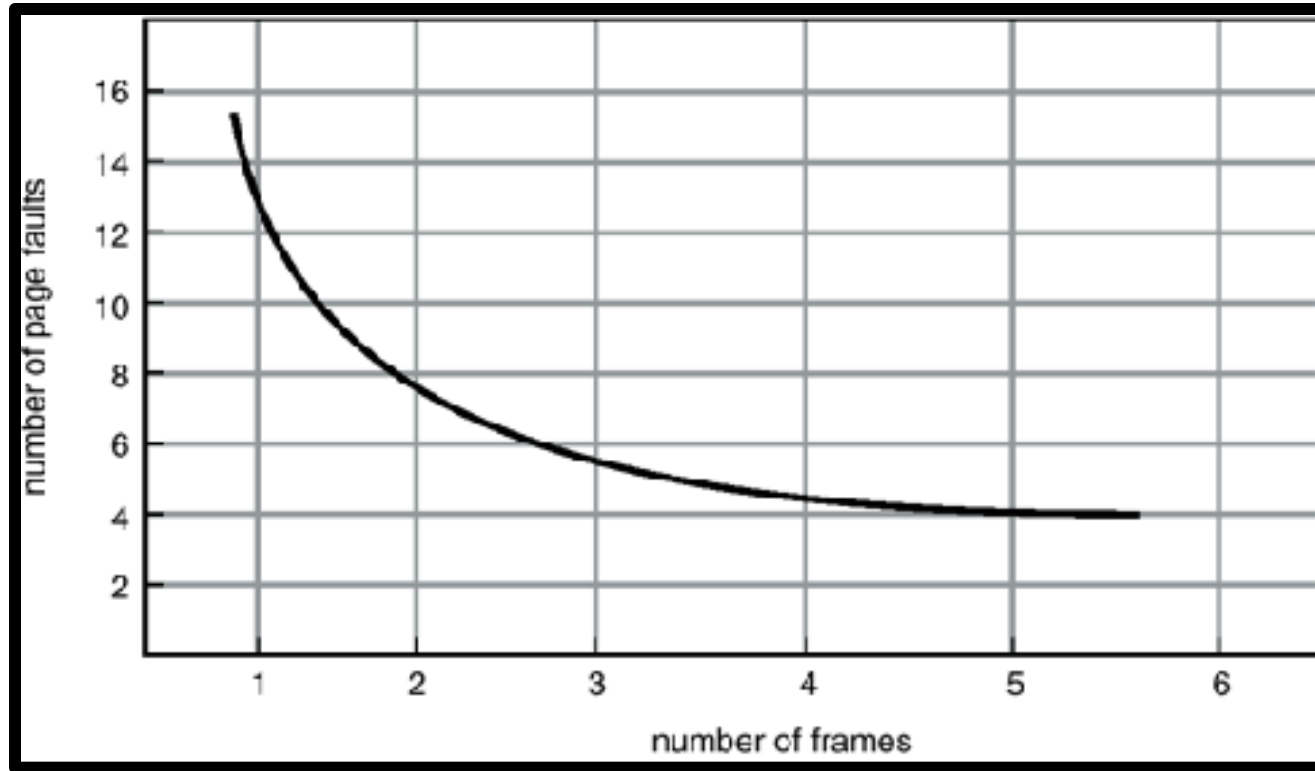
$$b_i = \frac{S_i}{S} \times m$$

b 应该取整，它必须大于系统最小帧数。

考虑优先权的分配算法❓

在实际应用中，为了照顾到重要的、紧迫的作业能尽快地完成，应为其分配较多的内存空间。通常采取的方法是把内存中可供分配的所有帧分成两部分：一部分按比例地分配给各进程；另一部分则根据各进程的优先权，适当地增加其相应份额后，分配给各进程。在有的系统中，如重要的实时控制系统，则可能是完全按优先权来为各进程分配其帧的。

理想的缺页与帧数关系图



- ❑ 给定一串内存引用序列:
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
- ❑ 如果有三帧: 3 次缺页
- ❑ 如果只有一帧: 11 次缺页

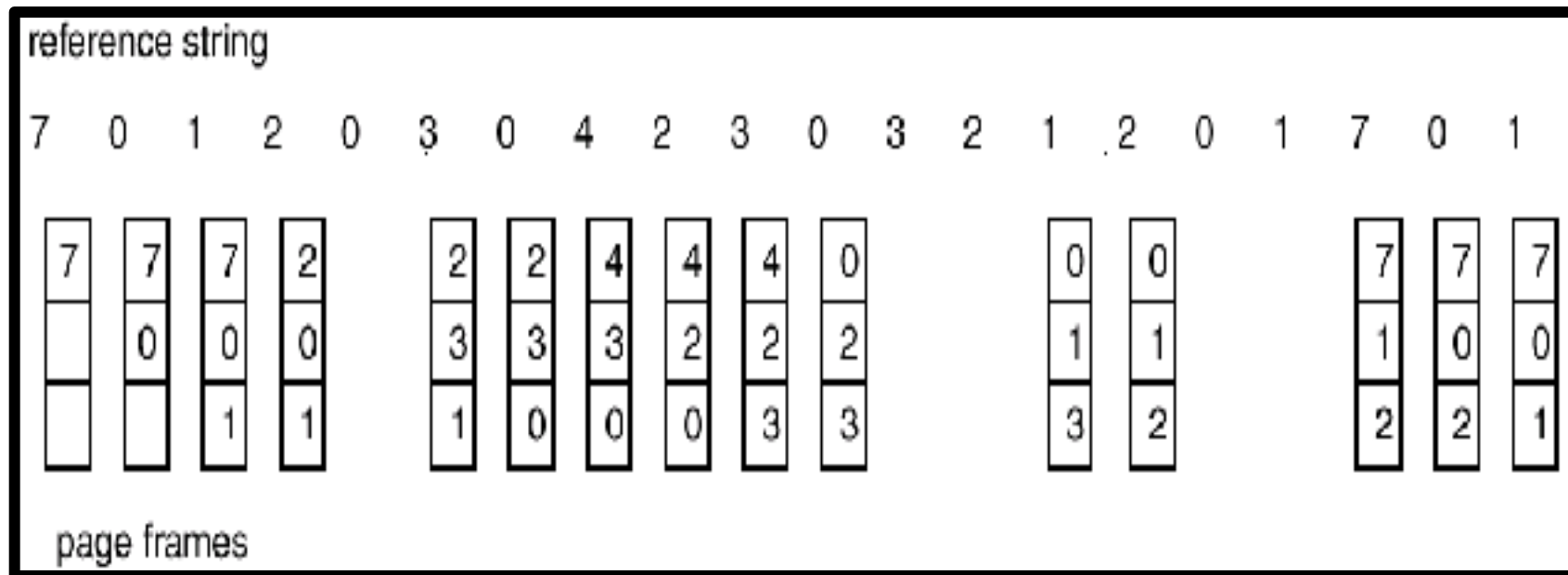
页替换算法

- FIFO
- 最优页替换
- 随机替换
- LRU
- 其他
- 如何评价页面替换算法
 - 应尽量避免“抖动”现象的出现
 - 衡量指标——缺页率 f : 受页替换算法、主存帧数、页大小、程序等影响

FIFO 页替换

- **FIFO**算法:

- 可以创建一个**FIFO** 队列来管理内存中的所有页
- 调入页时，将它加到队列的尾部
- 当必须替换一页时，将选择最旧的页



总共 **15** 次缺页

最优页替换算法

- 被替换的页是未来最长时间不被使用的页
- 很难实现，因为需要未来的知识
- 4帧的例子

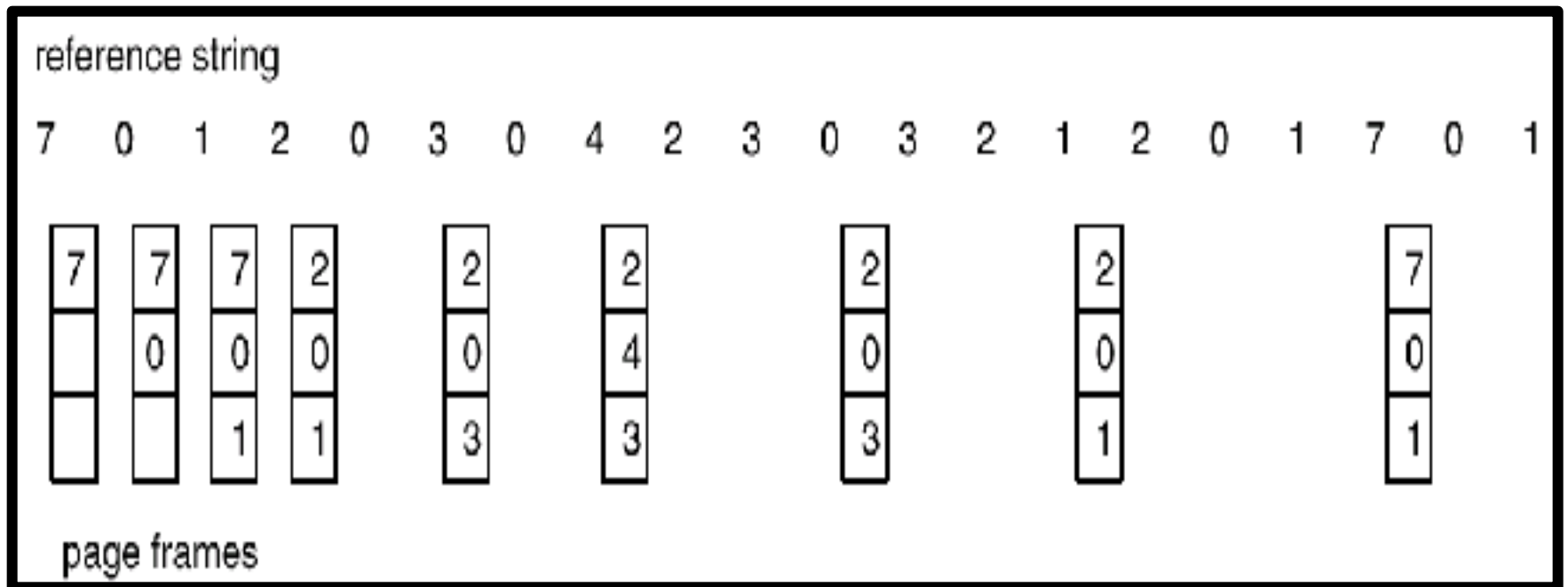
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

6 page faults

最优页替换的作用：用来衡量其他算法的性能

最优页替换



total 9 page faults

随机页替换算法

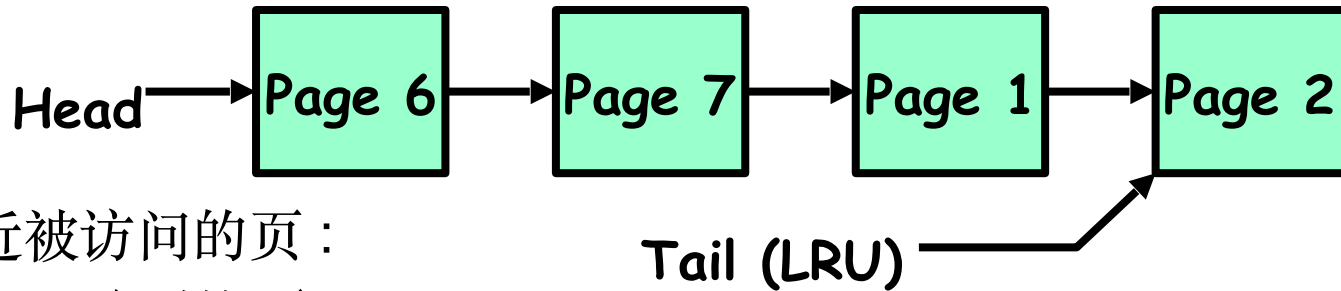
- 被替换的页是随机选择的页
 - TLB采用的一种应用方式，硬件结构简单
- 性能难以预测，实时性难以保证

Least Recently Used (LRU) 页替换

- LRU替换算法为每个页记录该页最后的使用时间
- 当必须进行页替换时，LRU选择最近最长未被使用的页。

LRU页替换

- 栈实现
 - 在一个双链表中保留一个记录页数目的栈



- 最近被访问的页：
 - 移到栈顶
 - 最坏情况下需要改变6个指针
- 无需查找，栈底部即是要找的页

Example: FIFO

- 3 frames, 4页, and following reference stream:
– A B C A B D A D B C B

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 次缺页.
- When referencing D, replacing A is bad choice, since need A again right away

Example: 最优页替换

- 考虑以下序列:
– **A B C A B D A D B C B**
- 最优页替换:

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

- 缺页: 5 次
- Where will D be brought in? Look for page not referenced farthest in future.
- What will LRU do?
 - Same decisions as MIN here, but won't always be true!

LRU 性能可能差?

- 考虑以下序列: **A B C D A B C D A B C D**
- LRU (same as FIFO here):

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

– Every reference is a page fault!

- 在该例中最优页替换性能好很多:

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

Adding memory => 降低缺页率?

- Yes for LRU and MIN
- Not necessarily for FIFO! (Belady's anomaly)

Ref:	A	B	C	D	A	B	E	A	B	C	D	E
Page:	1	A		D			E					
2		B			A					C		
3			C			B					D	

Ref:	A	B	C	D	A	B	E	A	B	C	D	E
Page:	1	A					E				D	
2		B						A				E
3			C						B			
4				D						C		

- After adding memory:
 - With FIFO, contents can be completely different
 - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

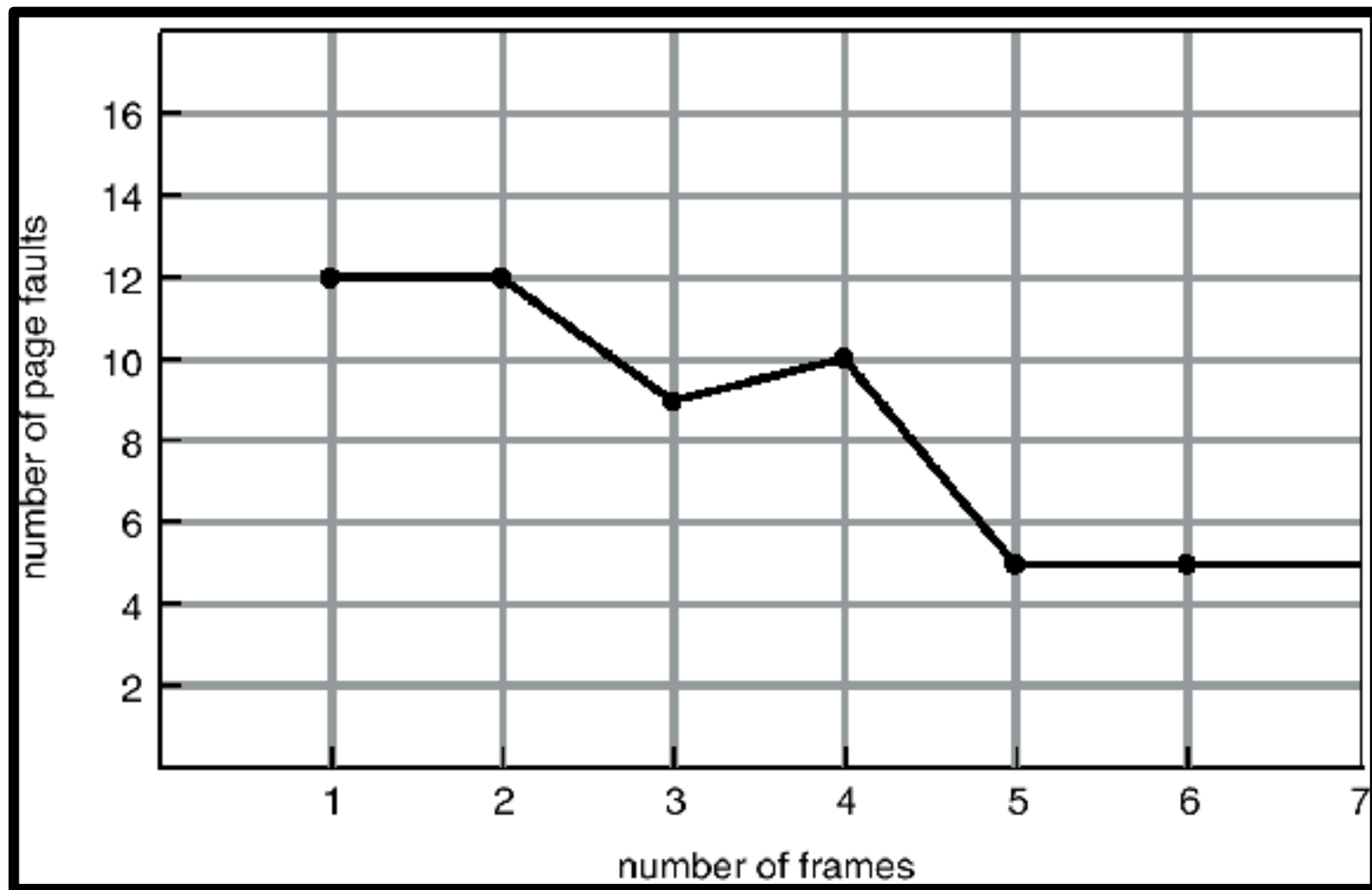
- 4 frames

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- FIFO Replacement – Belady's Anomaly
 - more frames (can) \Rightarrow more page faults (but not generally)

存在Belady 异常的FIFO替换缺页曲线图



Belady异常

- 引用串: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- 3 帧

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

- 4 帧

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

- FIFO 替换算法 – Belady异常
 - 期望: 增加帧数 \Rightarrow 降低缺页率?

LRU页替换

- 内存引用序列: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**

1	5	8 page faults	
2			
3	5	4	
4	3		

- 计数器的实现
 - 每一个页表项 有一个计数器，每次页通过这个表项被访问，把记录拷贝到计数器中
 - 当一个页需要改变是，查看计数器来觉得改变哪一个页**0**

LRU页替换

- 计数器实现

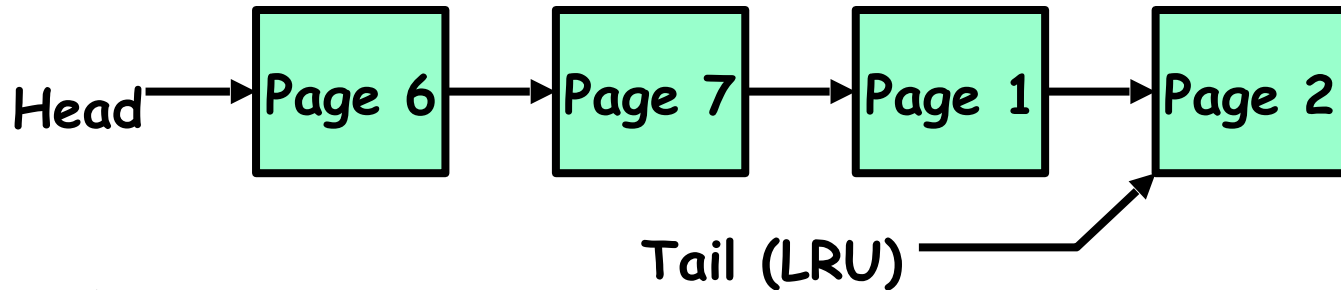
- 每个页表项都关联一个使用时间域
- 需要一个逻辑时钟或计数器，对每次内存引用，计数器都会增加。
- 每次内存引用时，时钟寄存器的内容都会复制到相应页表项的使用时间域内
- 进行页替换时，选择具有最小时间（或者计数器值）的页

- 问题

- 需要搜索页表
- 每次内存访问都需要写页表项的使用时间域
- 上下文切换时需要维护页表
- 需要考虑时钟溢出

LRU页替换

- 栈实现
 - 在一个双链表中保留一个记录页数目的栈



- 最近被访问的页：
 - 移到栈顶
 - 最坏情况下需要改变6个指针
- 无需查找，栈底部即是要找的页
- 理想实现方式：
 - 每次访问记录Timestamp
 - 将页按访问时间排序
 - 实现过于复杂，too expensive

LRU近似页替换

- **Reference bit**

- 每个页都与一位相关联，称为Reference bit, 初始值位0
- 当页被访问时，将该页的Reference bit设为1
- 替换Reference bit为0的第一个页。缺点：时间顺序未知

- 一些通过Reference bit位实现的LRU近似页替换算法

- 时钟算法
 - Arrange physical pages in circle with single clock hand
- 二次机会法
- 增强型二次机会法

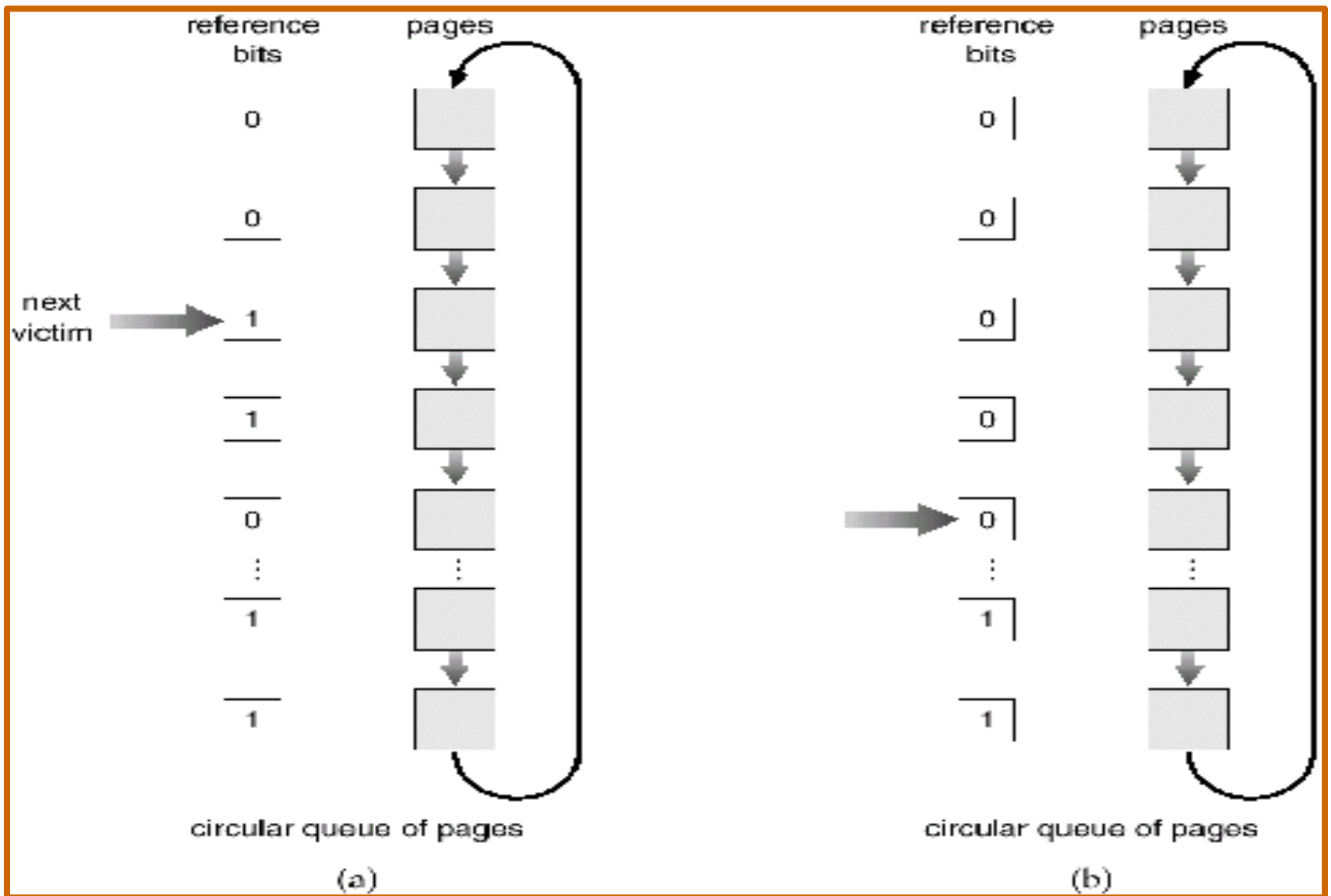
时钟（Clock）算法

- 时钟算法
 - Arrange physical pages in circle with single clock hand
 - Replace **an old** page, not **the oldest** page
- Details:
 - Hardware “reference” bit per physical page:
 - 替换Reference bit为0的第一个页。缺点：顺序未知
 - 如果Reference bit为0, 表明近期末用
 - Nachos hardware sets use bit in the TLB
 - 缺页时:
 - 沿Circle顺序检查
 - 检查reference bit: 1→used recently; clear and leave alone
0→selected candidate for replacement
 - Will always find a page or loop forever?
 - Even if all use bits set, will eventually loop around⇒FIFO

二次机会算法

- 二次机会算法
 - FIFO+ Reference bit
 - 所有帧形成一个队列
 - 每次内存访问，访问页的Reference bit置为1
 - 检查当前帧
 - 如果引用位为1，则置为0，并跳到下一帧
 - 如果引用位为0，则替换该页
 - 假如某个页被频繁访问，那么它就不会被替换出去

二次机会算法



增强型二次机会算法

- 增强型二次机会算法
 - 一个FIFO循环队列
 - 引用位
 - 修改位
- 四种类型（引用位，修改位）：
 - (0, 0)最近没有使用也没有修改过
 - (0, 1) 最近没有使用但曾经被修改过
 - (1, 0)最近使用过，但没有被修改过
 - (1, 1)最近使用过并且修改过
- 当需要替换时，检查页属于哪一类型
- 替换在最低非空类型中所碰到的页
 - 缺点: 需要多次搜索整个循环队列

- LFU：最不经常使用页替换算法
 - 替换具有最小计数的页
 - 定期将计数右移一位，以形成指数衰减的平均使用次数
- MFU：最常使用页替换算法
 - 因为具有最小次数的页可能刚刚被替换进来，并且可能尚未使用，所以替换最常使用的

Example:

- 考虑以下序列:

— A B C A B D A D B C B

Ref: Page:	A	B	C	A	B	D	A	D	B	C	B
1											
2											
3											

— 缺页:

LRU

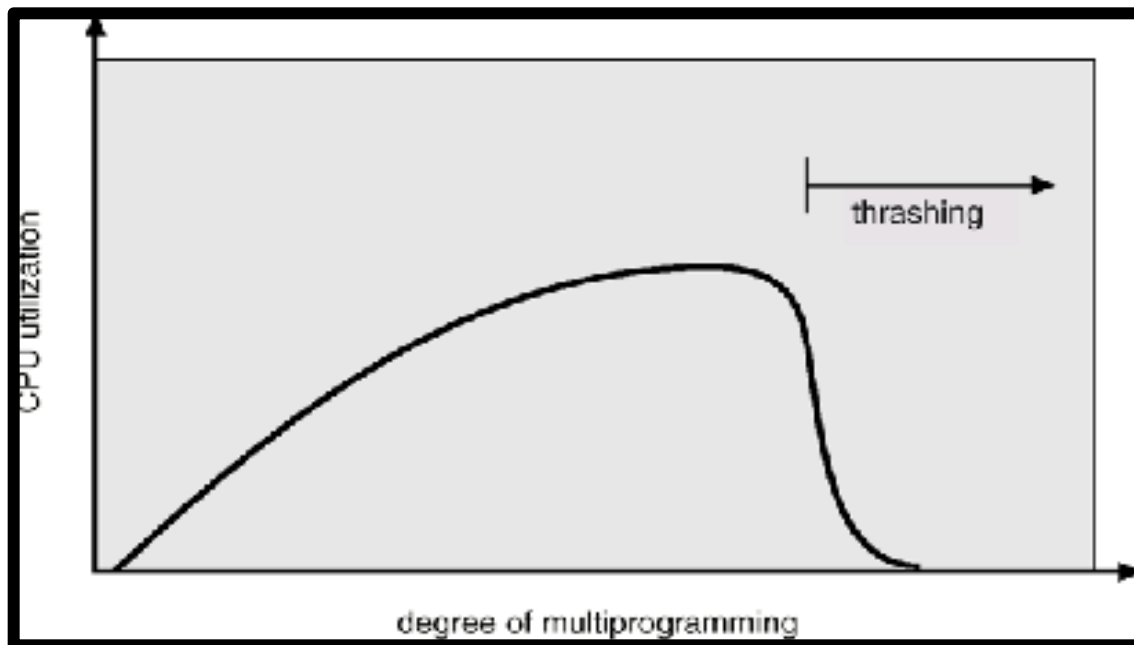
- 考虑以下序列: **A B C D A B C D A B C D**
- LRU (same as FIFO here):

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

内存抖动

- 如果一个进程没有足够的页, 那么缺页率将较高, 这将导致
 - CPU利用率低下
 - 操作系统认为需要增加多道程序设计的道数
- Thrashing (抖动) \Leftarrow 频繁换入换出页

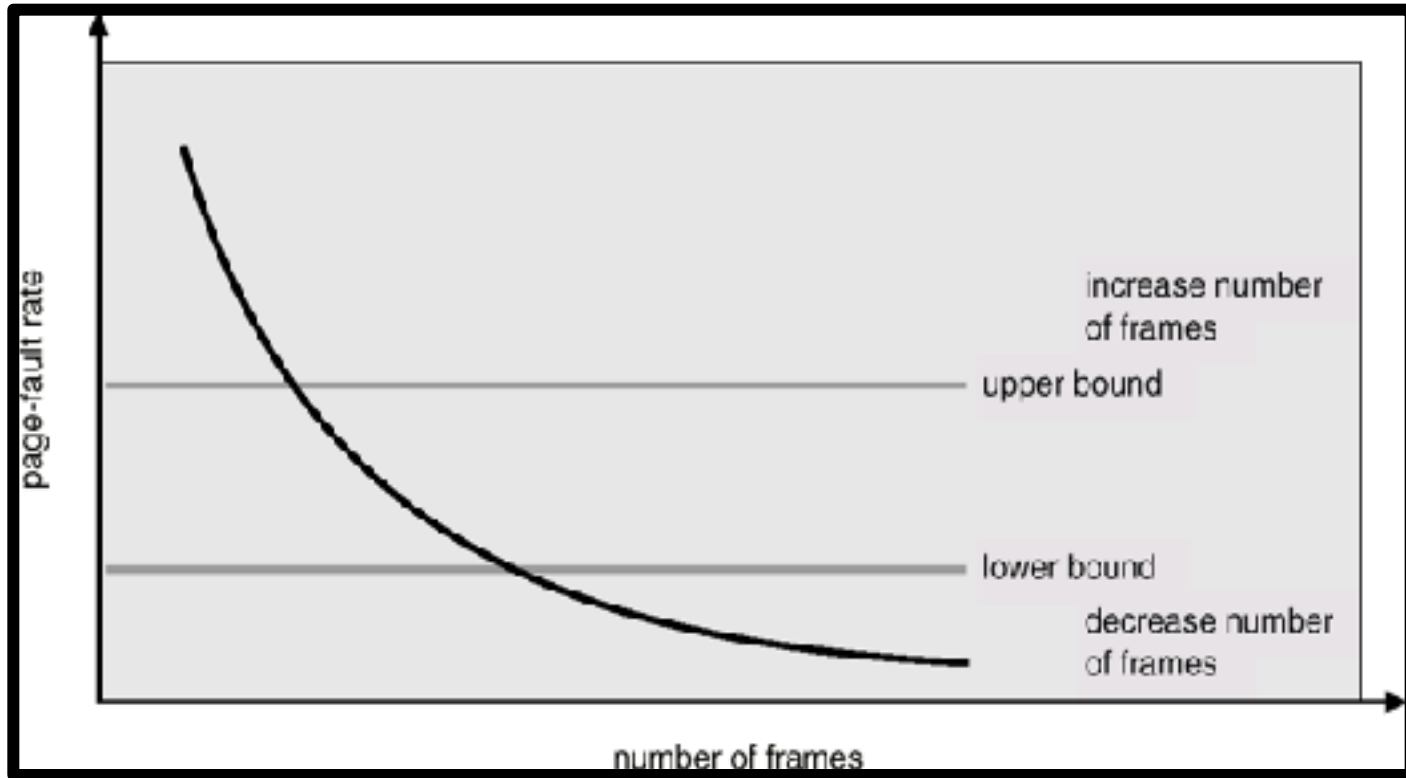
抖动



- 进程工作集合检查进程真正需要多少帧
- 为什么抖动会发生： Σ 局部 > 总的内存

策略：操作系统选择挂起一个进程，换出。

Page fault rate



- 另一种控制抖动的更为直接的方法是设置可接受的缺页率。
 - 如果缺页率太低，回收一些进程的帧。
 - 如果缺页率太高，就分给进程一些帧。

Windows XP

- 请求分页存储管理加 **clustering**: 把所缺页的邻近页也调入内存.
- **working set minimum** 和 **working set maximum**
 - Working set minimum:
 - 进程在内存中的最小页数
 - Working set maximum:
 - 进程在内存中的最大页数
- 当系统空闲内存小于门限值时, 删减**working set**以恢复足够的内存
 - 删去拥有页数超过 working set minimum的那些进程的一部分页数

其它考虑

- 预先调页（Prepaging）：将所需要的页一起调入到内存中
 - 降低缺页率
 - 可能造成浪费
- 页大小的考虑
 - 页表大小：
- 减小页大小- >增加页表大小
 - 碎片：小页更好利用内存
 - I/O开销：寻道和延迟时间远大于传输时间，需要小页
 - 局部：较小页允许每个页更精确的匹配程序局部
- 增加页大小。
 - 不是所有的应用程序都要求一个较大的页大小，会导致碎片的增加
 - 提供多种页大小。

其它考虑

- TLB范围- 从 TLB 可访问的内存量
- $\text{TLB范围} = (\text{TLB 条数}) * (\text{页大小})$
- 理想地, 每进程的工作集存储在 TLB 中。否则有很高的缺页率。
- 页面交换区
 - 专用的磁盘区域用于保存被淘汰的页面内容
 - 采用交换区映射表管理
- 锁定主存页
 - 某些页面在进行 I/O 操作时不能被替换

小结

- 虚拟内存技术允许执行一个进程，它的逻辑地址空间比物理空间大
- 虚拟内存技术提高了多道程序程度，CPU利用率和吞吐量
- 页替换算法
 - FIFO
 - Belady异常
 - 最优
 - LRU（最近最少使用）
 - 计数器
 - 最不经常使用 (LFU)

小结

- 帧分配策略
 - 固定 (i.e. equal share)
 - 按比例(to program size)
 - 优先级
- 抖动
 - 如果进程工作集未获得足够内存，将引起抖动

- 假定一个处理机正在执行两道作业，一道作业以计算为主，另一道以I/O为主，你将怎样为它们分配优先级？为什么？

- 考虑一仅460B的程序的内存访问序列（该序列的下标均从0开始）10，11，104，170，73，309，185，245，246，434，458，364，且页面大小为100B，则
 - （1）写出页面的访问序列。
 - （2）假设内存中仅有200 B可供程序使用且采用FIFO算法，那么共发生多少次缺页中断？
 - （3）如果采用最近最久未使用算法（LRU），则又会发生多少次缺页中断？

- 有一个矩阵为100行，200列。即：
- `varA: arrayll. . 100, 1. . 200] Of integer;`
- 在一个采用LRU淘汰算法的虚拟存储管理系统，系统分给该进程五个页面来存储数据（不包含程序），设每页的大小可以存放200个整数，该程序要对整个数组初始化，数组存放时是按行存放的。试计算下列两个程序各自的缺页次数。（假定所有页都以请求方式调入）
-
- 程序1:
- for i = 1 to 100 do
- for j = 1 to 200 do
- A[i, j] = i*j;
-
- 程序2:
- for j = 1 to 200 do
- for i = 1 to 100 do
- A[i, j] = i*j;