南京大学
NANJING UNIVERSITY

计算机操作系统

第三章 存储管理

南京大学软件学院

第三章 补充内容目录

* 补充1：虚拟存储器的概念(补充局部性特征)
* 补充2：伙伴系统
* 补充3：分页和分段的寻址计算
* 补充4：多级页表与反置页表
* 补充5：页的大小设计
* 补充6：页面替换算法
* 补充7：TLB快表，页表，缺页(讨论)

补充1：虚拟存储器的概念

局部性特征：

分页下的运行情况(补充)

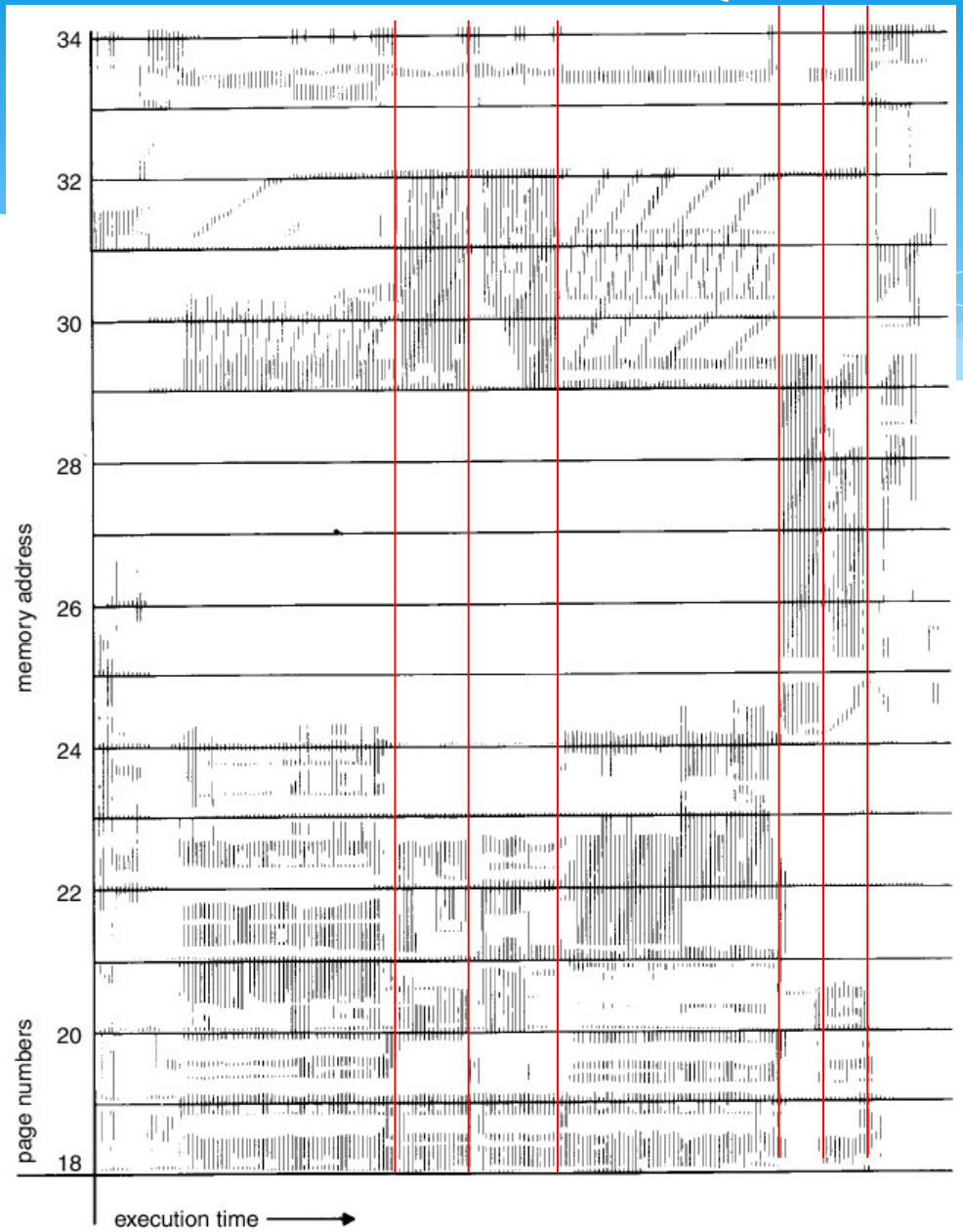
抖动

* 如果一块正好将要被用到之前扔出，操作系统有不得不很快把它取回来，太多的这类操作会导致一种称为系统抖动的情况
* 在处理缺页中断期间，处理器的大部分时间都用于交换块，而不是用户进程的执行指令

程序局部性原理(1)

* 指程序在执行过程中的一个较短时间内，所执行的**指令地址或操作数地址分别局限于一定的存储区域中**。又可细分时间局部性和空间局部性
* 早在1968年P. Denning研究程序执行时的局部性原理，对此进行研究的还有Knuth(分析一组学生的Fortran程序)、Tanenbaum (分析操作系统的过程)、Huck(分析通用科学计算程序)，**发现程序和数据的访问都有聚集成群的倾向**
* 某存储单元被使用，其相邻存储单元很快也被使用(称空间局部性spatial locality)，
* 或者最近访问过的程序代码和数据，很快又被访问(称时间局部性temporal locality)

分页下的运行情况


34
32
30
28
26
24
22
20
18
memory address
page numbers
execution time

程序局部性原理(2)

* (1) 程序中只有少量分支和过程调用，存在很多顺序执行的指令
* (2) 程序含有若干循环结构，由少量代码组成，而被多次执行
* (3) 过程调用的深度限制在小范围内，因而，指令引用通常被局限在少量过程中
* (4) 涉及数组、记录之类的数据结构，对它们的连续引用是对位置相邻的数据项的操作
* (5) 程序中有些部分彼此互斥，不是每次运行时都用到

程序局部性原理(3)

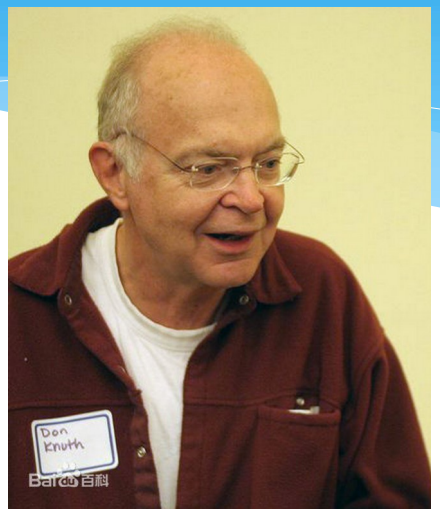
* 经验与分析表明，程序具有局部性，进程执行时没有必要把全部信息调入主存，只需装入一部分的假设是合理的，部分装入的情况下，只要调度得当，不仅可正确运行，而且能在主存中放置更多进程，充分利用处理器和存储空间

虚拟内存的技术需要

* 必须有对所采用的分页或分段方案的硬件支持
* 操作系统必须有管理页或者段在主存和辅助存储器之间移动的软件。

补充2：伙伴系统

Donald Ervin Knuth



Donald Ervin Knuth (1938~), 斯坦福大学教授

1963年获得加州理工学院博士学位, 高德纳是算法和程序设计技术的先驱者，计算机排版系统TEX和METAFONT的发明者, 1974年获得图灵奖。

经典著作: The Art of Computer Programming,

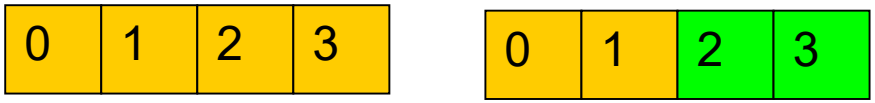
Volume 1: Fundamental Algorithms, first edition, 1968,

Volume 2: Seminumerical Algorithms, first edition, 1969,

Volume 3: Sorting and Searching, first edition, 1973

补充2：伙伴系统

* 伙伴系统(Knuth，1973)，又称buddy算法，是一种固定分区和可变分区折中的主存管理算法，基本原理是：任何尺寸为2^i的空闲块都可被分为两个尺寸为2^{i-1}的空闲块，这两个空闲块称作伙伴，它们可以被合并成尺寸为2^i的原先空闲块。
* 伙伴通过对大块的物理主存划分而获得
  * 假如从第0个页面开始到第3个页面结束的主存



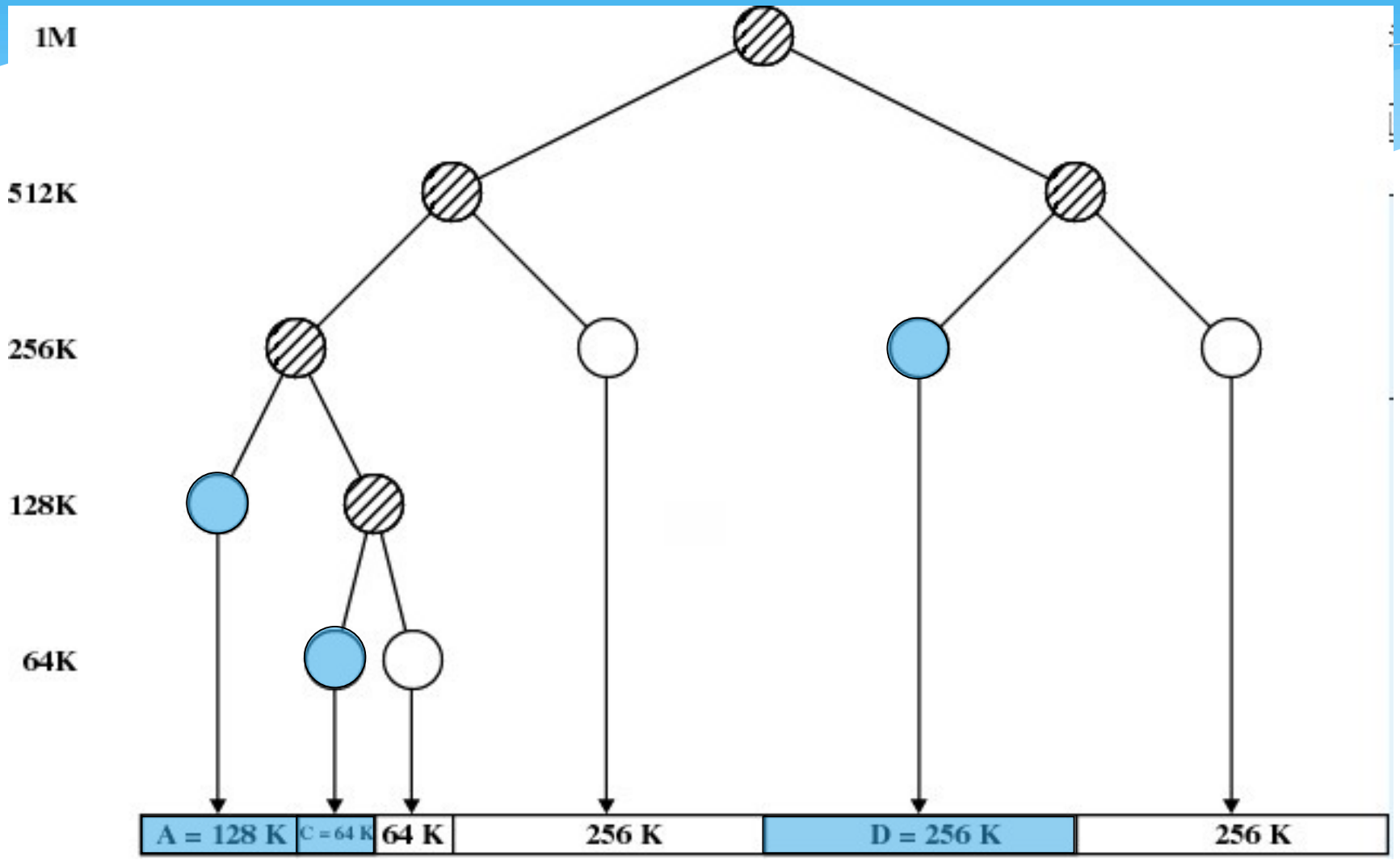
  * 每次都对半划分，那么第一次划分获得大小为2页的伙伴，如0、1和2、3
  * 进一步划分，可以获得大小为1页的伙伴，例如0和1，2和3

南京大学 NANJING UNIVERSITY

3.2.3 伙伴系统

1-Mbyte block	1M					
Request 100K	A = 128K	128K	256K	512K		
Request 240K	A = 128K	128K	B = 256K	512K		
Request 64K	A = 128K	C = 64K	64K	B = 256K	512K	
Request 256K	A = 128K	C = 64K	64K	B = 256K	D = 256K	256K
Release B	A = 128K	C = 64K	64K	256K	D = 256K	256K
Release A	128K	C = 64K	64K	256K	D = 256K	256K
Request 75K	E = 128K	C = 64K	64K	256K	D = 256K	256K
Release C	E = 128K	128K	256K	D = 256K	256K	
Release E	512K			D = 256K	256K	
Release D	1M					

伙伴系统



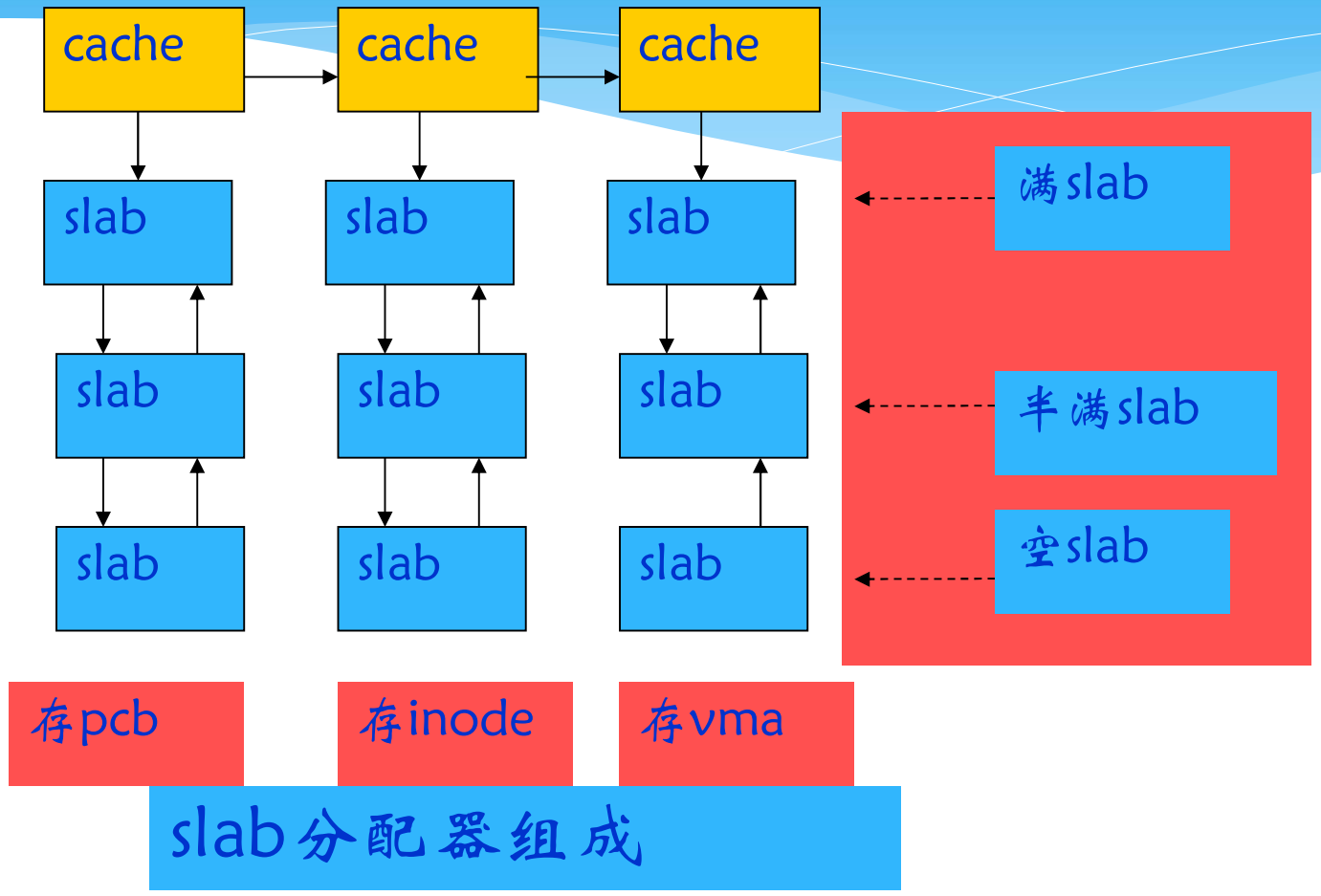
Linux伙伴系统

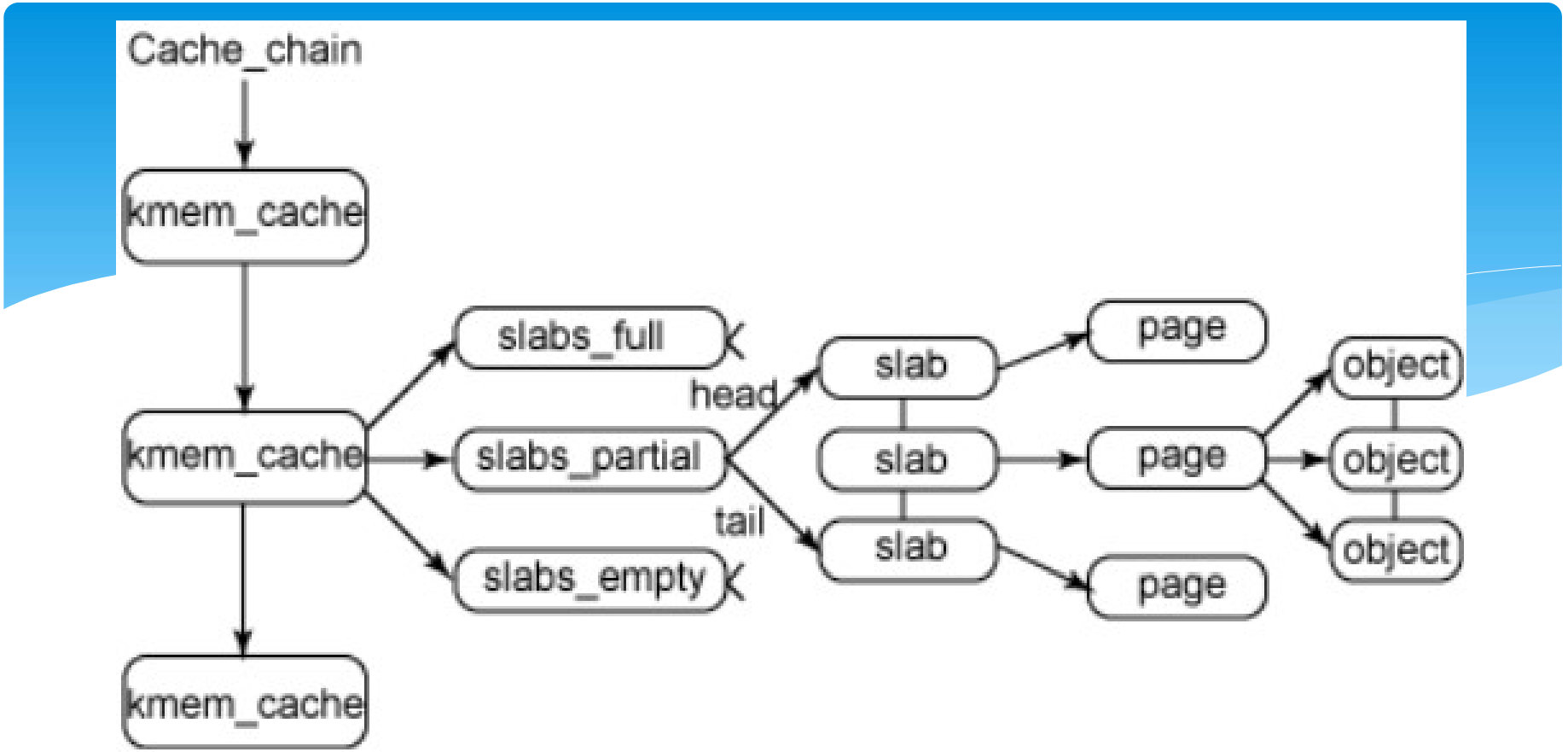
1) 以page结构为数组元素的mem_map[ ]数组
2) 以free_area_struct结构为数组元素的free_area数组
3) 位图数组(bitmap)

Linux基于伙伴的slab分配器(1)

* 为什么要使用slab分配器?
  * 伙伴系统以页框为基本分配单位，内核在很多情况下，需要的主存量远远小于页框大小，如inode、vma、task_struct等，为了更经济地使用内核主存资源，引入SunOS操作系统中首创的基于伙伴系统的slab分配器，其基本思想是：为经常使用的小对象建立缓存，小对象的申请与释放都通过slab分配器来管理，仅当缓存不够用时才向伙伴系统申请更多空间。//页内可以按2的幂次拆分。
  * 优点：充分利用主存，减少内部碎片，对象管理局部化，尽可能少地与伙伴系统打交道，从而提高效率。
* slab的结构
* slab的操作
* Slab举例

Linux基于伙伴的slab分配器(2)





M. Tim Jones, Linux slab 分配器剖析
https://www.ibm.com/developerworks/cn/linux/l-linux-slab-allocator/

Linux还提供十三种通用缓存，其存储对象的大小分别为32B、64B、128B、256B、512B、1KB、2KB、4KB、8KB、16KB、32KB、64KB和128KB，这些缓存用来满足特定对象之外的普通主存需求，单位的大小呈2的幂数增长，保证内部碎片率不超过50%。

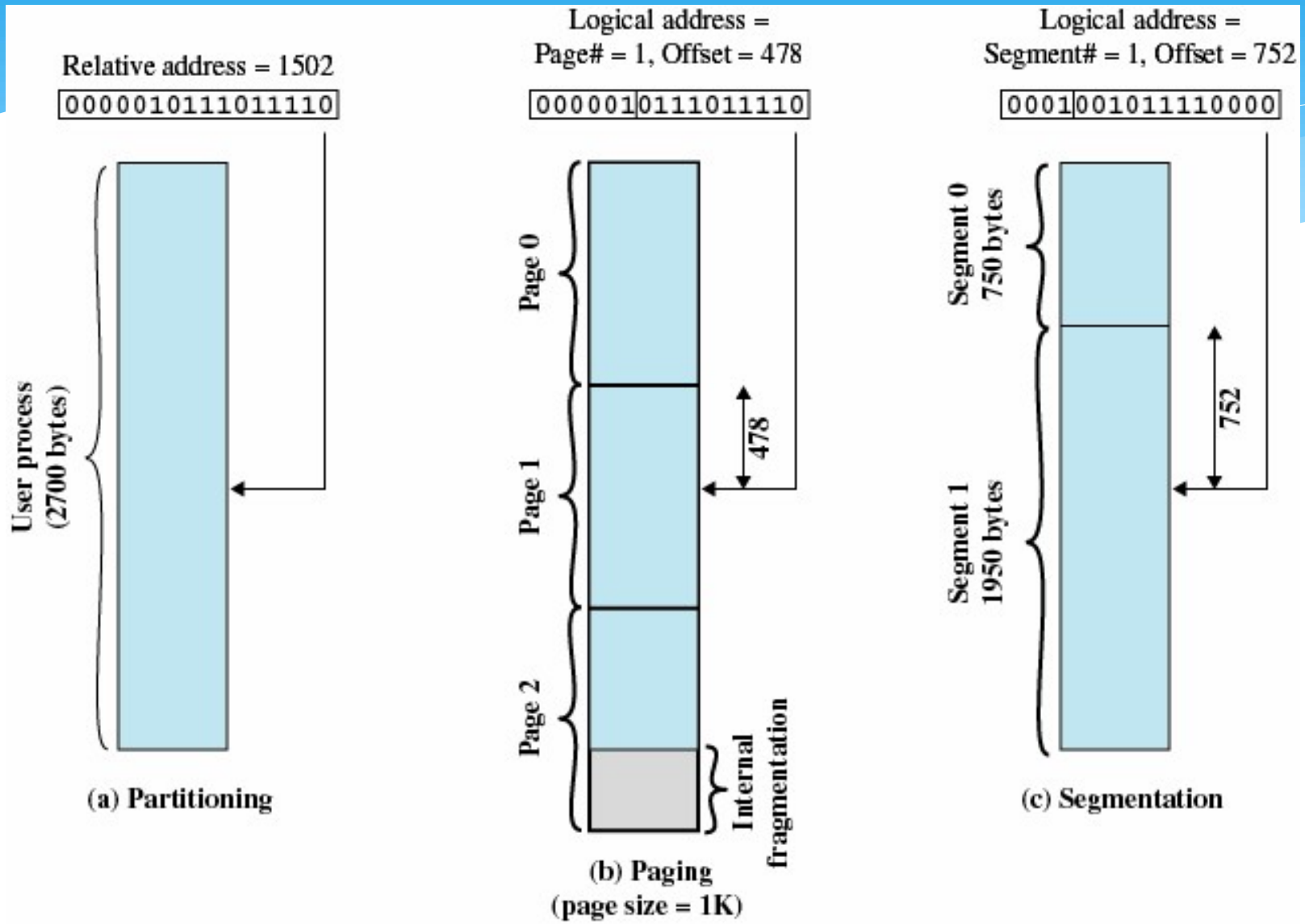
例子task_struct slab

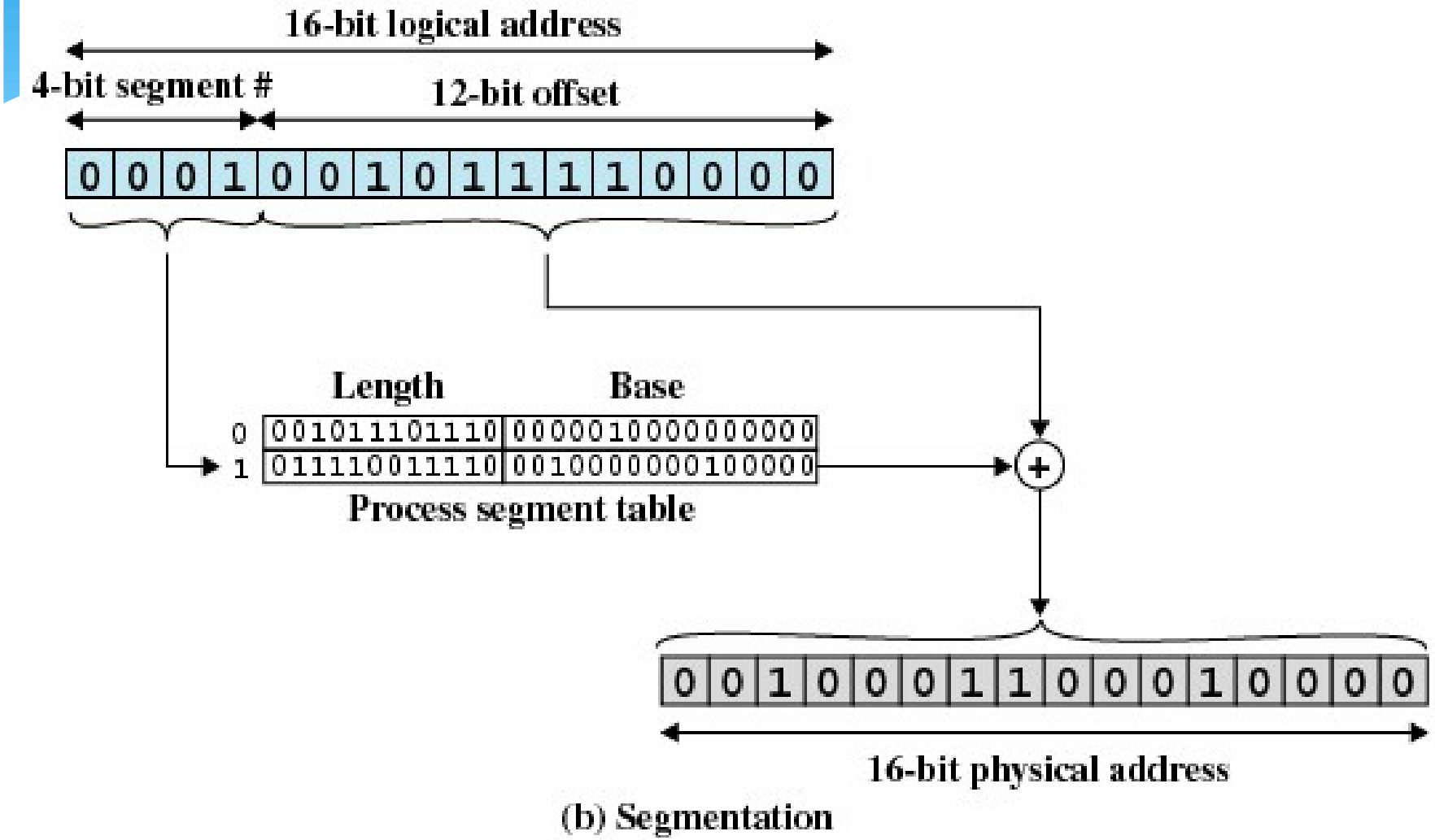
* 内核用全局变量存放指向task_struct slab的指针：kmem_struct_t *task_struct_cachep;初始化时，在fork_init()中调用kmem_cache_create()函数创建高速缓存，存放类型为task_struct的对象。
* 每当进程调用fork()时，调用内核函数do_fork()，它再使用kmem_cache_alloc()函数在对应slab中建立一个task_struct对象。
* 进程执行结束后，task_struct对象被释放，返还给task_struct_cachep slab。

slab分配器主要操作

* 1)kmem_cache_create()函数：创建专用cache，规定对象的大小和slab的构成，并加入cache管理队列；
* 2)kmem_cache_alloc()与kmem_cache_free()函数：分别用于分配和释放一个拥有专用slab队列的对象；
* 3)kmem_cache_grow()与kmem_cache_reap()函数：
kmem_cache_grow()它向伙伴系统申请向cache增加一个slab；
kmem_cache_reap()用于定时回收空闲slab；
* 4)kmem_cache_destroy( )与kmem_cache_shrink( )：用于cache的销毁和收缩；
* 5)kmalloc()与kfree()函数：用来从通用的缓冲区队列中申请和释放空间；
* 6) kmem_getpages( )与kmem_freepages( )函数：slab与页框级分配器的接口，当slab分配器要创建新的slab或cache时，通过kmem_getpages( )向内核提供的伙伴算法来获得一组连续页框。如果释放分配给slab分配器的页框，则调用kmem_freepages( )函数。

补充3：分页和分段的寻址计算(例)


Relative address = 1502
0000010111011110
User process
(2700 bytes)
(a) Partitioning
Logical address =
Page# = 1, Offset = 478
000001 0111011110
Page 0
Page 1
Page 2
478
Internal
fragmentation
(b) Paging
(page size = 1K)
Logical address =
Segment# = 1, Offset = 752
0001 001011110000
Segment 0
750 bytes
Segment 1
1950 bytes
752
(c) Segmentation

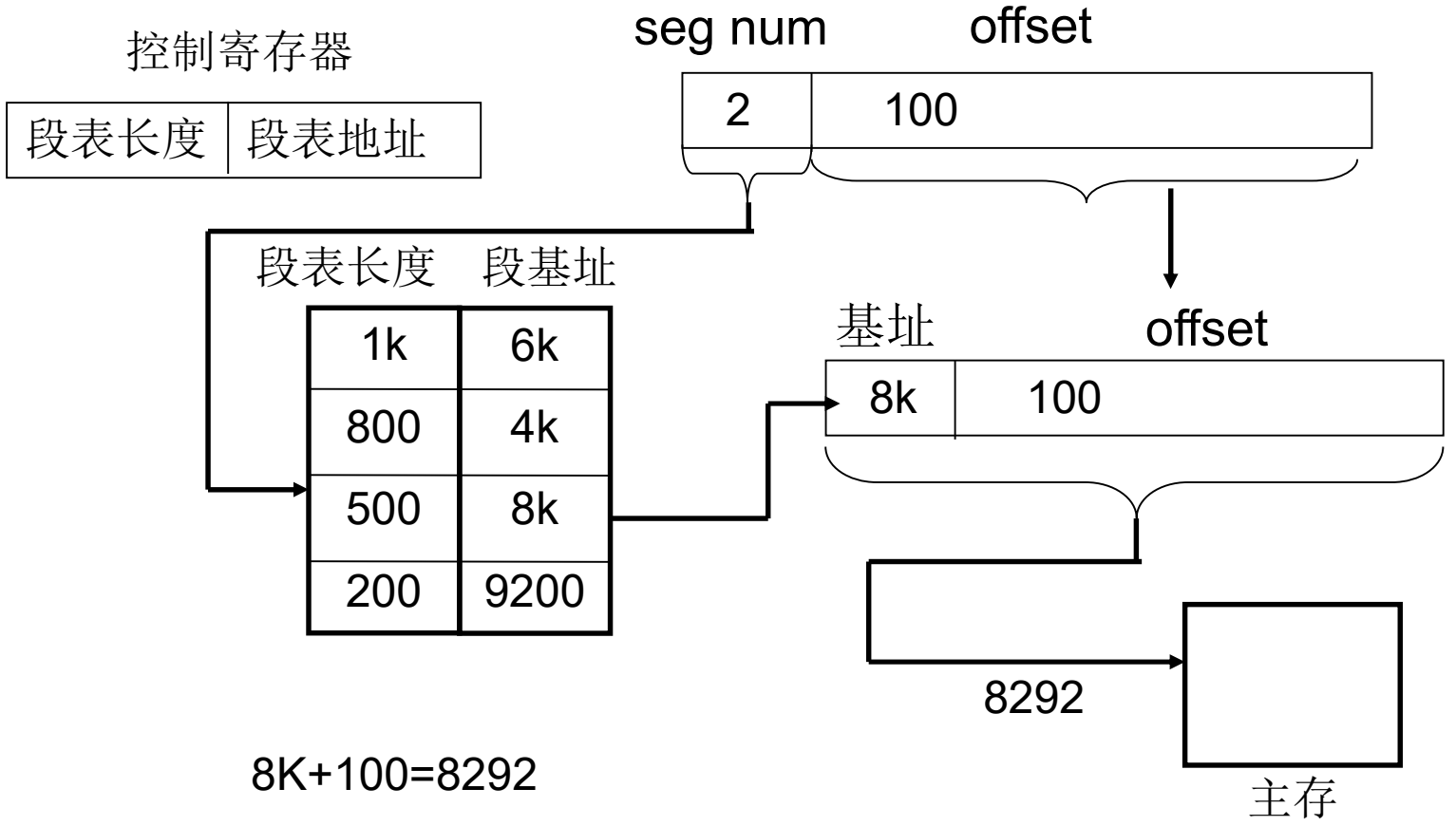


(b) Segmentation

南京大学 NANJING UNIVERSITY

分段

* 采用分段法，
* 某个分段的逻辑地址的段号为2，段内偏移量为100，计算它的物理地址



分段和分页的比较(1)

* 分段是信息的逻辑单位，由源程序的逻辑结构所决定，用户可见，
* 段长可根据用户需要来规定，段起始地址可从任何主存地址开始。
* 分段方式中，源程序(段号，段内位移)经连结装配后地址仍保持二维结构。

分段和分页的比较(2)

* 分页是信息的物理单位，与源程序的逻辑结构无关，用户不可见，
* 页长由系统确定，页面只能以页大小的整倍数地址开始
* 分页方式中，源程序(页号，页内位移)经连结装配后地址变成了一维结构

南京大学
NANJING UNIVERSITY

分页：逻辑地址→物理地址

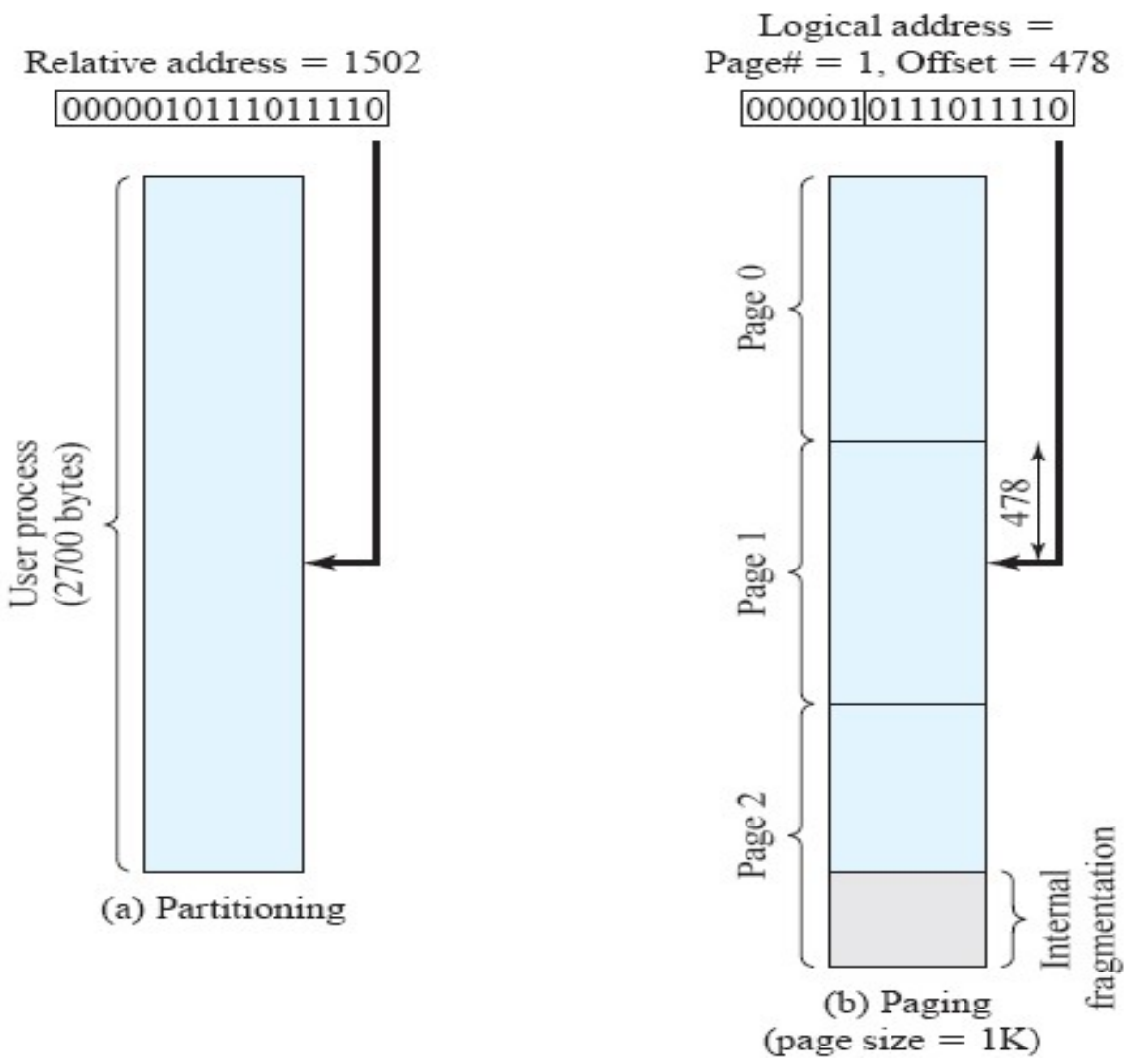
* 逻辑地址

Page Num	offset

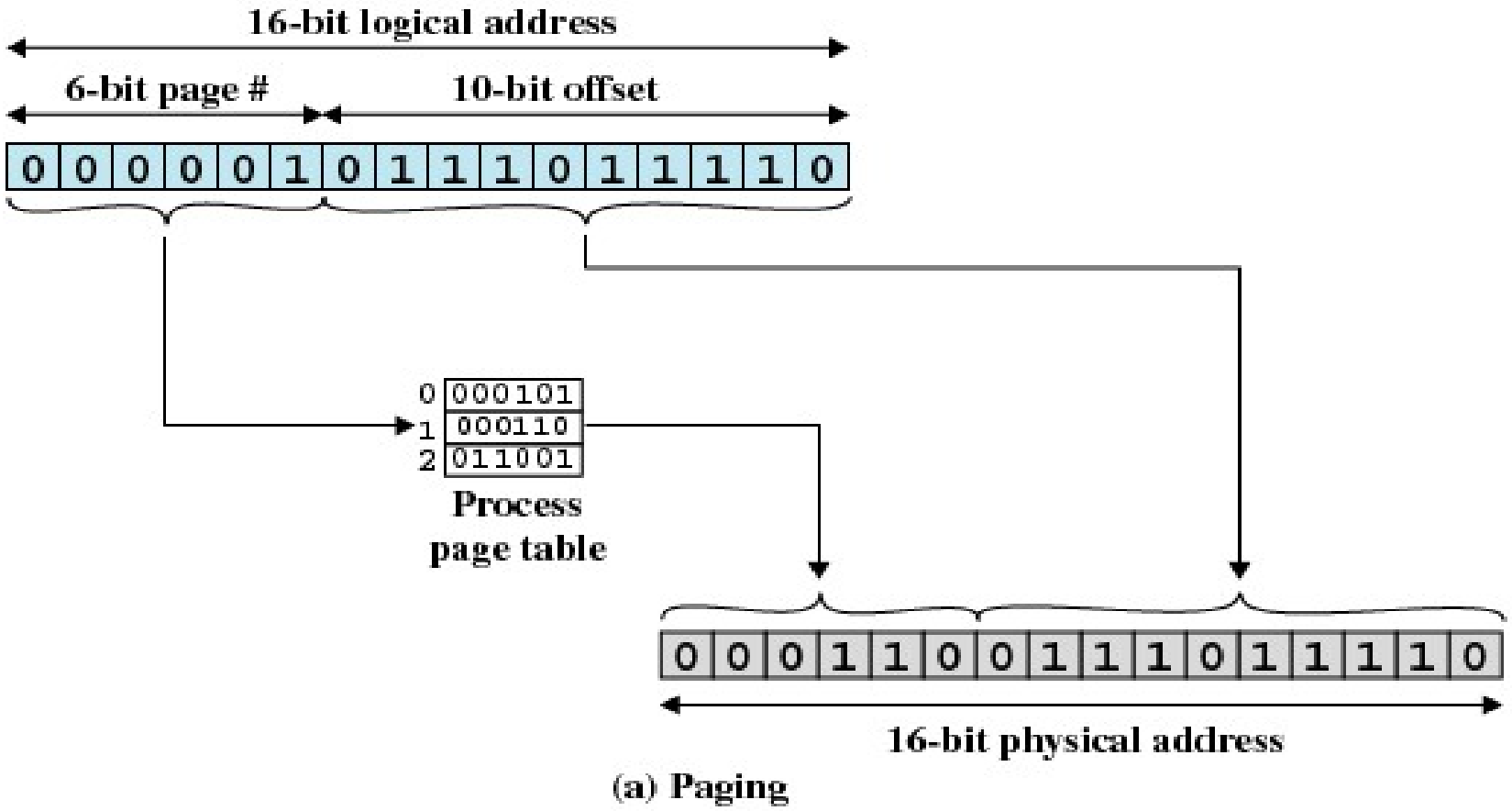
* 物理地址

Frame Num	offset

分页：逻辑地址→物理地址


Relative address = 1502
0000010111011110
User process
(2700 bytes)
(a) Partitioning
Logical address =
Page# = 1, Offset = 478
000001 0111011110
Page 0
Page 1
Page 2
478
Internal
fragmentation
(b) Paging
(page size = 1K)

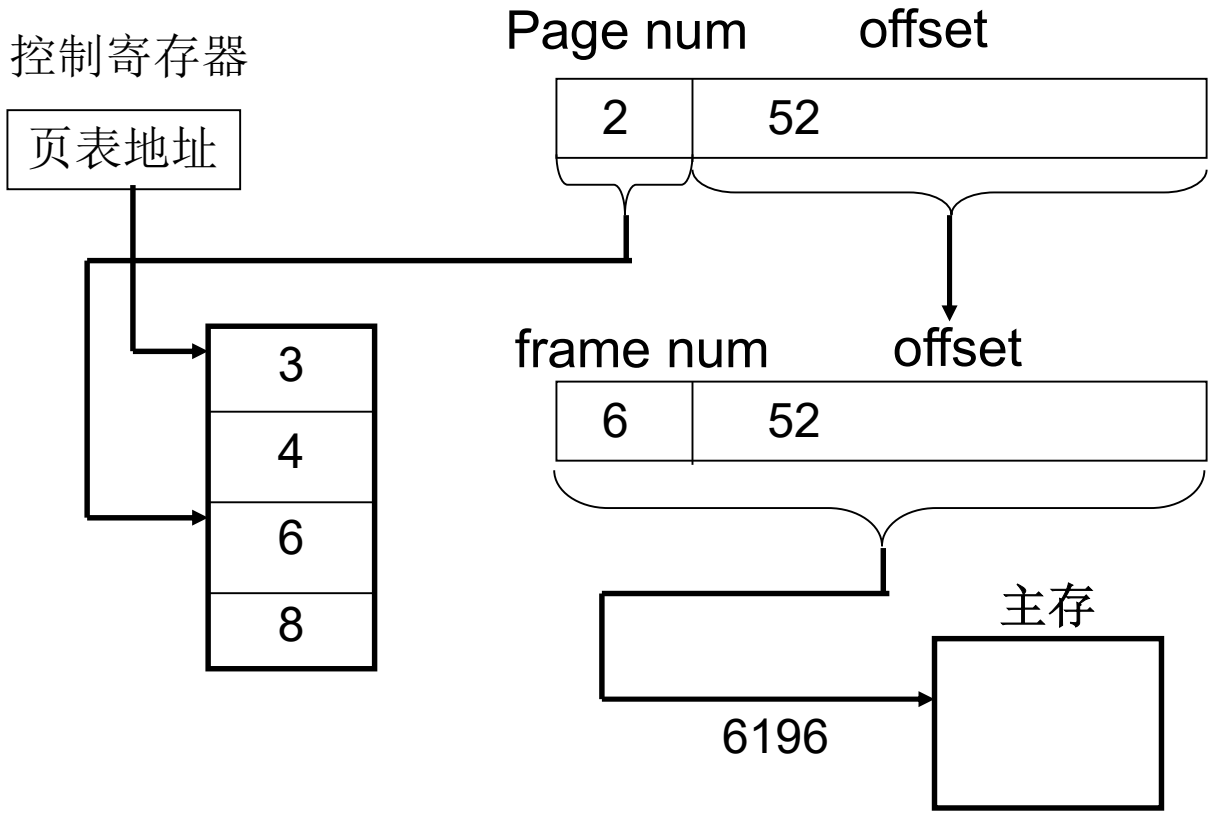
分页：逻辑地址→物理地址



(a) Paging

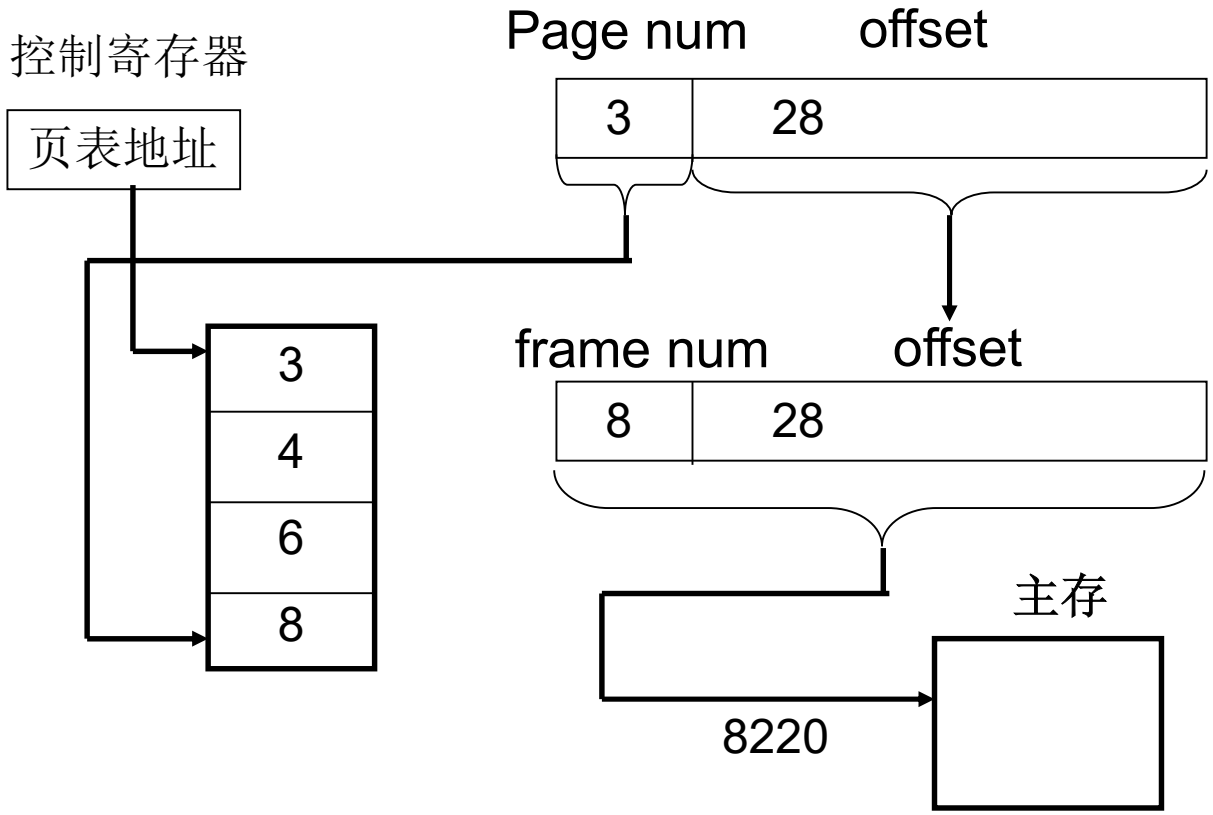
分页地址转换(例)

* 页面与页框的大小为1024字节，指令 MOV 2100, 3100
* 求MOV指令中两个操作数的物理地址
* 2100 = 1024×2 + 52



分页地址转换(例)

* 页面与页框的大小为1024字节，指令 MOV 2100, 3100
* 求MOV指令中两个操作数的物理地址
* 3100？



补充4：多级页表与反置页表

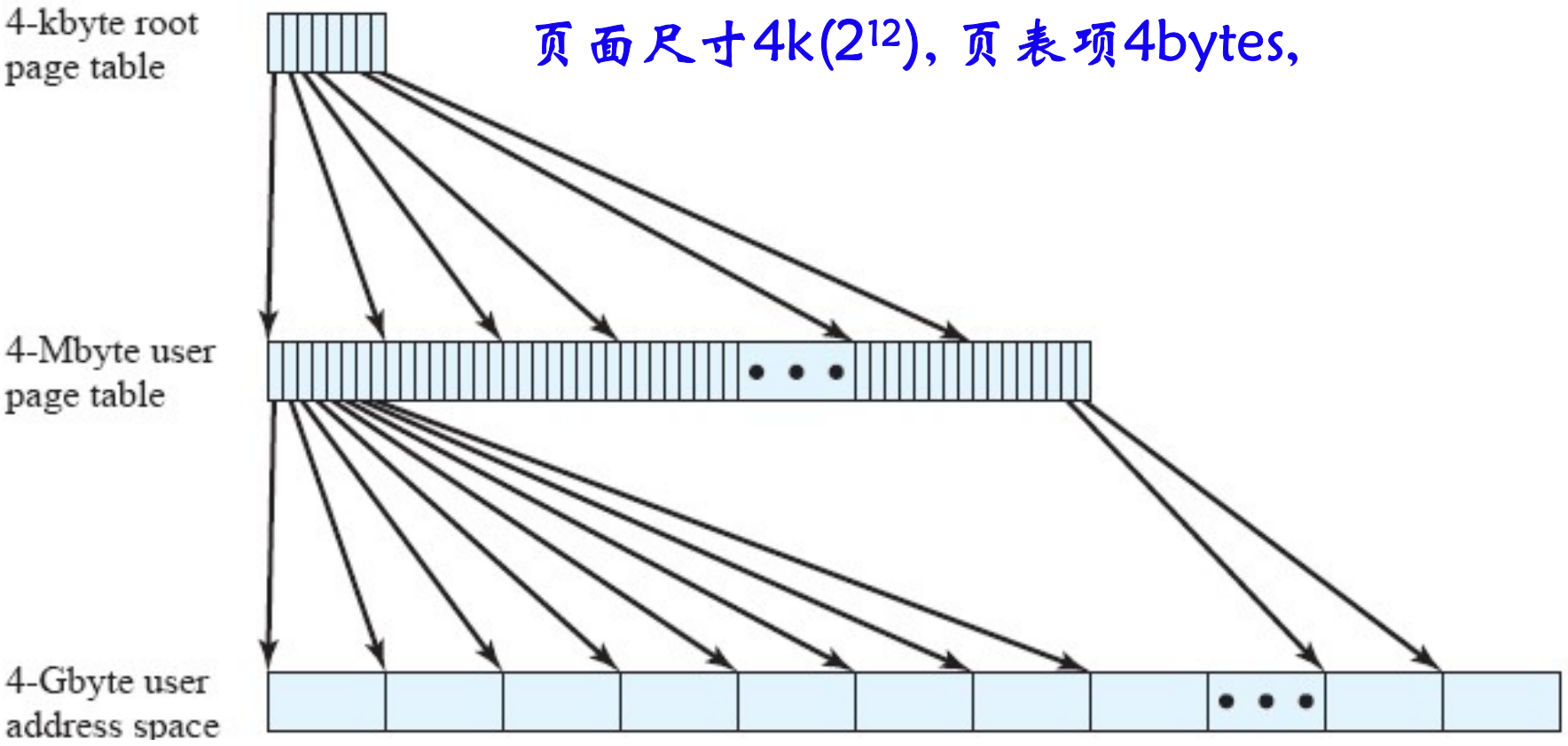
多级页表

* 多级页表的概念
  * 现代计算机普遍支持2^{32}～2^{64}容量的逻辑地址空间，采用分页存储管理时，页表相当大，以Windows为例，其运行的Intel x86平台具有32位地址，规定页面4KB(2^{12})时，那么，4GB(2^{32})的逻辑地址空间由1兆(2^{20})个页组成，若每个页表项占用4个字节，则需要占用4MB(2^{22})连续主存空间存放页表。系统中有许多进程，因此页表存储开销很大。
* 多级页表的具体做法
* 逻辑地址结构
* 逻辑地址到物理地址转换过程

多级页表的概念

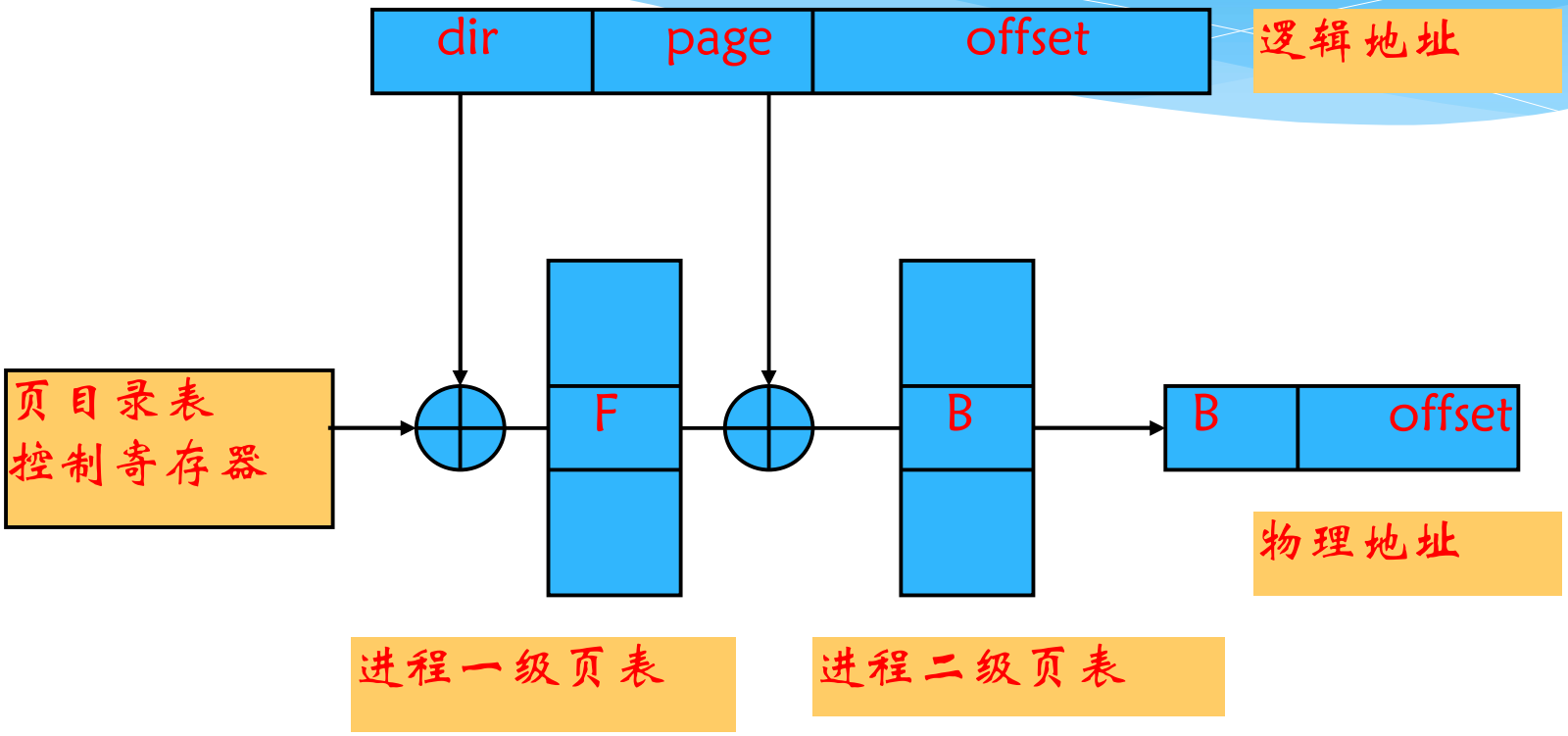
* 系统为每个进程建一张页目录表，它的每个表项对应一个页表页，而页表页的每个表项给出了页面和页框的对应关系，页目录表是一级页表，页表页是二级页表。
* 逻辑地址结构有三部分组成：页目录、页表页和位移

两级页表(32位地址)

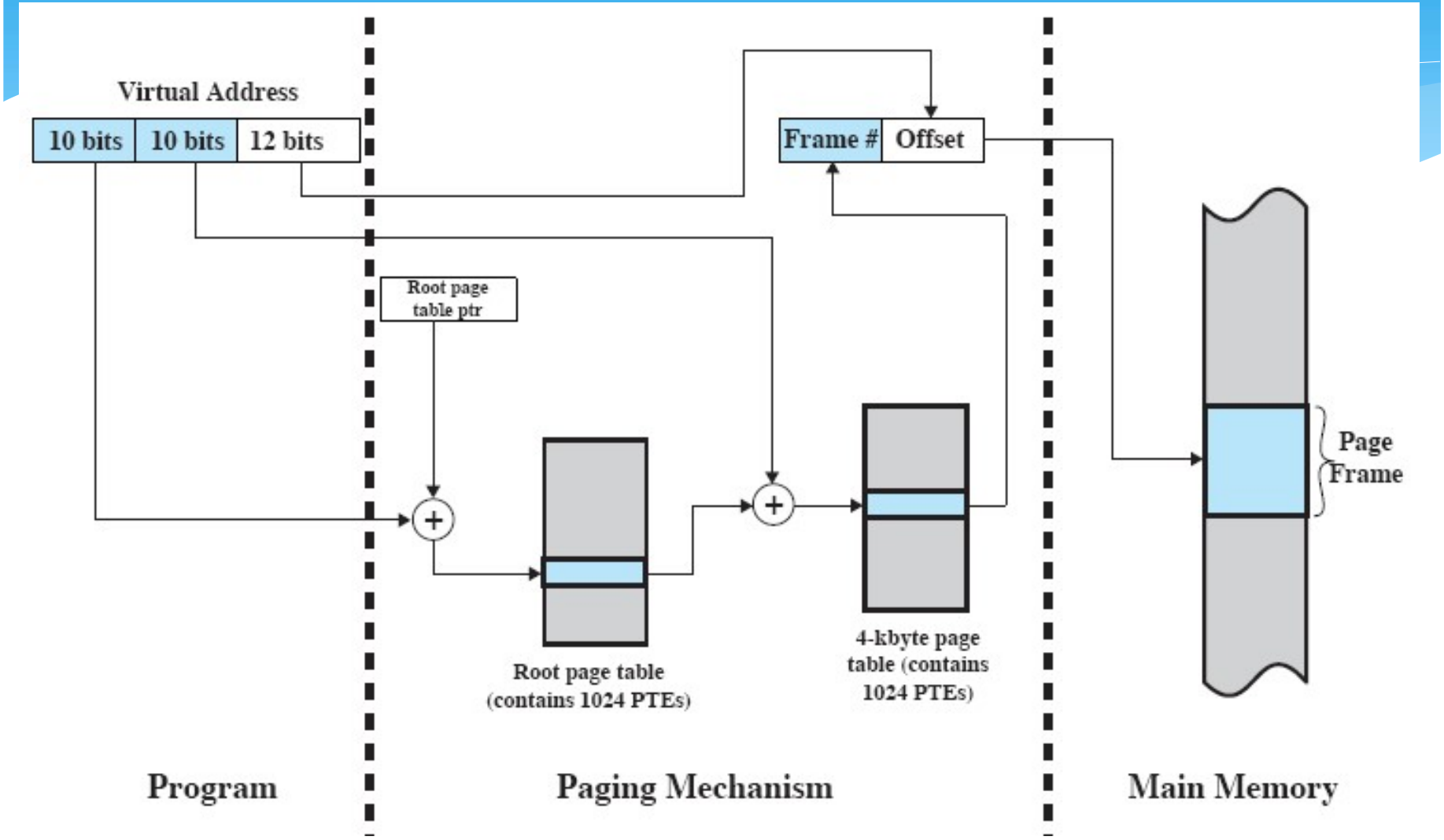


页面尺寸4k(2^{12})，页表项4bytes，

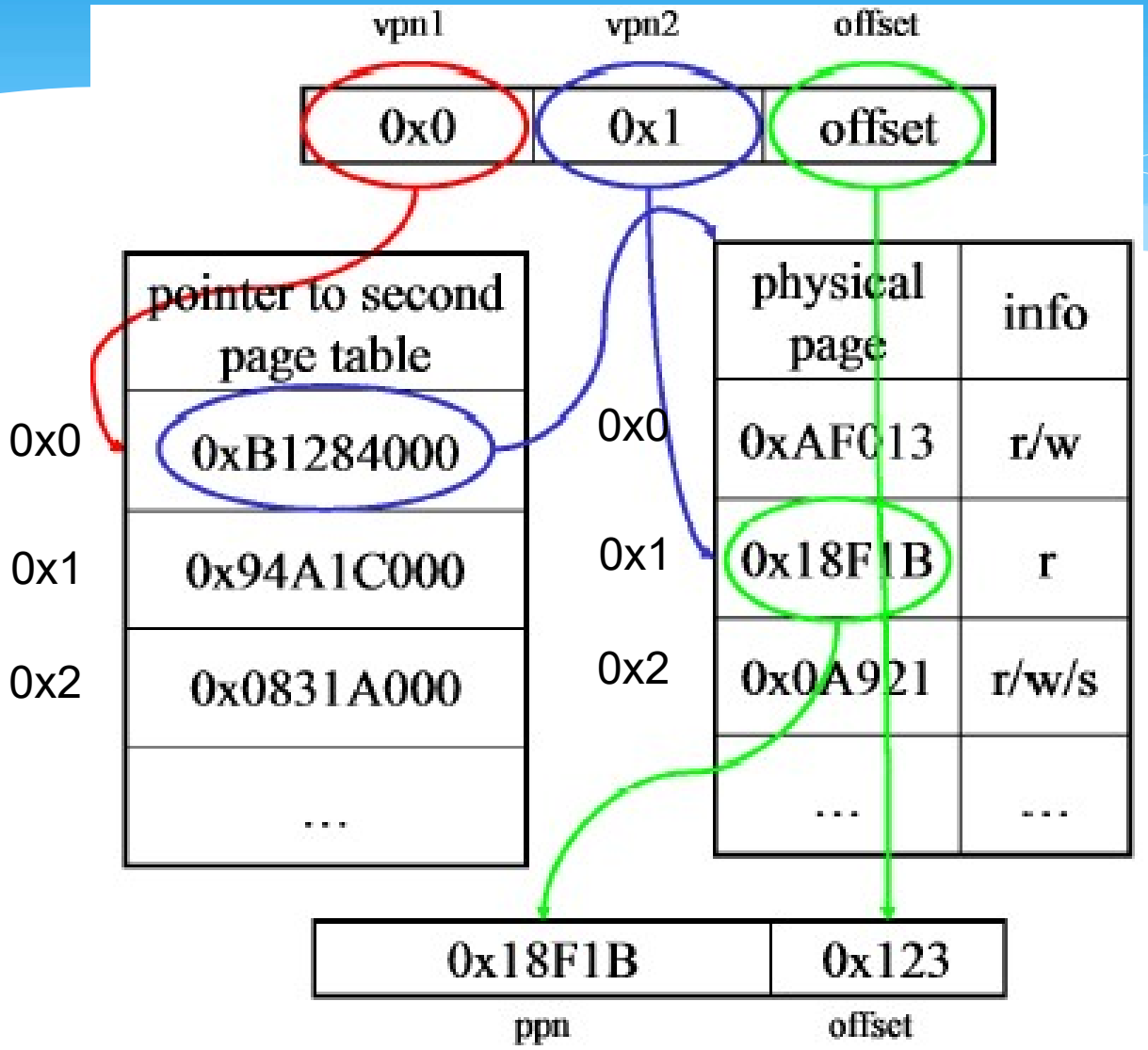
多级页表地址转换过程


dir
page
offset
逻辑地址
页目录表
控制寄存器
F
B
B
offset
物理地址
进程一级页表
进程二级页表

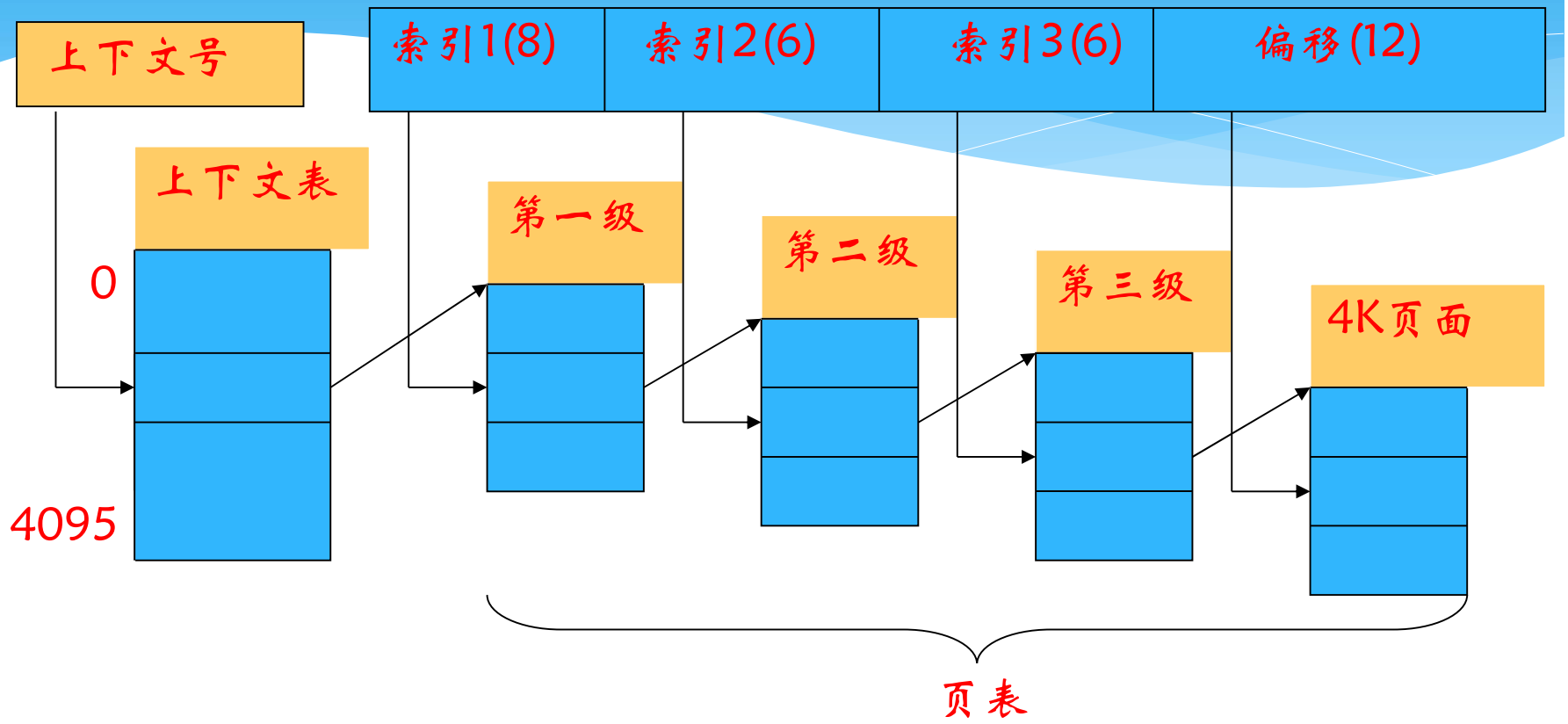
Two-Level Scheme for 32-bit Address



二级页表


vpn1
vpn2
offset
0x0
0x1
offset
pointer to second page table
0x0
0xB1284000
0x1
0x94A1C000
0x2
0x0831A000
...
0x0
0x1
0x2
physical page
info
0xAF013
r/w
0x18F1B
r
0x0A921
r/w/s
...
...
0x18F1B
0x123
ppn
offset

SUN SPARC计算机三级分页结构



问题：增加了寻址时间，在计算机系统中时间与空间总是存在一些矛盾，因此经常会采取折衷的方案，以时间换空间，或者以空间换取时间。

多级页表结构的本质

* 多级不连续导致多级索引。
* 以二级页表为例，用户程序的页面不连续存放，要有页面地址索引，该索引是进程页表；进程页表又是不连续存放的多个页表页，故页表页也要页表页地址索引，该索引就是页目录。
* 页目录项是页表页的索引，而页表页项是进程程序的页面索引。

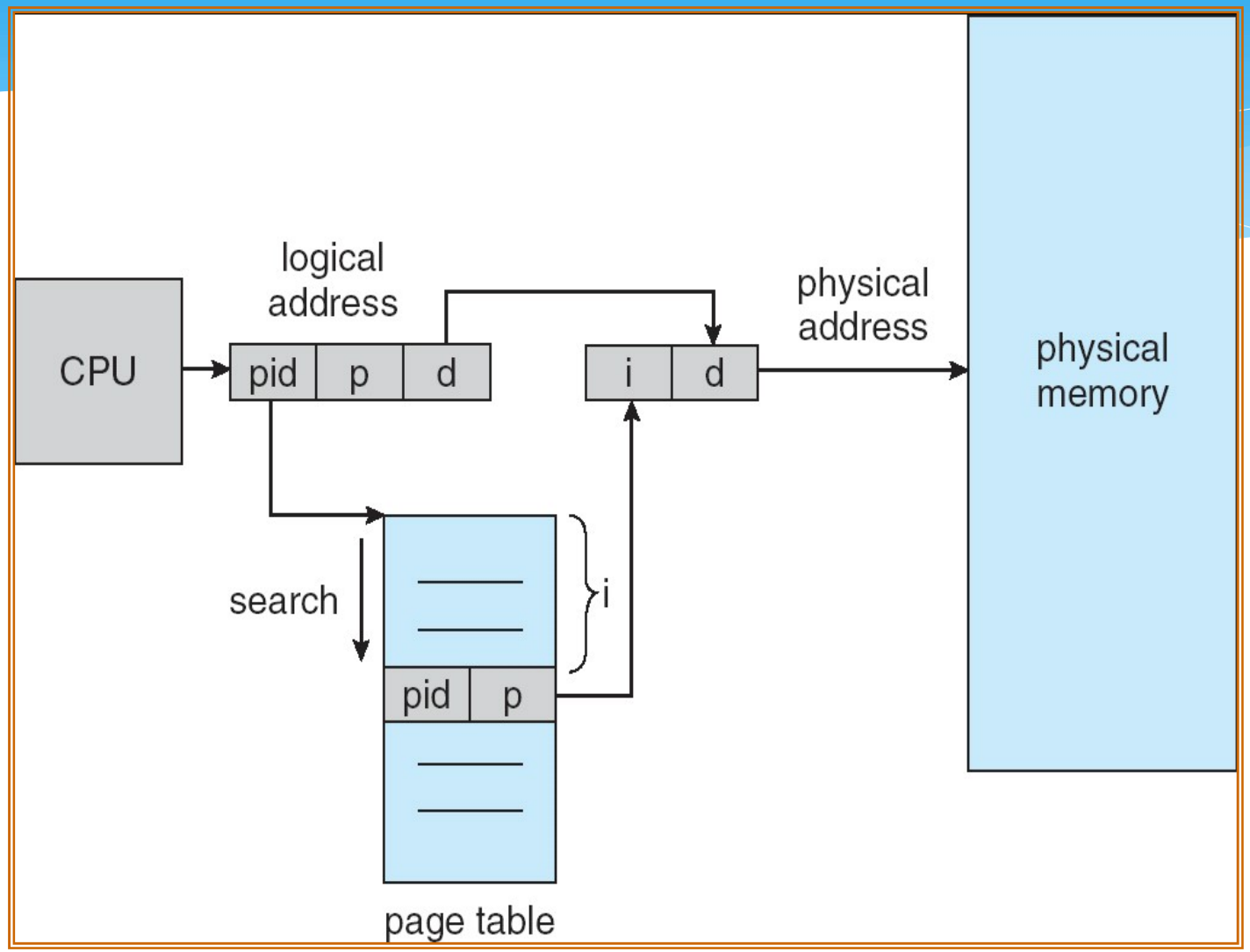
反置页表

* 页表设计的一个重要缺陷是页表的大小与虚拟地址空间的大小成正比
* 在反向页表方法中，虚拟地址的页号部分使用一个简单散列函数映射到哈希表中。哈希表包含一个指向反向表的指针，而反向表中含有页表项。
* 通过这个结构，哈希表和反向表中只有一项对应于一个实存页(面向实存)，而不是虚拟页(面向虚存)。
* 因此，不论由多少进程、支持多少虚拟页，页表都只需要实存中的一个固定部分。
* PowerPC, UltraSPARC, and IA-64

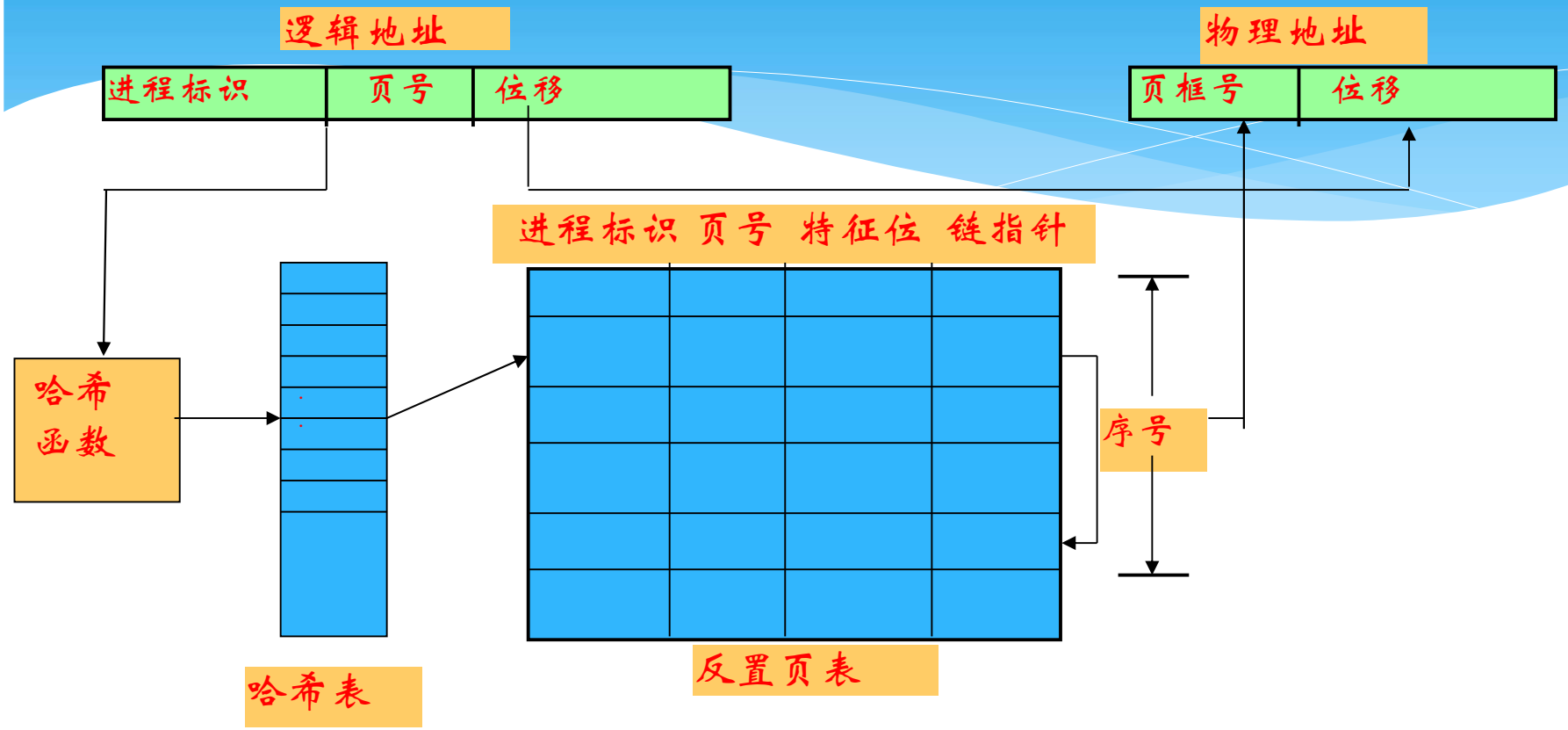
反置页表

* 页号：虚拟地址页号部分。
* 进程标志符：使用该页的进程。页号和进程标志符结合起来标志一个特定的进程的虚拟地址空间的一页。
* 控制位：该域包含一些标记，比如有效、访问和修改，以及保护和锁定的信息。
* 链指针：如果某个项没有链项，则该域为空(允许用一个单独的位来表示)。

反置页表的结构


CPU
logical address
pid
p
d
i
d
physical address
physical memory
search
i
pid
p
page table

反置页表



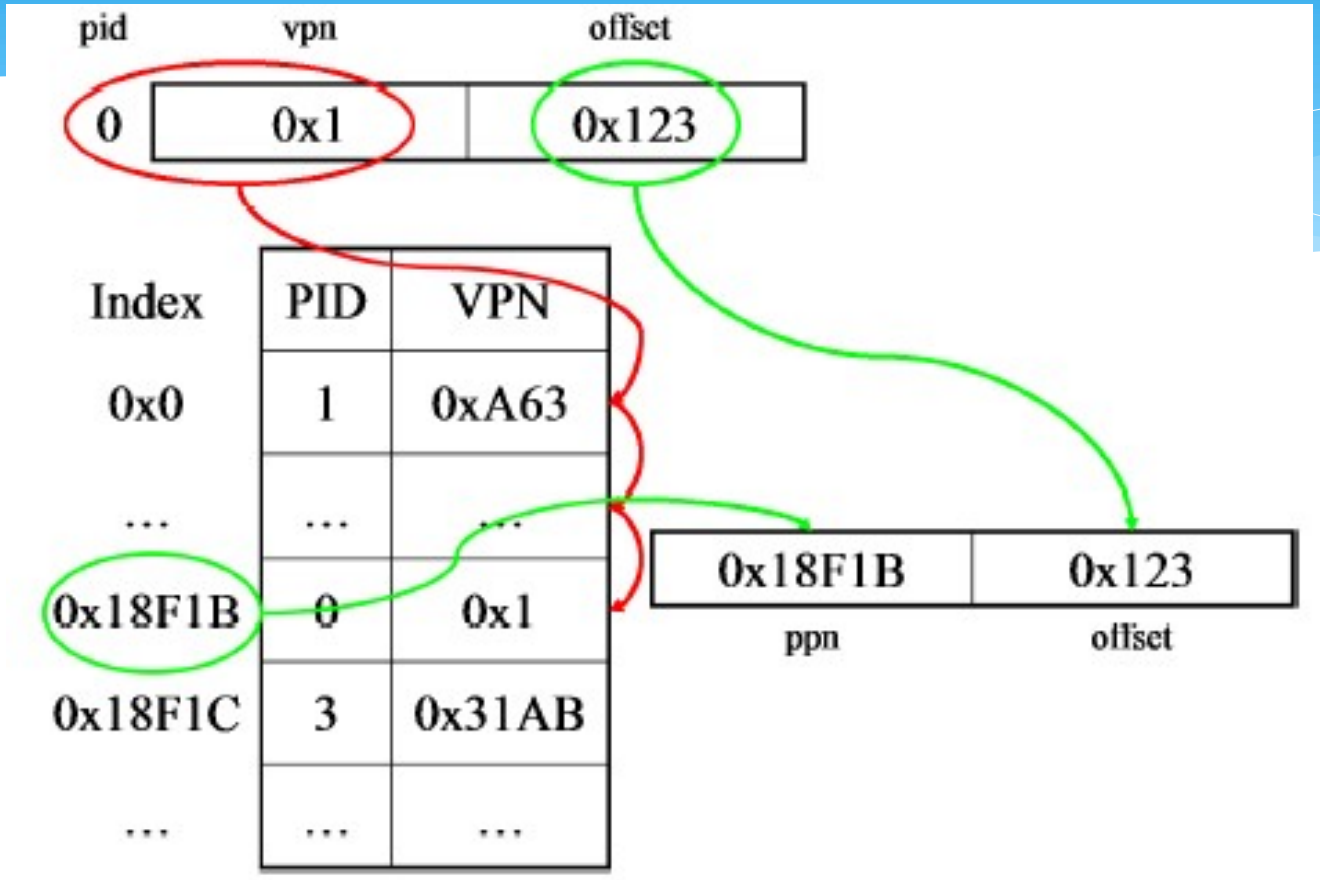
反置页表及其地址转换

反置页表

反置页表地址转换过程如下：

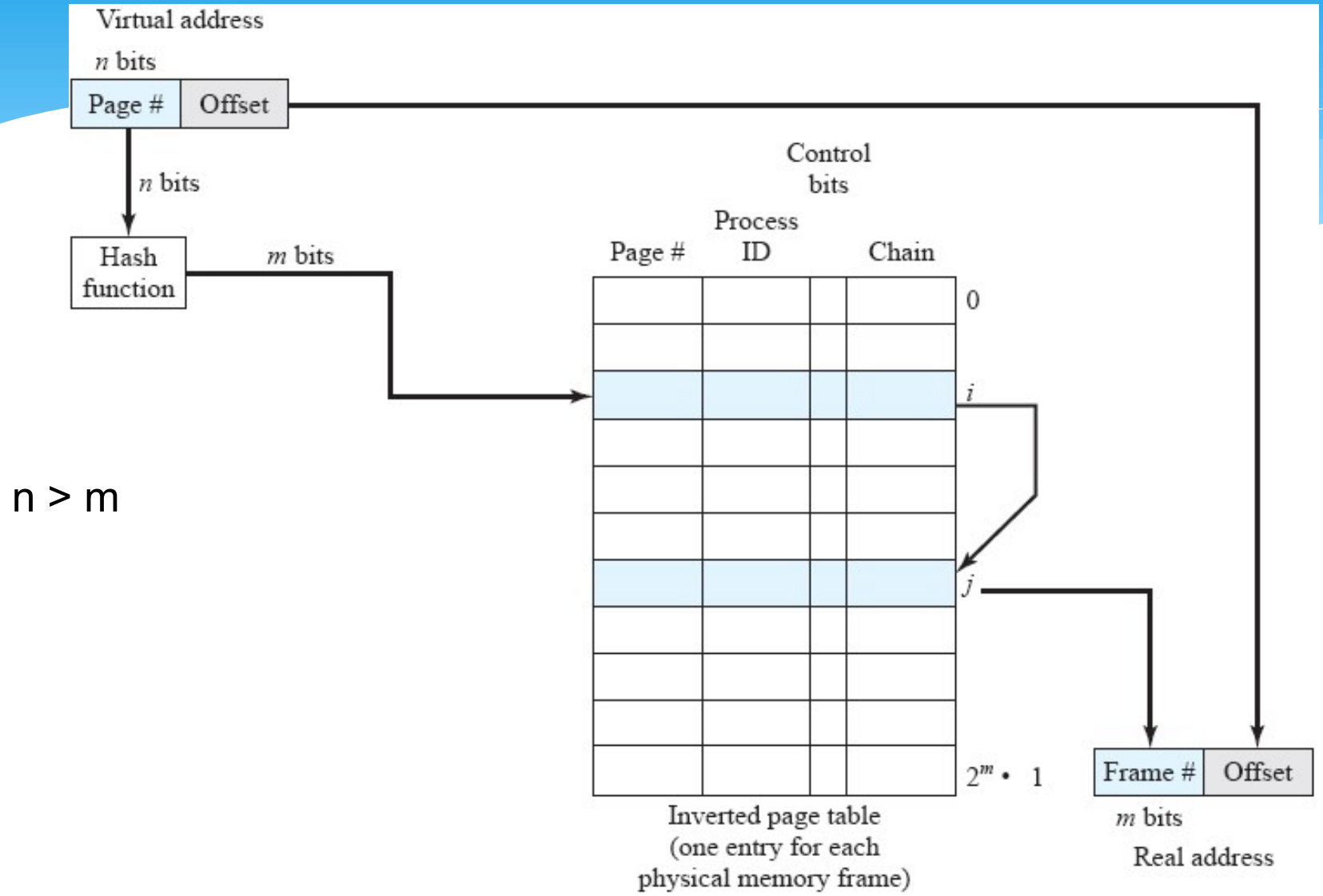
逻辑地址给出进程标识和页号，用它们去比较IPT，若整个反置页表中未能找到匹配的页表项，说明该页不在主存，产生缺页中断，请求操作系统调入；否则，该表项的序号便是页框号，块号加上位移，便形成物理地址。

线性反置页表



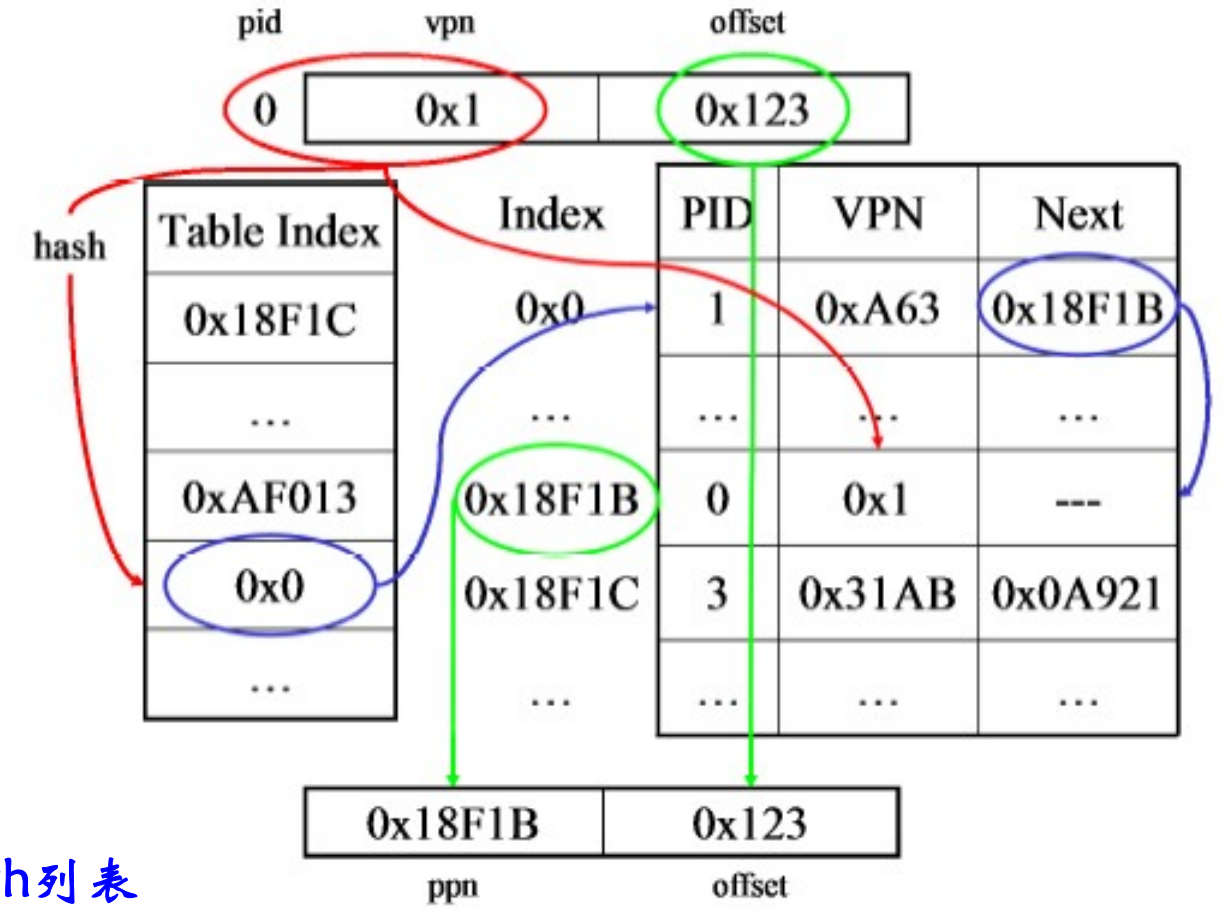
linear inverted page table.

反置页表


Virtual address
n bits
Page #
Offset
n bits
Hash function
m bits
Control bits
Process ID
Page #
Chain
0
i
j
2^m • 1
Frame #
Offset
m bits
Real address
Inverted page table
(one entry for each
physical memory frame)

n > m

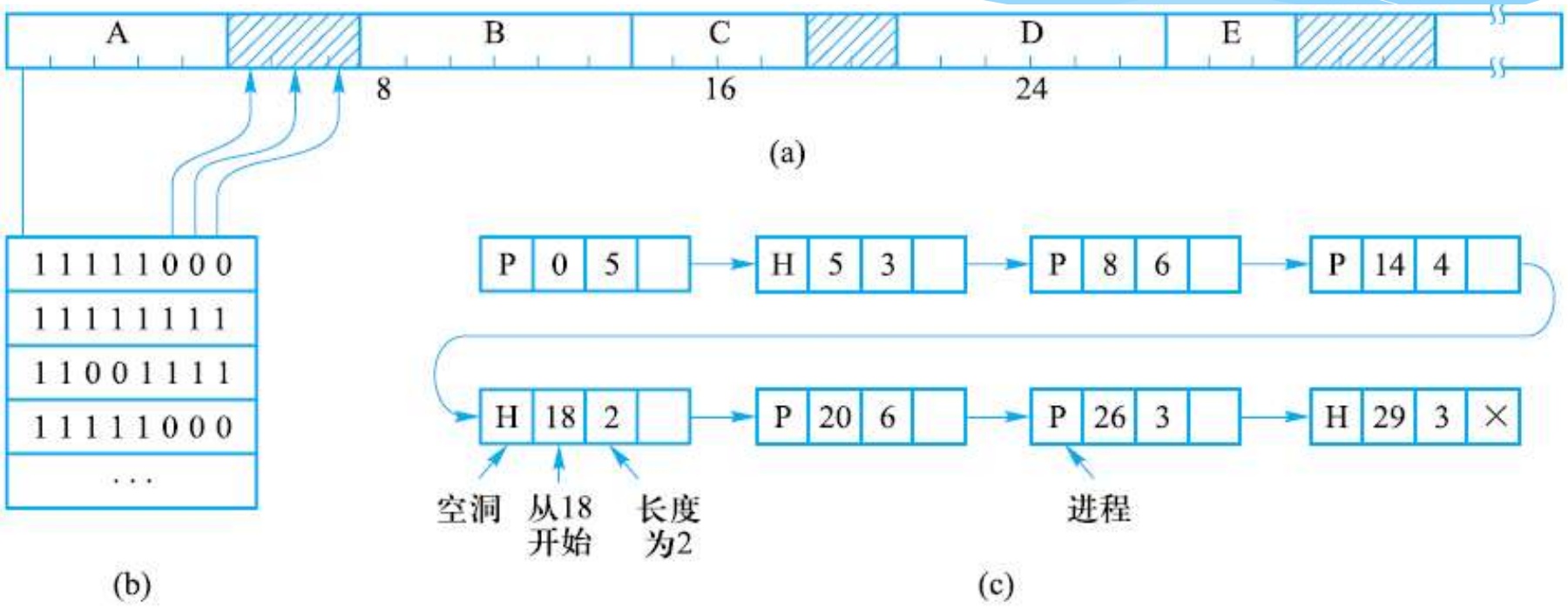
哈希线性反置页表



使用Hash列表

页表的结构称为“反向”是因为它使用帧号而不是虚拟页号来索引页表项

主存分配的位示图和链表方法



南京大学
NANJING UNIVERSITY

分页存储空间的页面共享和保护

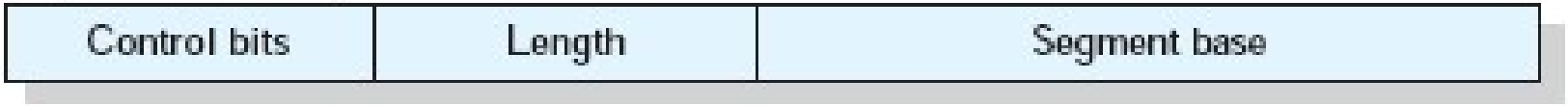
* 数据共享--允许不同进程对共享的数据页用不同的页号，只要让各自页表中的有关表项指向共享的数据页框
* 程序共享--由于指令包含指向其他指令或数据的地址，进程依赖于这些地址才能执行，不同进程中正确执行共享代码页面，必须为它们在所有逻辑地址空间中指定同样页号

南京大学
NANJING UNIVERSITY

段页式存储管理

Virtual address

Segment table entry



Page table entry

P • present bit
M • modified bit

补充5：页的大小设计

Page Size

* Smaller page size, less amount of internal fragmentation
* Smaller page size, more pages required per process
* More pages per process means larger page tables, and larger page tables means large portion of page tables in virtual memory
* Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better

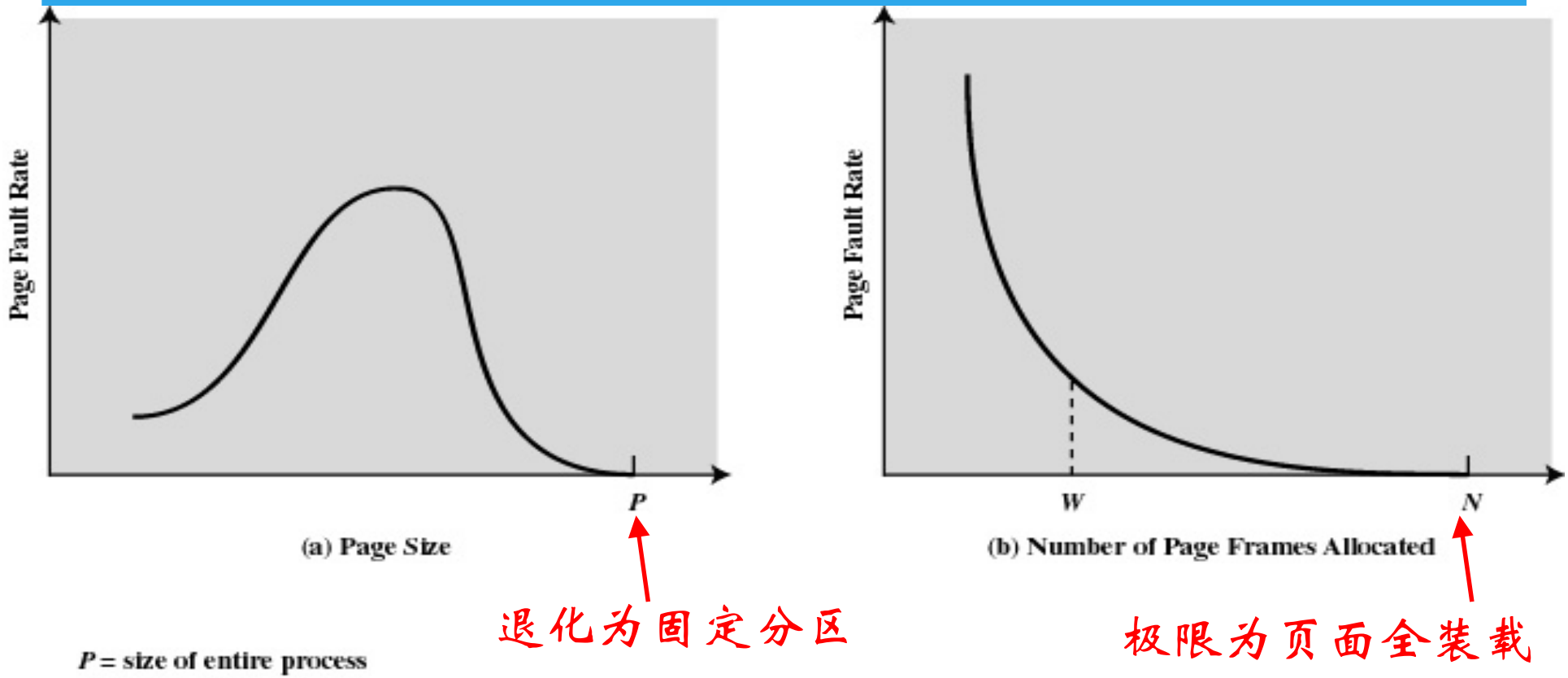


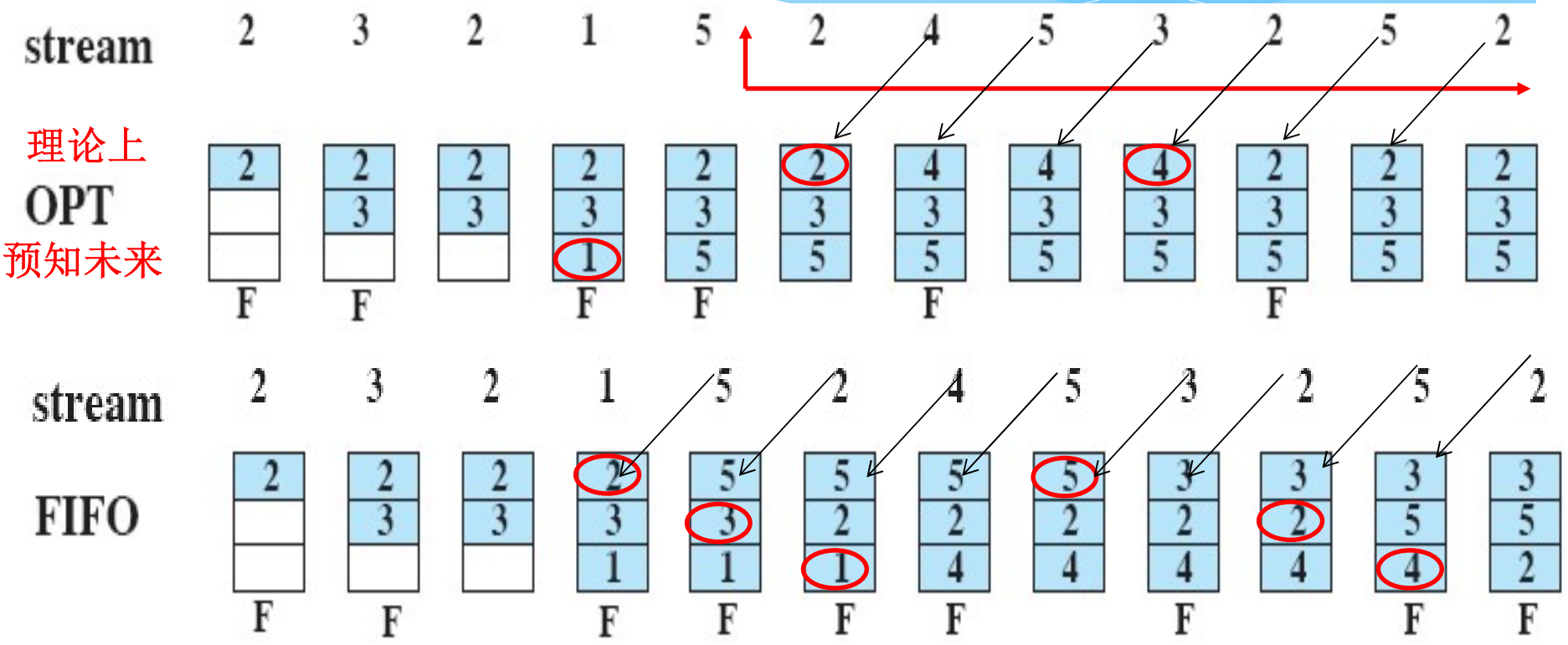
Figure 8.11 Typical Paging Behavior of a Program

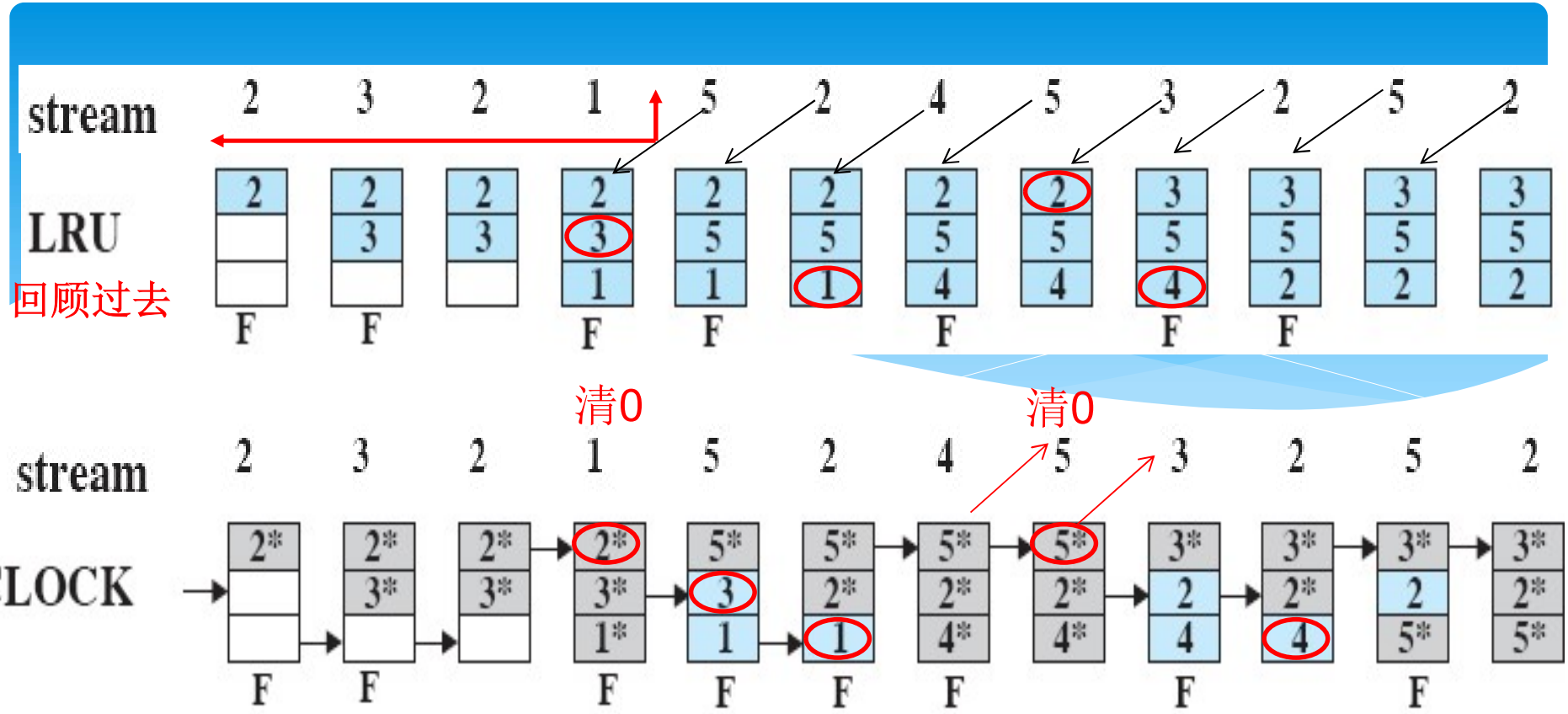
Page Size

* Multiple page sizes provide the flexibility needed to effectively use a TLB
* Most operating system support only one page size

补充6：页面替换算法

示例:页面替换算法


stream
2 3 2 1 5 2 4 5 3 2 5 2
理论上
OPT
预知未来
F F F F F F
stream
2 3 2 1 5 2 4 5 3 2 5 2
FIFO
F F F F F F F F F F



星号表示相应的使用位等于1，箭头表示指针的当前位置。

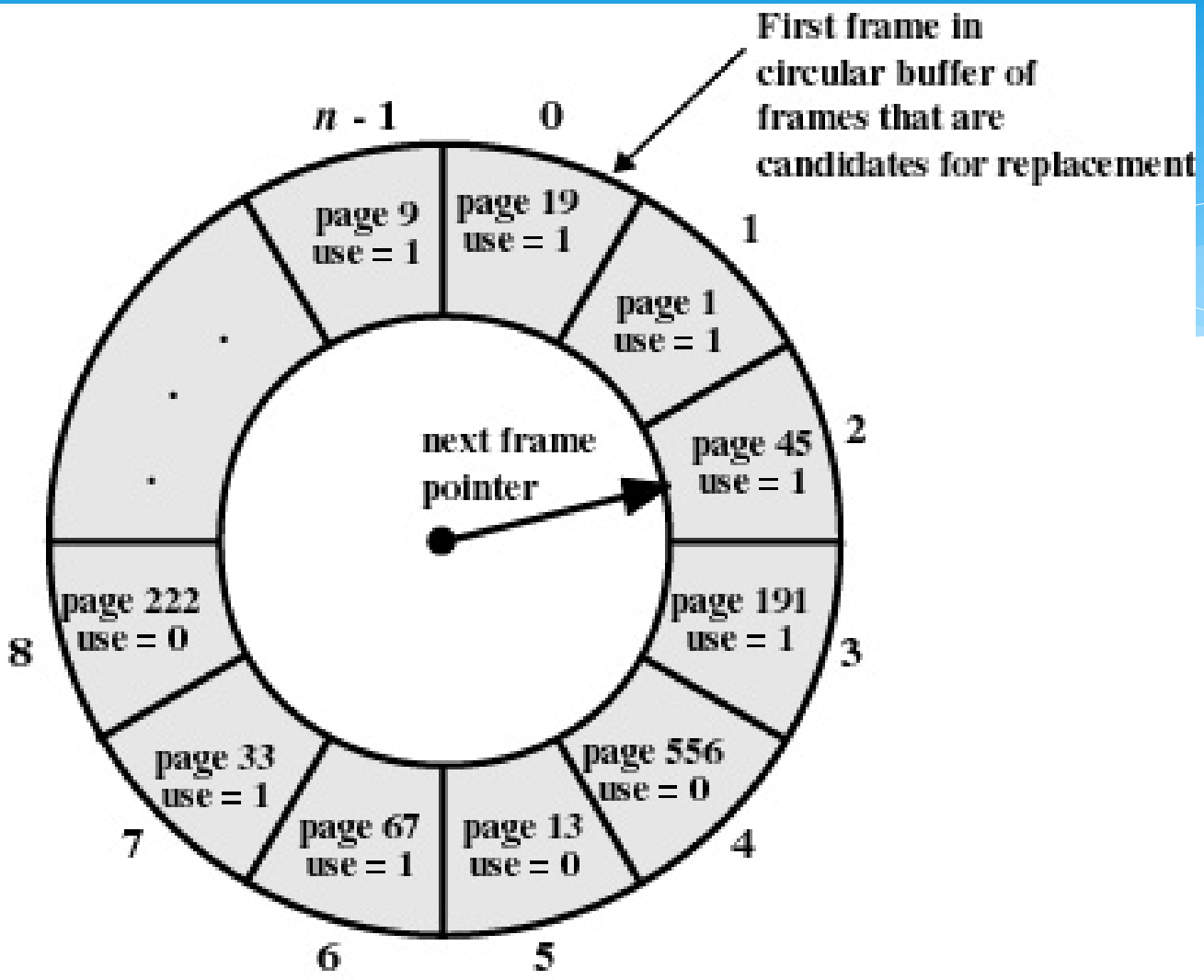
当一页被替换时，指向下一帧。 虽然早就进来，但是最近使用过，所以不急着替换

当需要替换一页时，扫描缓冲区，查找使用位被置为0的一帧。

每当遇到一个使用位为1的帧时，就将该位重新置为0；

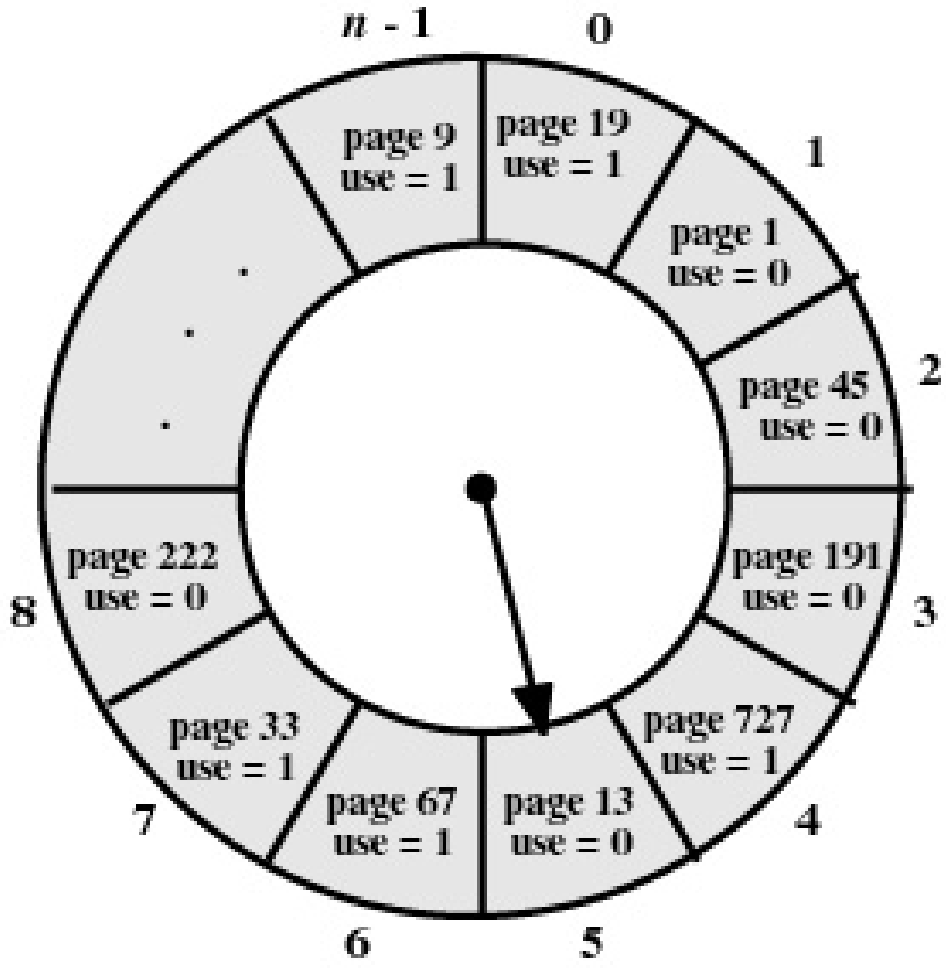
如果在这个过程开始时，所有帧的使用位均为0，选择遇到的第一个帧替换；

如果所有帧的使用位为1，则指针在缓冲区中完整地循环一周，把所有使用位都置为0，并且停留在最初的位置上，替换该帧中的页。



(a) State of buffer just prior to a page replacement

Figure 8.16 Example of Clock Policy Operation



(b) State of buffer just after the next page replacement

Figure 8.16 Example of Clock Policy Operation

Belady's Anomaly (Belady异常)

Belady 异常

FIFO	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
	F	F	F	F	F	F	F			F	F	

3 frames, 9 page-faults

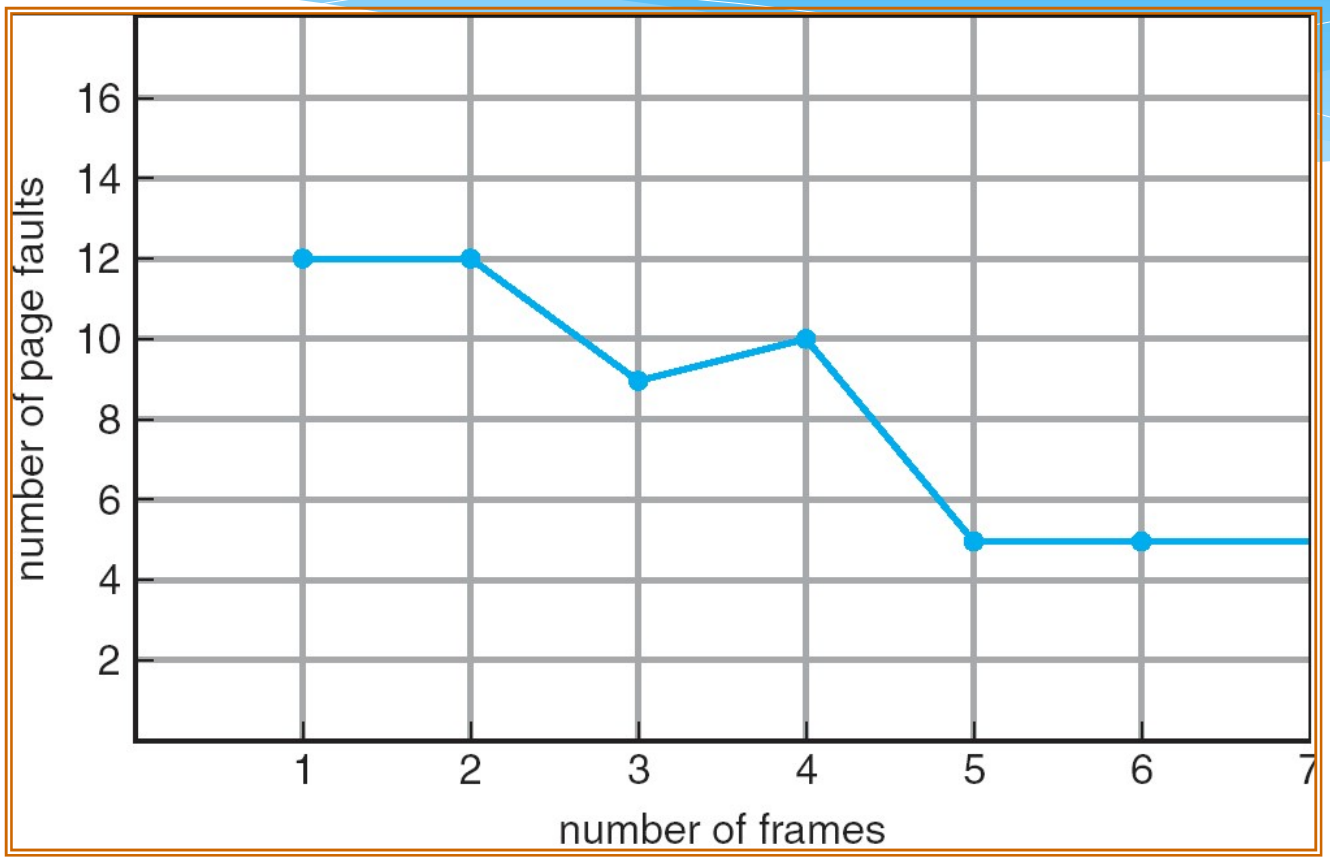
FIFO	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
	F	F	F	F			F	F	F	F	F	F

4 frames, 10 page-faults

more frames ⇒ more page faults

对应《操作系统教程(第4版)》pp.265 “Belady异常”

FIFO Illustrating Belady's Anomaly (FIFO算法的Belady异常)



对应《操作系统教程(第5版)》pp.225 “Belady异常”

Comparison of Placement Algorithms

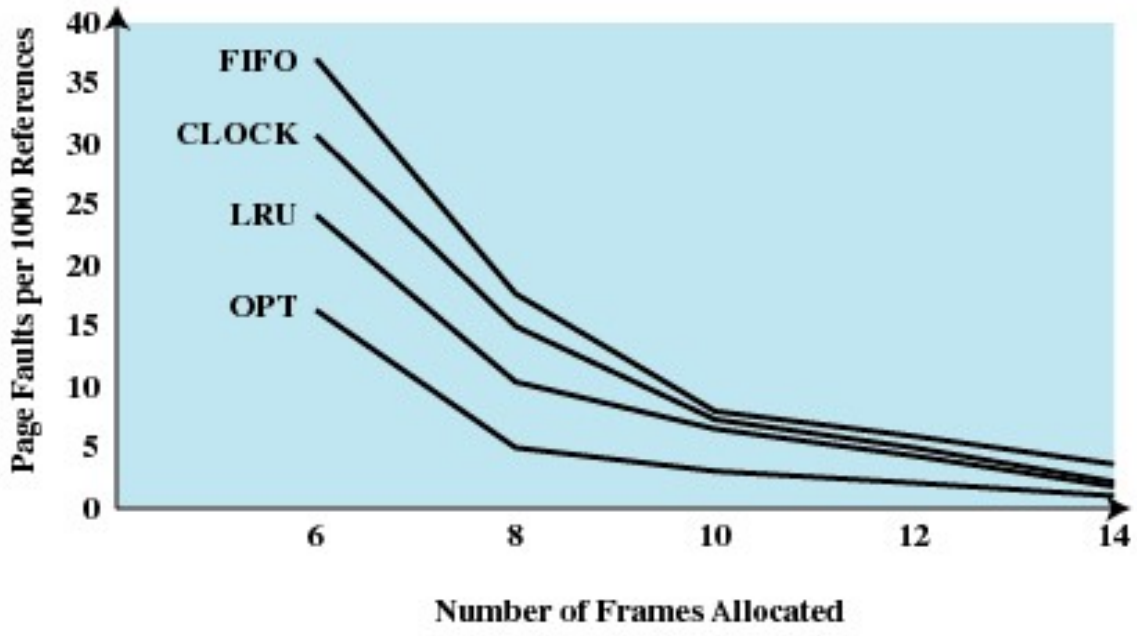


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

这是一个整体的性能比较，对于个案，有可能Clock优于LRU

Basic Replacement Algorithms

* Page Buffering
  * Replaced page is added to one of two lists
    * free page list if page has not been modified
    * modified page list
  * Replaced page remains in memory
    * If referenced again, it is returned at little cost
    * Modified pages are written out in cluster

Resident Set Size
（驻留集规模）

* Fixed-allocation
  * gives a process a fixed number of pages within which to execute
  * when a page fault occurs, one of the pages of that process must be replaced
* Variable-allocation
  * number of pages allocated to a process varies over the lifetime of the process

Fixed Allocation, Local Scope

* Number of frames allocated to process is fixed
* Page to be replaced is chosen from among the frames allocated to the process

Variable Allocation, Global Scope

* Number of frames allocated to process is variable
* Page to be replaced is chosen from all frames
* Easiest to implement
* Adopted by many operating systems
* Operating system keeps list of free frames
* Free frame is added to resident set of process when a page fault occurs

Variable Allocation, Local Scope

* Number of frames allocated to process is variable
* Page to be replaced is chosen from among the frames allocated to the process
* When new process added, allocate number of page frames based on application type, program request, or other criteria
* When page fault occurs, select page from among the resident set of the process that suffers the fault
* Reevaluate allocation from time to time

局部页面替换算法

1) 局部最佳页面替换算法

2) 工作集模型和工作集置换算法

1) 局部最佳页面替换算法(1)

Barton G. Prieve, Robert S. Fabry: VMIN-An Optimal Variable-Space Page Replacement Algorithm. Commun. ACM 19(5): 295-297 (1976)

* 1976年Prieve提出一种局部最佳页面替换算法MIN (Local Minimum)，它与全局最佳替换算法类似，需事先知道程序的页面引用串，再根据进程行为改变驻留页面数量
* 实现思想：进程在时刻t访问某页面，如果该页面不在主存中，导致一次缺页，把该页面装入一个空闲页框
* 不论发生缺页与否，算法在每一步要考虑引用串，**如果该页面在时间间隔(t, t+ τ)内未被再次引用，那么就移出；否则，该页被保留在进程驻留集中**
* τ为一个系统常量，间隔(t, t+ τ)称作滑动窗口。例子中τ=3

局部最佳页面替换算法(2)

时刻 t	0	1	2	3	4	5	6	7	8	9	10
引用串	P_4	P_3	P_3	P_4	P_2	P_3	P_5	P_3	P_5	P_1	P_4
P_1	–	–	–	–	–	–	–	–	–	√	–
P_2	–	–	–	–	√	–	–	–	–	–	–
P_3	–	√	√	√	√	√	√	√	–	–	–
P_4	√	√	√	√	–	–	–	–	–	–	√
P_5	–	–	–	–	–	–	√	√	√	–	–
In_t		P_3			P_2		P_5			P_1	P_4
Out_t					P_4	P_2			P_3	P_5	P_1

局部最佳页面替换算法(3)

时刻	引用串	驻留集		Out_t	滑动窗口
		已驻留	In_t		
T0	P4	P4			(0,0+3)看到p4
T1	P3	P4	P3		(1,1+3)看到p3, p4
T2	P3	P3,p4			(2,2+3)看到p3, p4
T3	P4	P3,p4			(3,3+3)看到p3, p4
T4	P2	P3	P2	p4	(4,4+3)中看不到p4
T5	P3	P3		P2	(5,5+3)中看不到p2
T6	P5	P3	P5		(6,6+3)看到p3, p5
T7	P3	P3, P5			(7,7+3)看到p3, p5
T8	P5	P5		P3	(8,8+3)中看不到p3
T9	P1		P1	P5	(9,9+3)中看不到p5
T10	P4		P4	P1	(10,10+3)中看不到p1

缺页总数为5次，驻留集大小在1-2之间变化，任何时刻至多两个页框被占用，通过增加τ值，缺页数目可减少，但代价是花费更多页框。

2) 工作集模型和工作集置换算法

* 进程工作集指“在某一段时间间隔内进程运行所需访问的页面集合”
* 实现思想：工作集模型用来对局部最佳页面替换算法进行模拟实现，不向前查看页面引用串，而是基于程序局部性原理向后看
* 任何给定时刻，进程不久的将来所需主存页框数，可通过考查其过去最近的时间内的主存需求做出估计

http://denninginstitute.com/denning/
Prof. Peter Denning
Distinguished Professor
Naval Postgraduate School
Computer Science, Code CS

2) 工作集模型和工作集置换算法

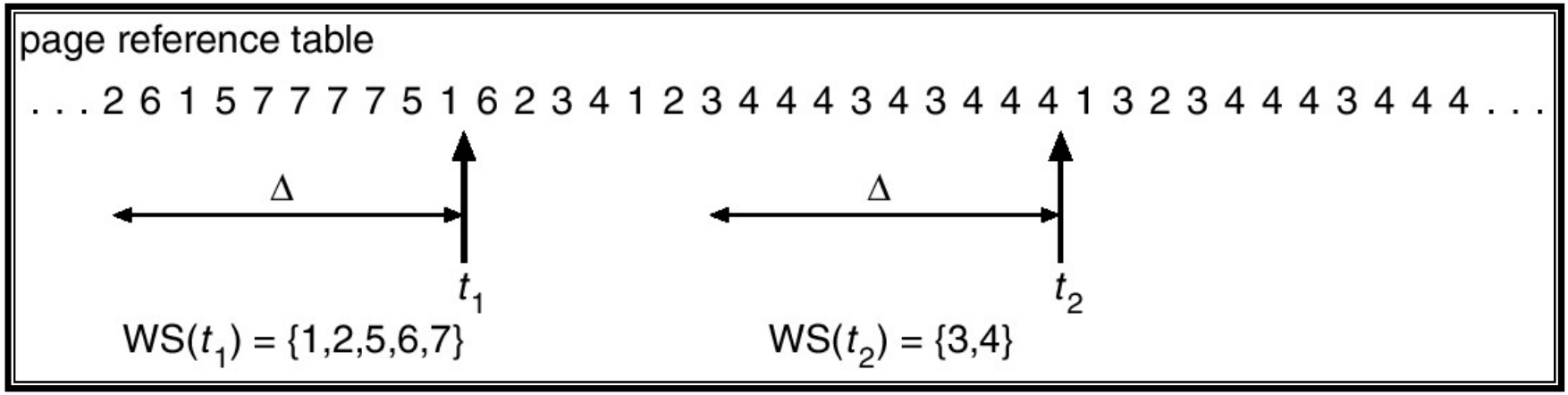
Denning教授与工作集相关的系列论文

1. Peter J. Denning: The Working Set Model for Program Behaviour. Commun. ACM 11(5): 323-333 (1968)
2. Peter J. Denning, Stuart C. Schwartz: Properties of the Working Set Model (Abstract). SOSP 1971: 130-140
3. Peter J. Denning, Stuart C. Schwartz: Properties of the Working Set Model. Commun. ACM 15(3): 191-198 (1972)
4. Peter J. Denning, Donald R. Slutz: Generalized Working Sets for Segment Reference Strings. Commun. ACM 21(9): 750-759 (1978)
5. Peter J. Denning: Working Sets Past and Present. IEEE Trans. Software Eng. 6(1): 64-84 (1980)
6. Peter J. Denning: The Working Set Model for Program Behaviour (Reprint). Commun. ACM 26(1): 43-48 (1983)
7. Peter J. Denning: Working Set Analytics. ACM Comput. Surv. 53(6): 113:1-113:36 (2021) //关于工作集最新的综述论文

进程工作集

* 指“在某一段时间间隔内进程运行所需访问的页面集合”，W(t，Δ)表示在时刻t-Δ到时刻t之间((t-Δ，t))所访问的页面集合，进程在时刻t的工作集
* Δ是系统定义的一个常量。变量Δ称为“工作集窗口尺寸”，可通过窗口来观察进程行为，还把工作集中所包含的页面数目称为“工作集尺寸”
* Δ=3

Working-set model



Sequence of Page References	Window Size, •			
	2	3	4	5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

命中

工作集替换示例

时刻 t	0	1	2	3	4	5	6	7	8	9	10
引用串	P_1	P_3	P_3	P_4	P_2	P_3	P_5	P_3	P_5	P_1	P_4
P_1	√	√	√	√	—	—	—	—	—	√	√
P_2	—	—	—	—	√	√	√	√	—	—	—
P_3	—	√	√	√	√	√	√	√	√	√	√
P_4	√	√	√	√	√	√	√	—	—	—	√
P_5	√	√	—	—	—	—	√	√	√	√	√
In_t		P_3			P_2		P_5			P_1	P_4
Out_t			P_5		P_1			P_4	P_5		

其中，p1在时刻t=0被引用，p4在时刻t=-1被引用，
而p5在时刻t=-2被引用，Δ=3

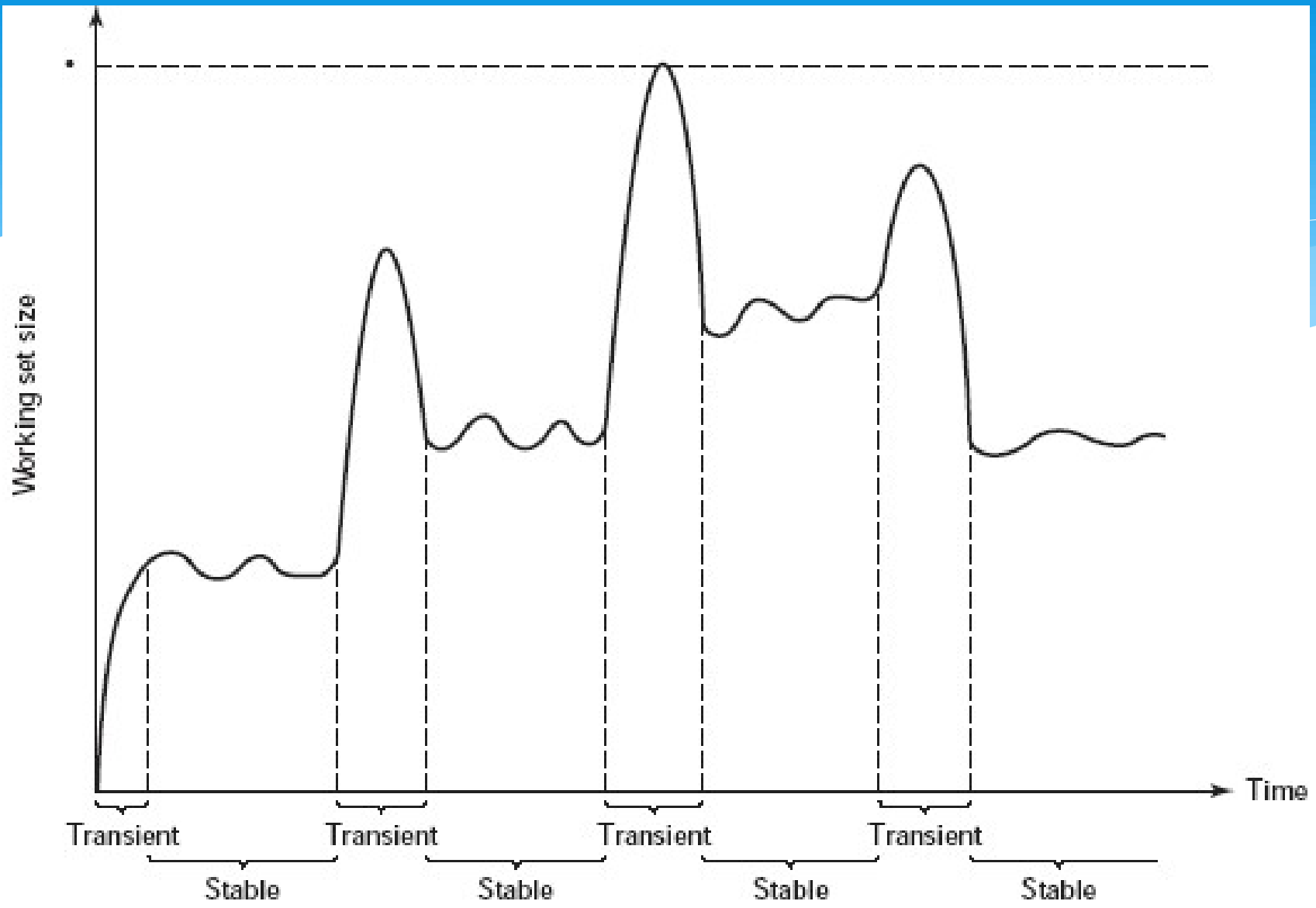
工作集替换示例进一步说明

时刻t	-2	-1	0	1	2	3	4	5	6	7	8	9	10
引用串	p5	p4	p1	p3	p3	p4	p2	p3	p5	p3	p5	p1	p4
p1			√	√	√	√	—	—	—	—	—	√	√
p2			—	—	—	—	√	√	√	√	—	—	—
p3			—	√	√	√	√	√	√	√	√	√	√
p4		√	√	√	√	√	√	√	√	—	—	—	√
p5	√	√	√	√	—	—	—	—	√	√	√	√	√
In_t				p3			p2		p5			p1	p4
OUT_t					p5		p1			p4	p2		

工作集页面替换算法

时刻	引用串	工作集		Outi	(t-Δ，t)= (t-3，t)
		已驻留	Ini		
T-2	P5		P5		
T-1	P4	P5	P4		
T0	P1	P4,p5	P1		
T1	P3	P1,P4,p5	P3		(1-3,1)看到p1, p3 p4, p5
T2	P3	P1,P3,p4		p5	(2-3,2)看到p1, p3, p4。 P5出。
T3	P4	P1,P3,p4			(3-3,3)看到p1, p3, p4
T4	P2	P3,p4	P2	P1	(4-3,4)看到p2, p3, p4。 P1出。
T5	P3	P2,P3, P4			(5-3,5)看到p2, p3, p4
T6	P5	P2,P3, P4	P5		(6-3,6)看到p2, p3, p4, p5
T7	P3	P2,P3, P5		P4	(7-3,7)看到p2, p3, p5。 P4出。
T8	P5	P3, P5		P2	(8-3,8)看到p3, p5。 P2出。
T9	P1	P3, P5	P1		(9-3,9)看到p1, p3, P5
T10	P4	P1,P3, P5	P4		(10-3,10)看到p1, p3, p4,p5

工作集的大小会随着命中率而调整



Typical Graph of Working Set Size [MAEK87]

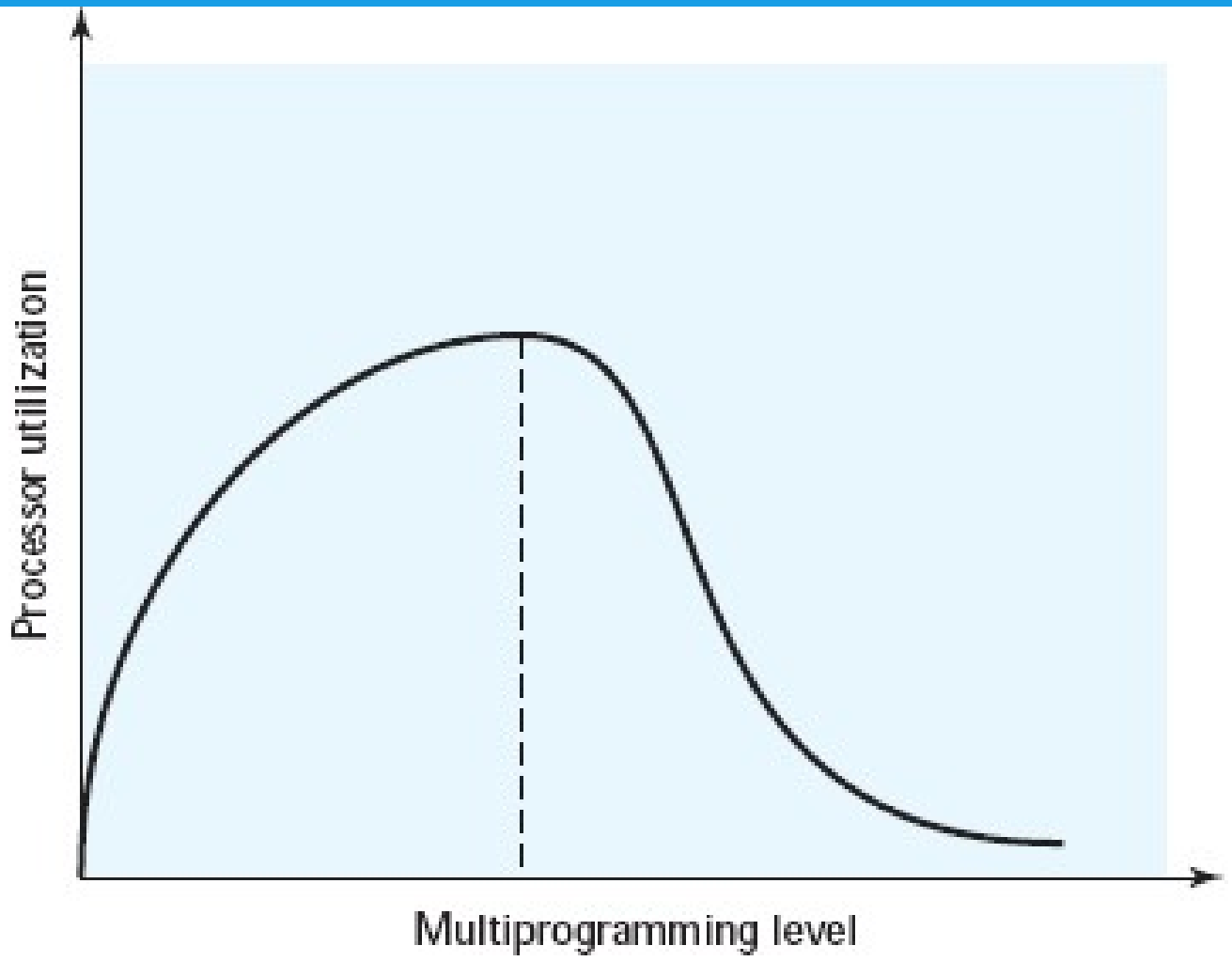
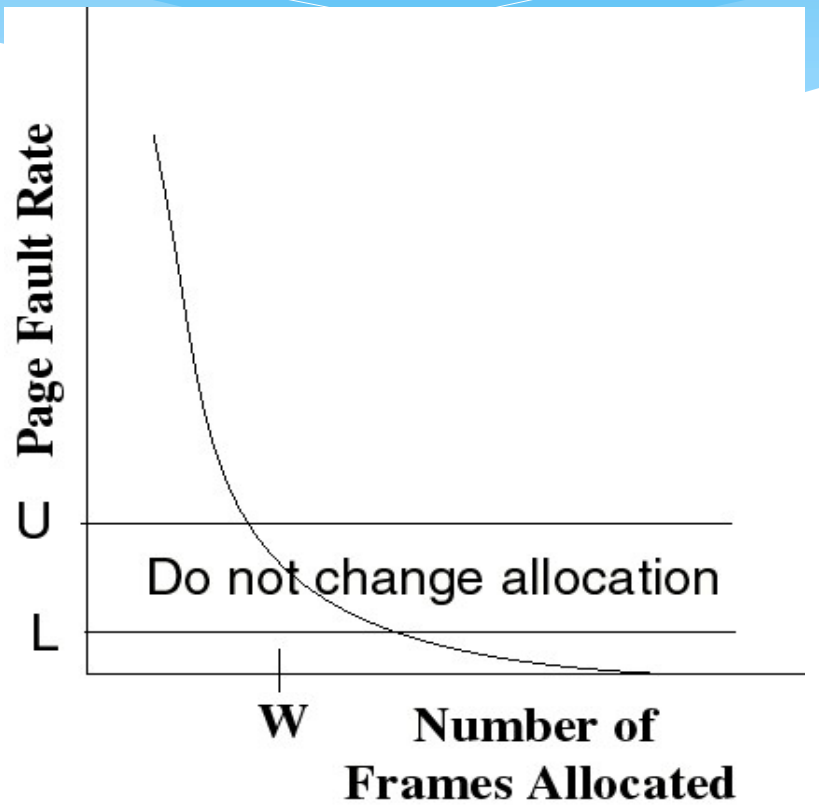


Figure 8.21 Multiprogramming Effects

The Page-Fault Frequency Strategy

* Define an upper bound U and lower bound L for page fault rates
* Allocate more frames to a process if fault rate is higher than U
* Allocate less frames if fault rate is < L
* The resident set size should be close to the working set size W
* We suspend the process if the PFF > U and no more free frames are available



通过工作集确定驻留集大小

* (1)监视每个进程的工作集，只有属于工作集的页面才能留在主存；
* (2)定期地从进程驻留集中删去那些不在工作集中的页面；
* (3)仅当一个进程的工作集在主存时，进程才能执行。

补充7：TLB快表，页表，缺页

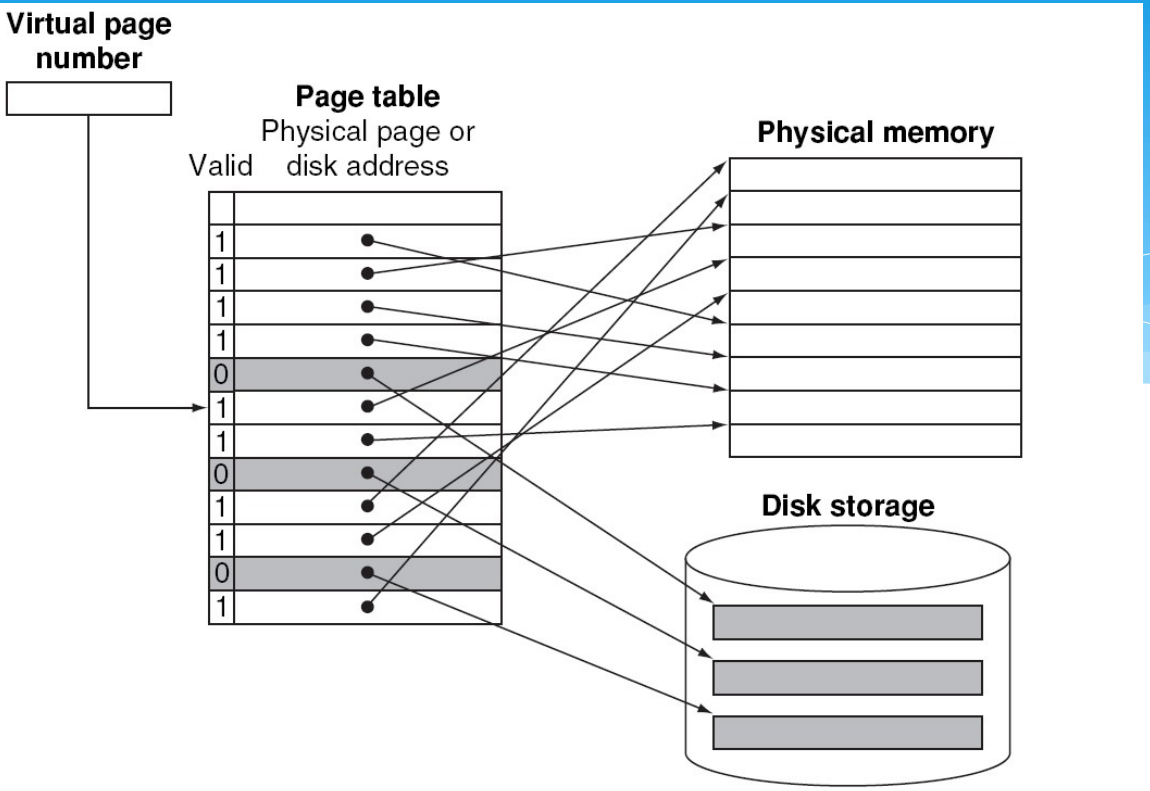


FIGURE 5.28 The page table maps each page in virtual memory to either a page in main memory or a page stored on disk, which is the next level in the hierarchy. The virtual page number is used to index the page table. If the valid bit is on, the page table supplies the physical page number (i.e., the starting address of the page in memory) corresponding to the virtual page. If the valid bit is off, the page currently resides only on disk, at a specified disk address. In many systems, the table of physical page addresses and disk page addresses, while logically one table, is stored in two separate data structures. Dual tables are justified in part because we must keep the disk addresses of all the pages, even if they are currently in main memory. Remember that the pages in main memory and the pages on disk are the same size.

Computer Organization and Design 5th the Hardware Software Interface, pp.435

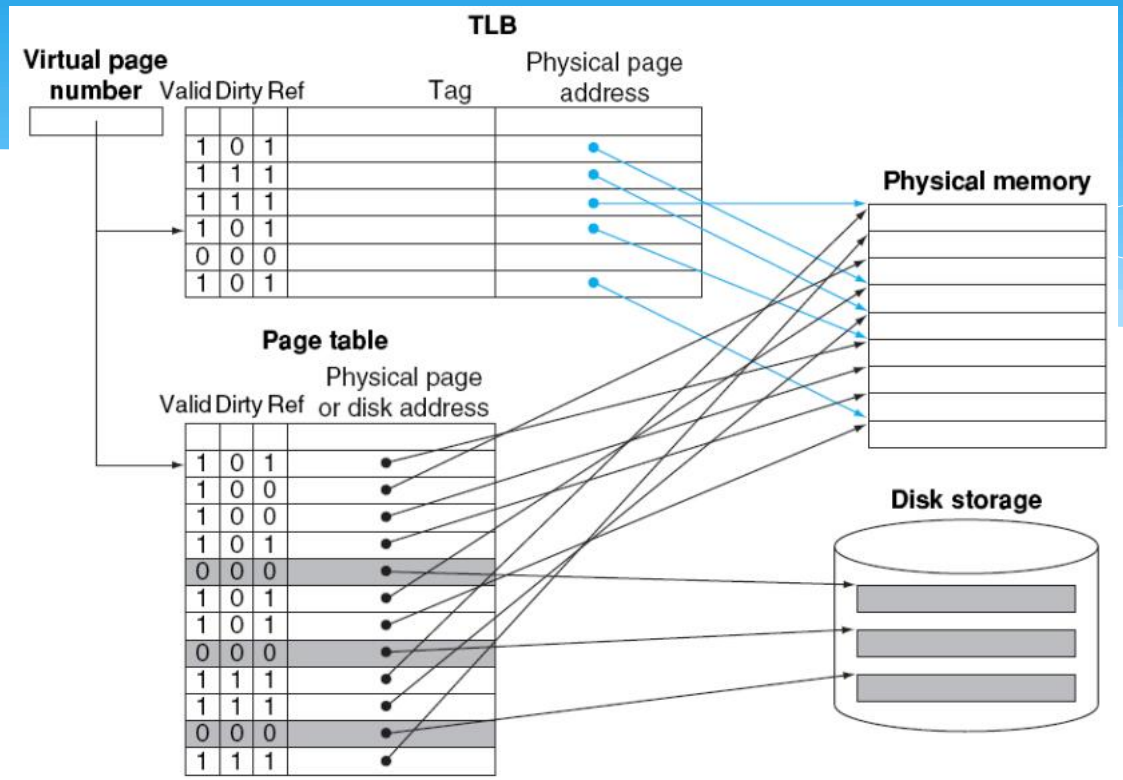


FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the page table. The TLB mappings are shown in color. Because the TLB is a cache, it must have a tag field. If there is no matching entry in the TLB for a page, the page table must be examined. The page table either supplies a physical page number for the page (which can then be used to build a TLB entry) or indicates that the page resides on disk, in which case a page fault occurs. Since the page table has an entry for every virtual page, no tag field is needed; in other words, unlike a TLB, a page table is *not* a cache.

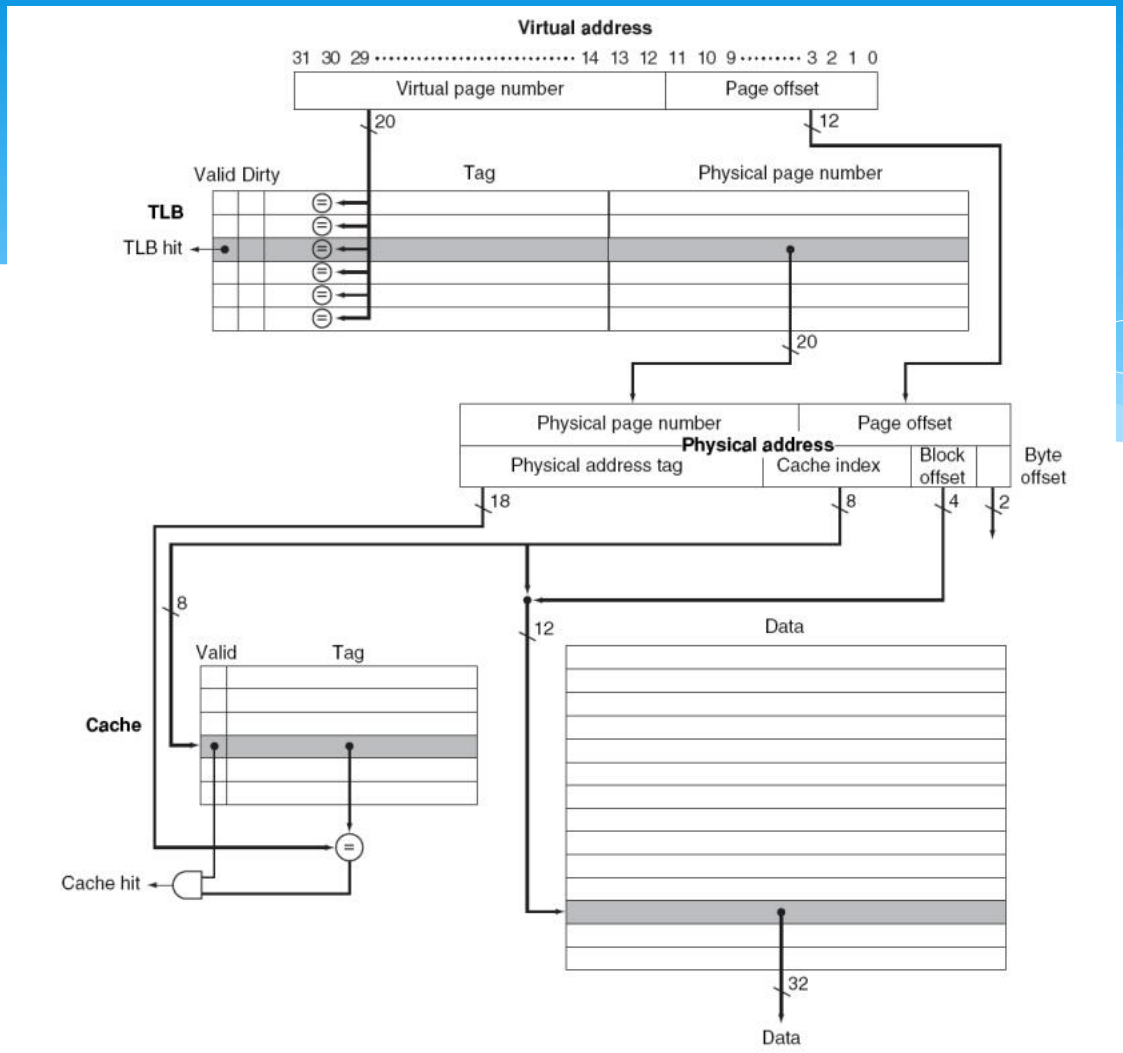


FIGURE 5.30 The TLB and cache implement the process of going from a virtual address to a data item in the Intrinsity FastMATH. This figure shows the organization of the TLB and the data cache, assuming a 4 KiB page size. This diagram focuses on a read; Figure 5.31 describes how to handle writes. Note that unlike Figure 5.12, the tag and data RAMs are split. By addressing the long but narrow data RAM with the cache index concatenated with the block offset, we select the desired word in the block without a 16:1 multiplexor. While the cache is direct mapped, the TLB is fully associative. Implementing a fully associative TLB requires that every TLB tag be compared against the virtual page number, since the entry of interest can be anywhere in the TLB. (See content addressable memories in the *Elaboration* on page 408.) If the valid bit of the matching entry is on, the access is a TLB hit, and bits from the physical page number together with bits from the page offset form the index that is used to access the cache.

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.32 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, virtual memory hit, cache miss) but never detected.