第6次编程作业：寻找互素数对

$$4 \perp 9, \quad 34 \perp 35 \tag{1}$$

$$\Pr(m \perp n) = \frac{6}{\pi^2} \approx 0.607927102 \tag{2}$$

第6次编程作业：寻找互素数对

- 硬件平台：Intel Core2 (Duo)
- 计算100000以内互素的数对的个数
- 共涉及100亿个数对
- 计算结果：6079301507
- 理论预测：6079271020
- 相对误差：万分之0.05
- 耗时：18分钟
- 本次作业：用多线程编程技术加速

第6次编程作业–要求

1. 输出结果：必须与单线程结果（6079301507）保持一致
2. 电子邮件发送内容
   - 源程序
   - 文档：描述软硬件平台，记录原始testgcd.c运行时间，2-10个线程加速后运行时间，你们的分析
3. 手写报告内容
   - 每位队员的贡献（简短描述）
   - 全体队员签名

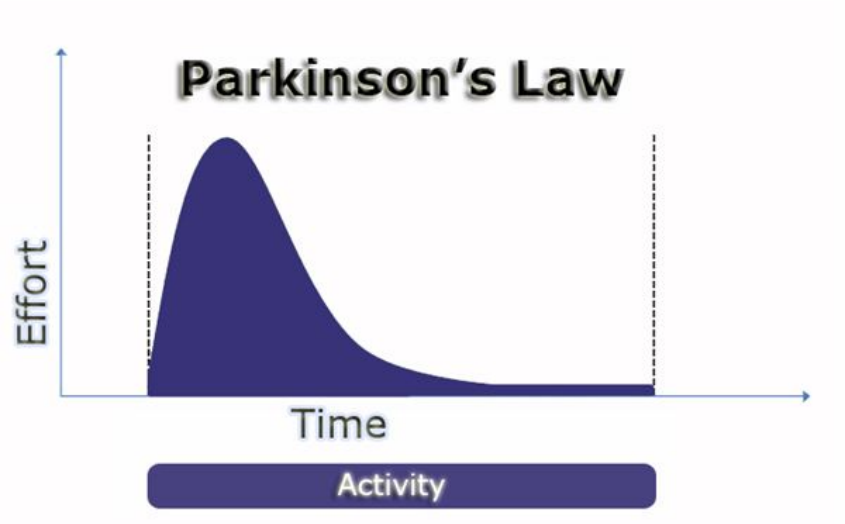
推荐阅读文章

1. 如何成为一名黑客(作者: Eric S. Raymond, 英文How to become a hacker)
2. 完全用Linux工作(作业：王垠)

Parkinson's law

1. Work expands so as to fill the time available for its completion.

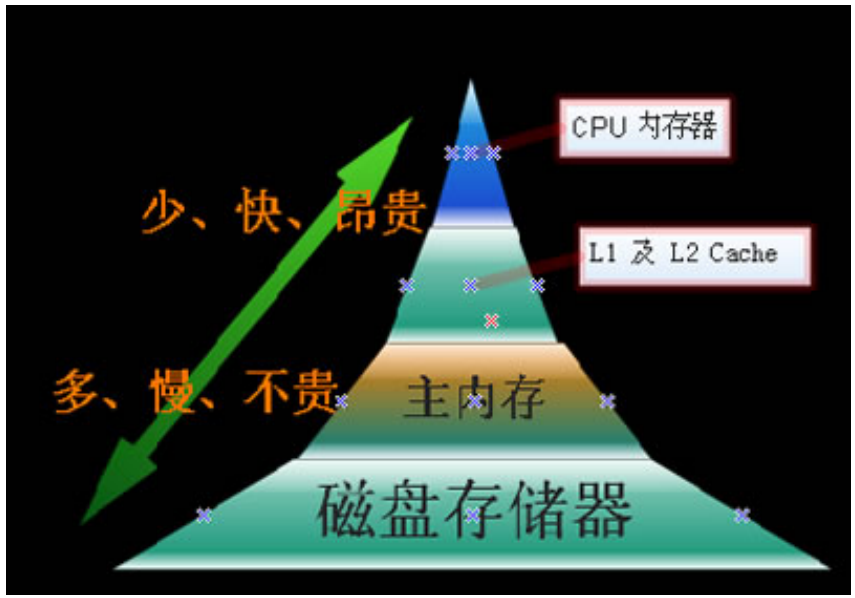
2. Data expands to fill the space available for storage.



程序员希望的内存

联系上次作业

私有的、无穷大、无穷快、便宜、持久性



内存管理器(Memory Manager)的任务

- 提供内存抽象界面
- 分配物理内存，回收物理内存
- 记录内存使用情况等

没有内存抽象：程序员直接操作物理内存

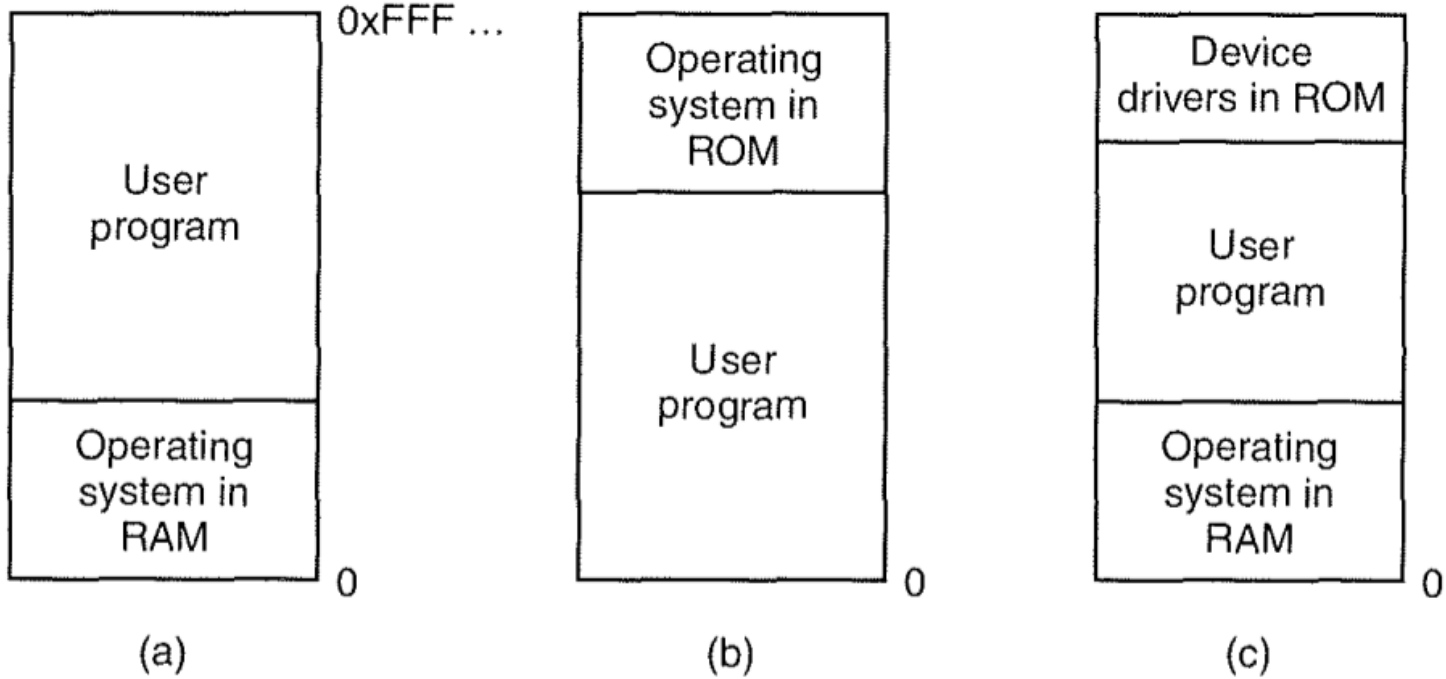


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

a. 内存中一次只能驻留一个程序：MOV REGISTER1, 1000
b. 操作系统自身代码难以保护(没有地址空间概念)

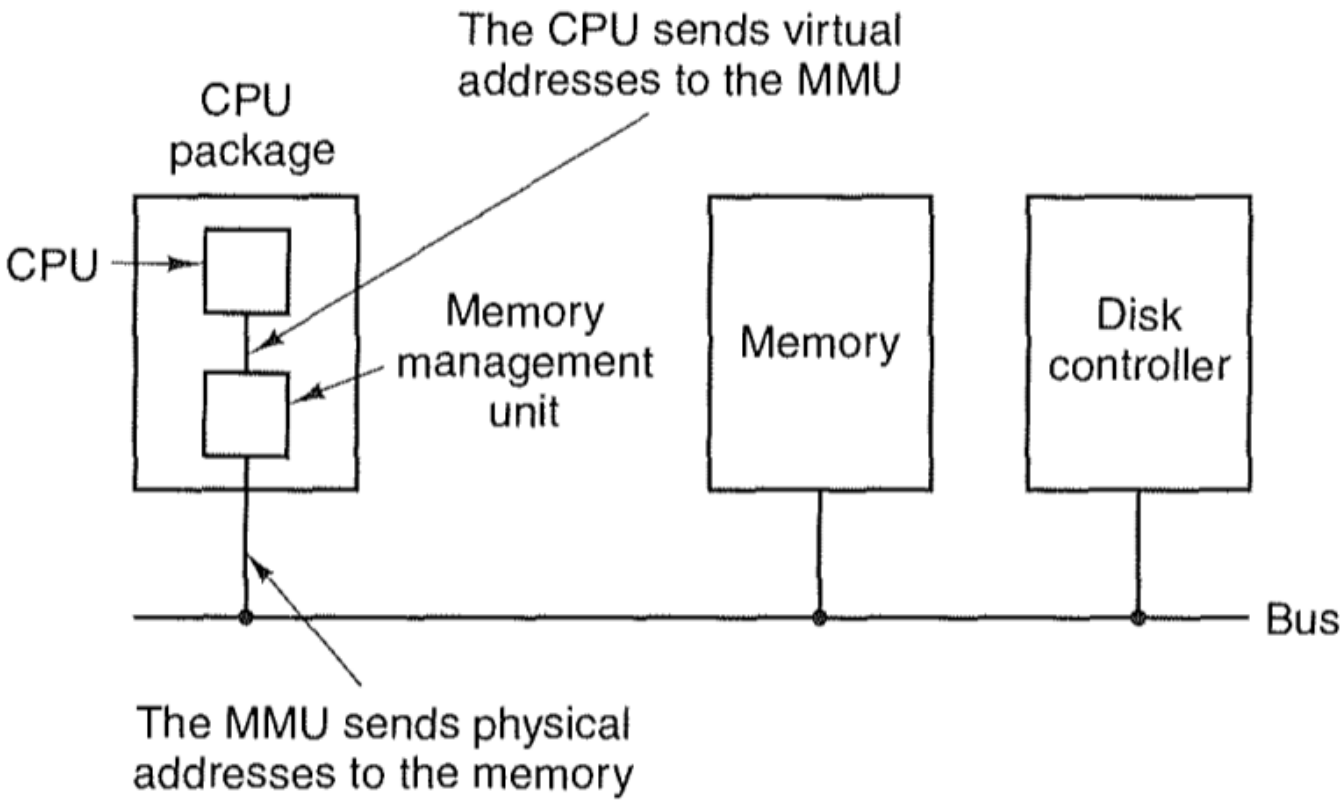
虚拟内存技术

- 问题一：如何让多个程序驻留内存?
  - 每个程序有专属地址空间
  - 程序不能非法访问其他程序的地址空间
- 问题二：如何满足程序对内存的无限需求?
  - 地址空间分成若干页面
  - 地址空间的页面映射到物理内存的页框内
  - 利用页表实现页面号到页框号的映射
  - 不是所有的页面都需要放到物理内存中
  - 缺页中断技术

虚拟内存技术

分页技术的精髓：不是所有页面都需要同时调入内存

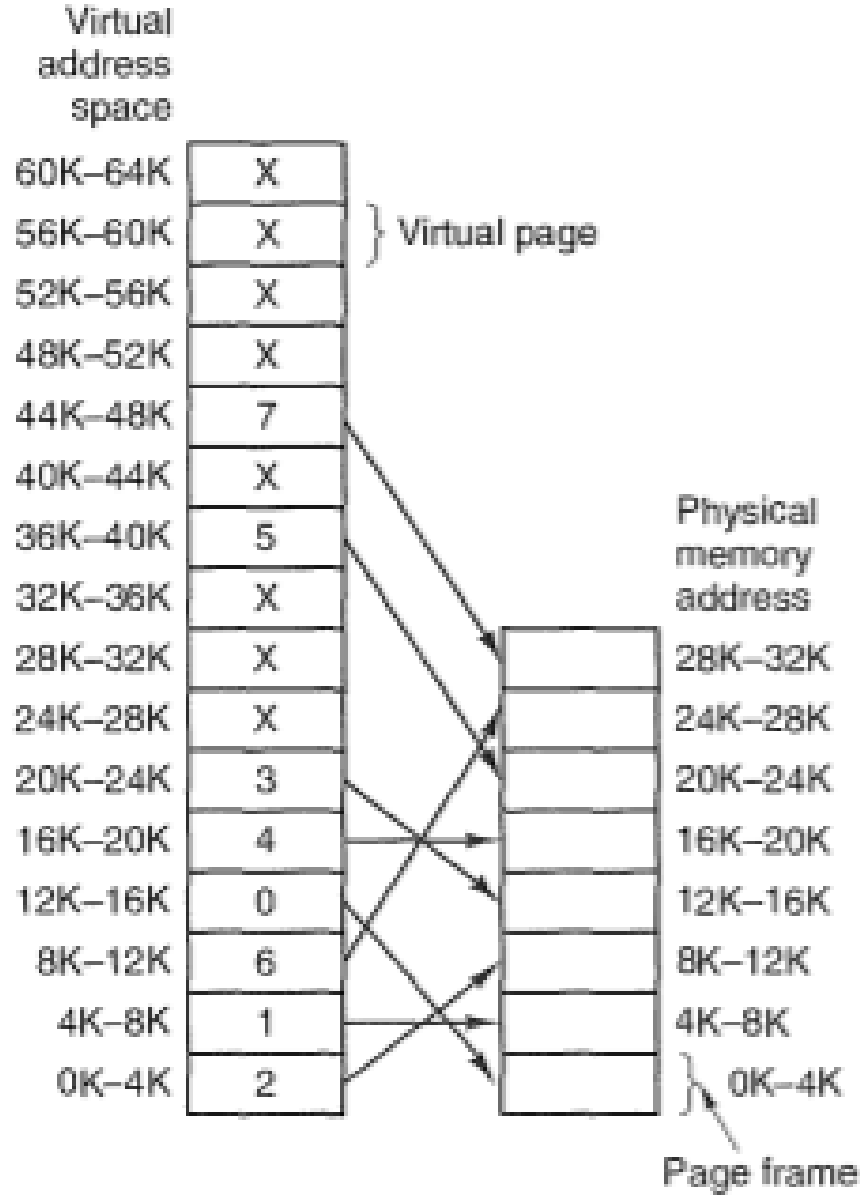
虚拟地址、物理地址及内存管理单元



注意:程序中的地址全都是虚拟地址

虚拟地址、物理地址及内存管理单元

- 虚拟地址空间：64K (16 bit)
- 物理地址空间：32K (15 bit)
- 页面大小：4K
- 共16个（虚拟）页面，8个（物理）页框
- 对于大于32K的程序，只能有32K驻留物理内存(右图数字部分)



虚拟地址、物理地址及内存管理单元

MOV REG, 20500

20500 = 5 * 4K + 20

页面5 → 页框3

12K + 20 = 12308（物理地址）

MOV REG, 32780

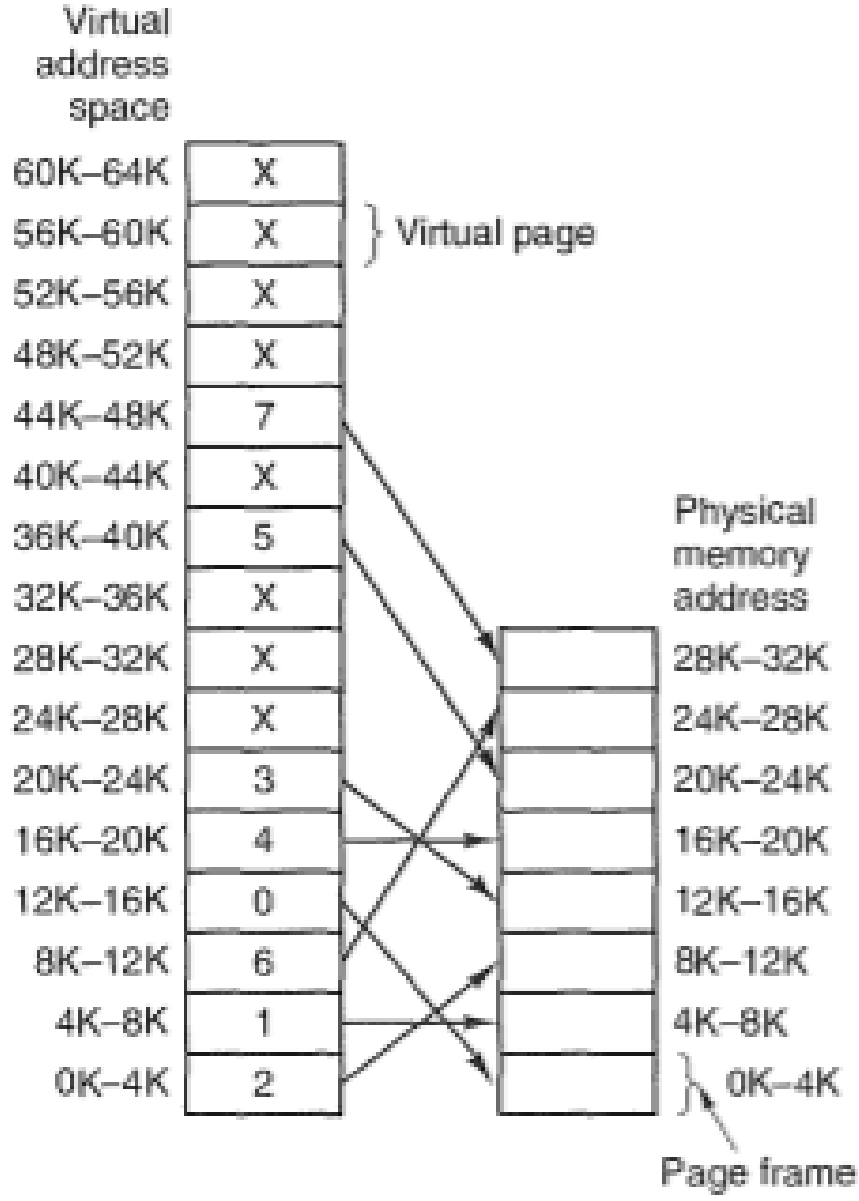
32780 = 8 * 4K + 12

对应虚拟页面8（缺页）

what next?

页表内容

需要由操作系统维护



虚拟地址、物理地址及内存管理单元

虚拟地址映射到物理地址，考虑右图例子：

- 页面大小：4K
- 页内地址为12位
- 对于16位机器而言，有4位用于页表索引
- 因此共有16个虚拟页面
- 8个物理页框（需3位)

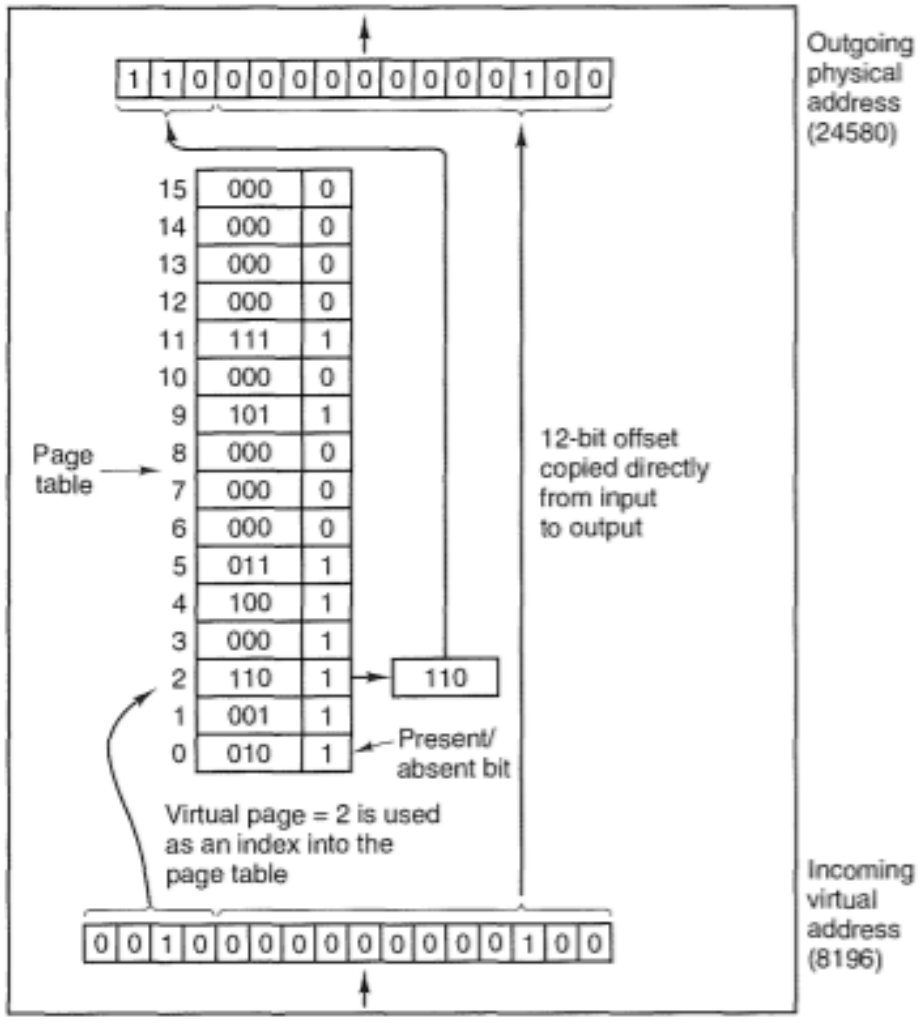
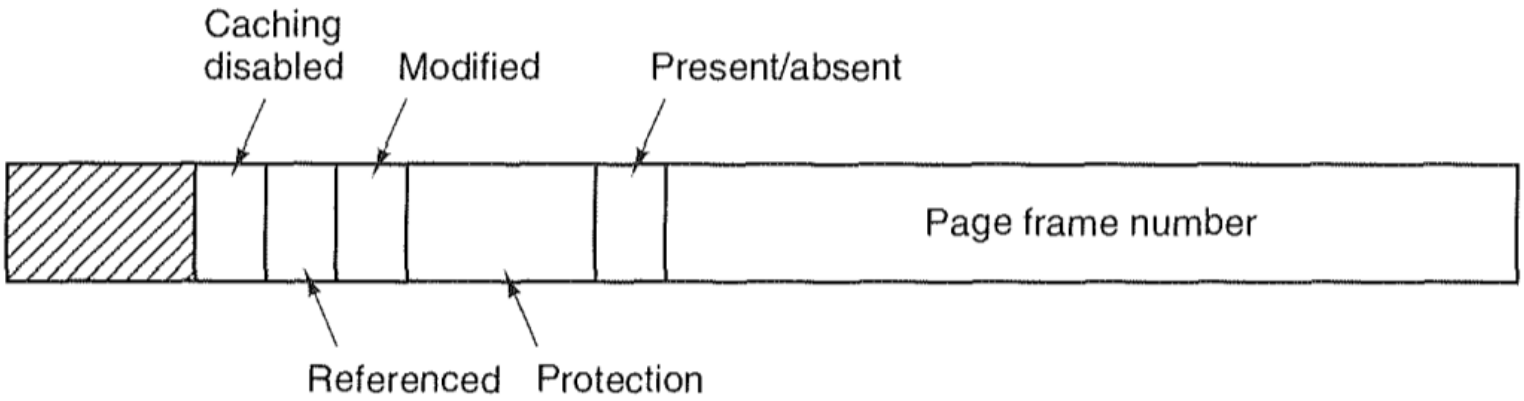


Figure 3-10. The internal operation of the MMU with 16 4-KB pages.

页表项



page frame number 描述该页表项对应的页框编号

present / absent 表示该页表是否在内存中

protection 含读、写、执行等权限信息

modified 该页表内容是否被修改过

referenced 该页表内容是否被用过（读写）

caching disabled 用于memory mapped I/O（后续）

虚拟内存技术与分页技术面临的两大问题

1. 从虚拟地址到物理地址的映射必须快
   - 每条指令都需从内存取出
   - 大量指令涉及读写内存(CISC机器)
2. 如果虚拟地址空间很大，则页表规模会特别大
   考虑页面大小4K的虚拟内存系统：
   - 32位虚拟地址: 100万个页表项
   - 64位虚拟地址：45035996亿个页表项
   - 注意：每个进程都需要单独的页表！！

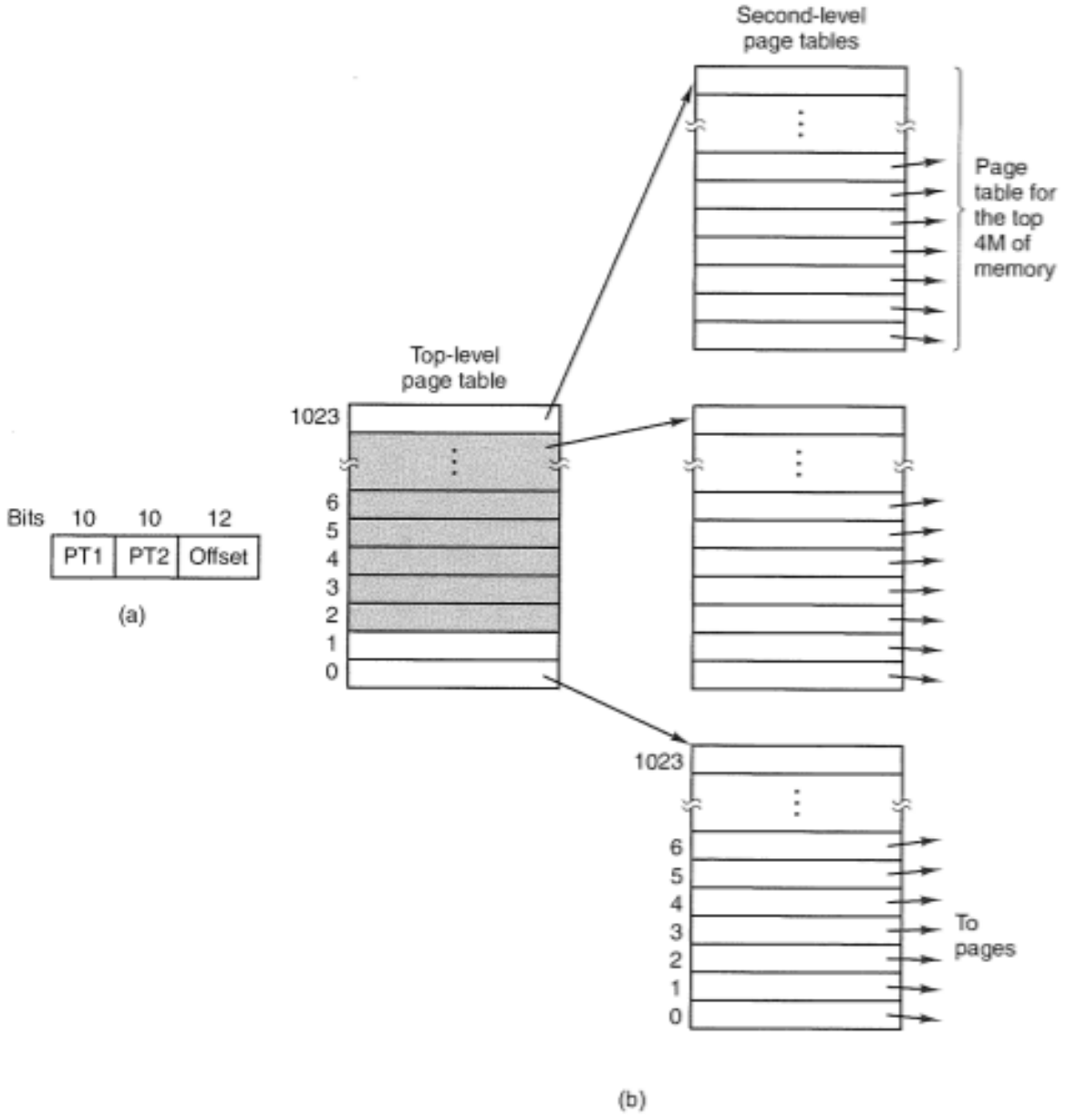
从虚拟地址到物理地址的快速映射：TLB(联想式存储)

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

放置在MMU里面。来虚拟地址，先查TLB。

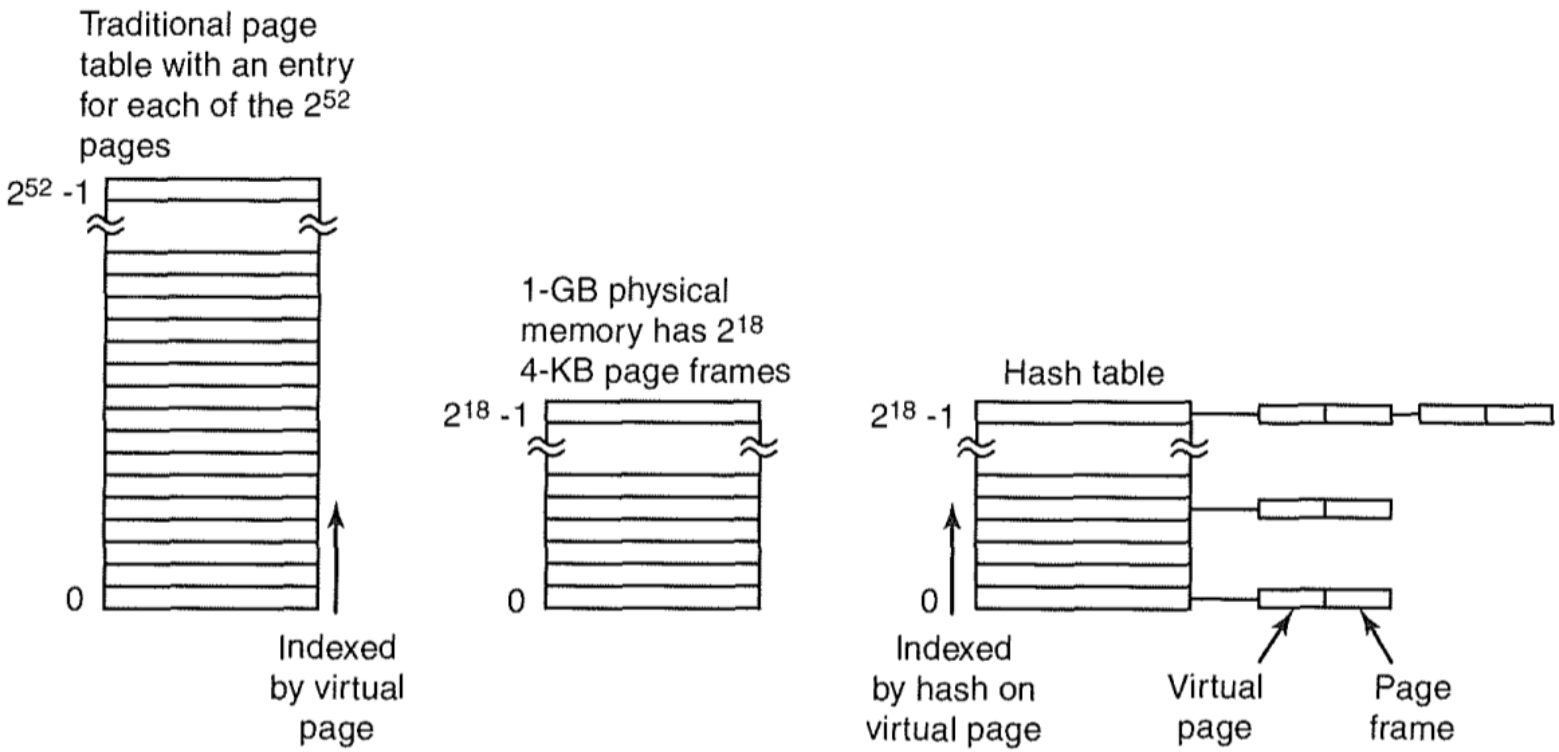
- 若能在TLB中查到该虚拟地址，则直接输出物理页框号码
- 否则，去内存中的页表中查找对应页框号码，并将其调入TLB

处理大规模地址空间的方法一：多级页表



多级页表：页表的全部内容不必都放在内存中

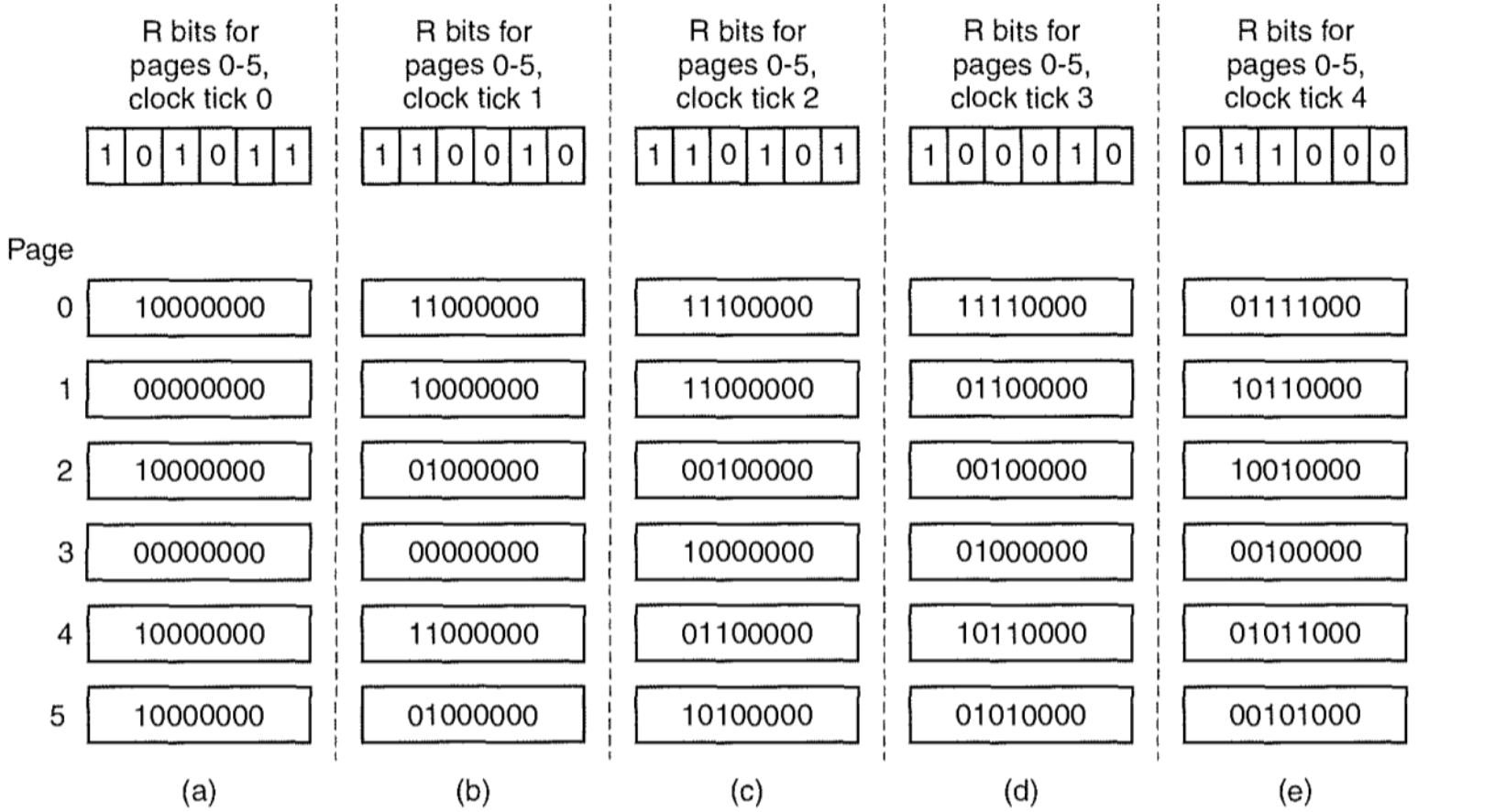
处理大规模地址空间的方法二：倒排页表



页面置换算法

当发生缺页时，OS需要选择一个页面将其从物理内存转移至硬盘，然后从硬盘调入所缺页面。

老化(aging)页面置换算法



工作集(working set)页面置换算法

- 访问的局部性：在任一时间段内，程序仅仅访问其所有页面的一小部分。
- 我们将程序在某时间段内密集访问的页面集合成为工作集

工作集(working set)页面置换算法

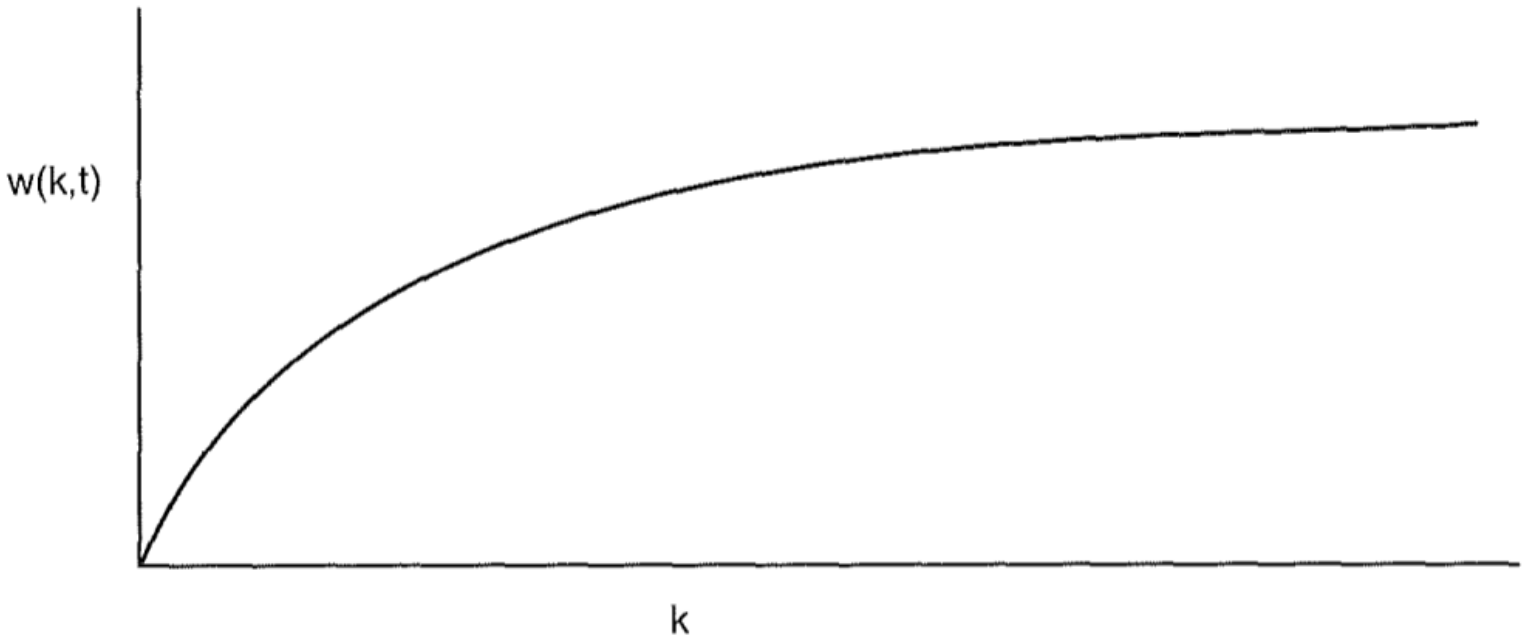
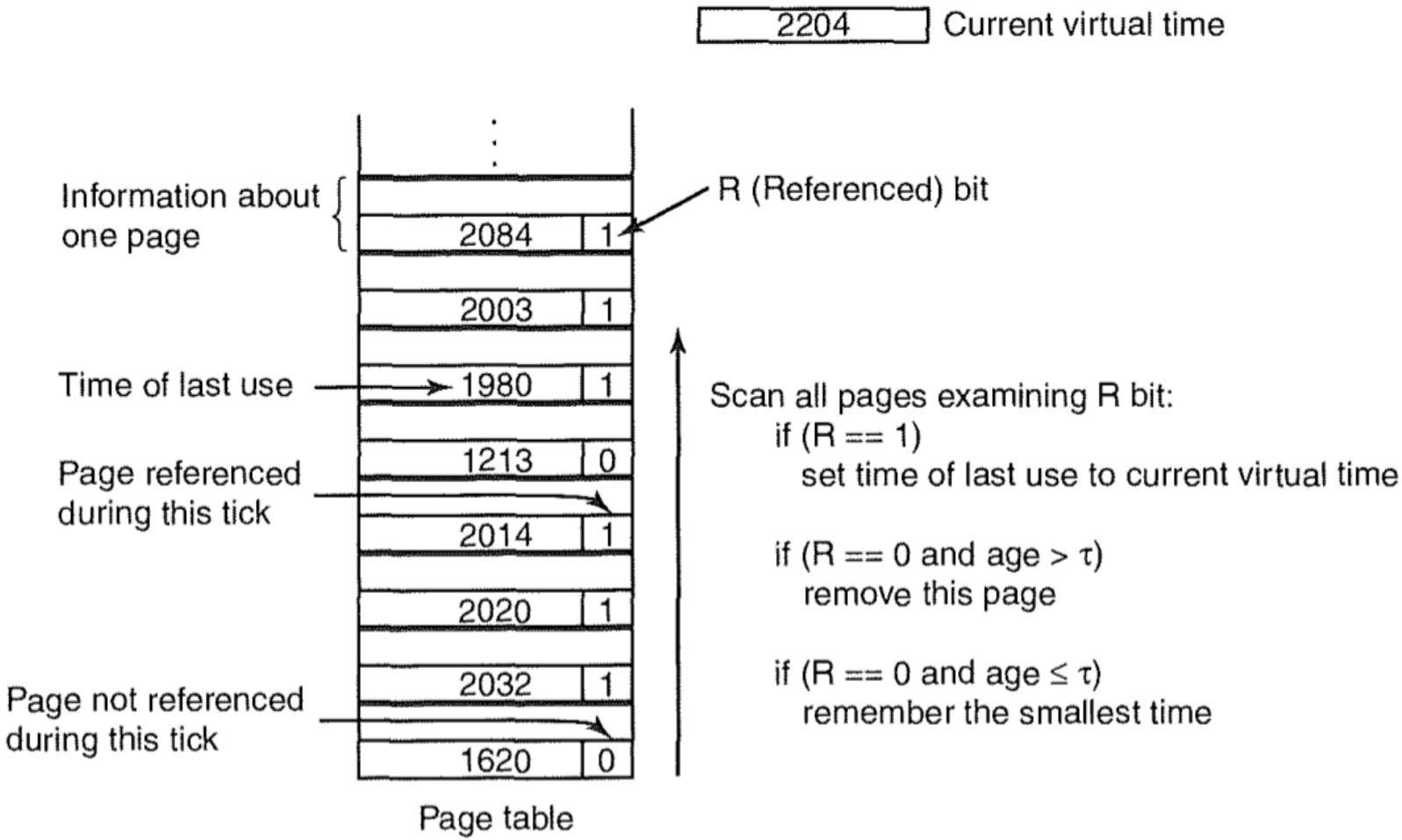


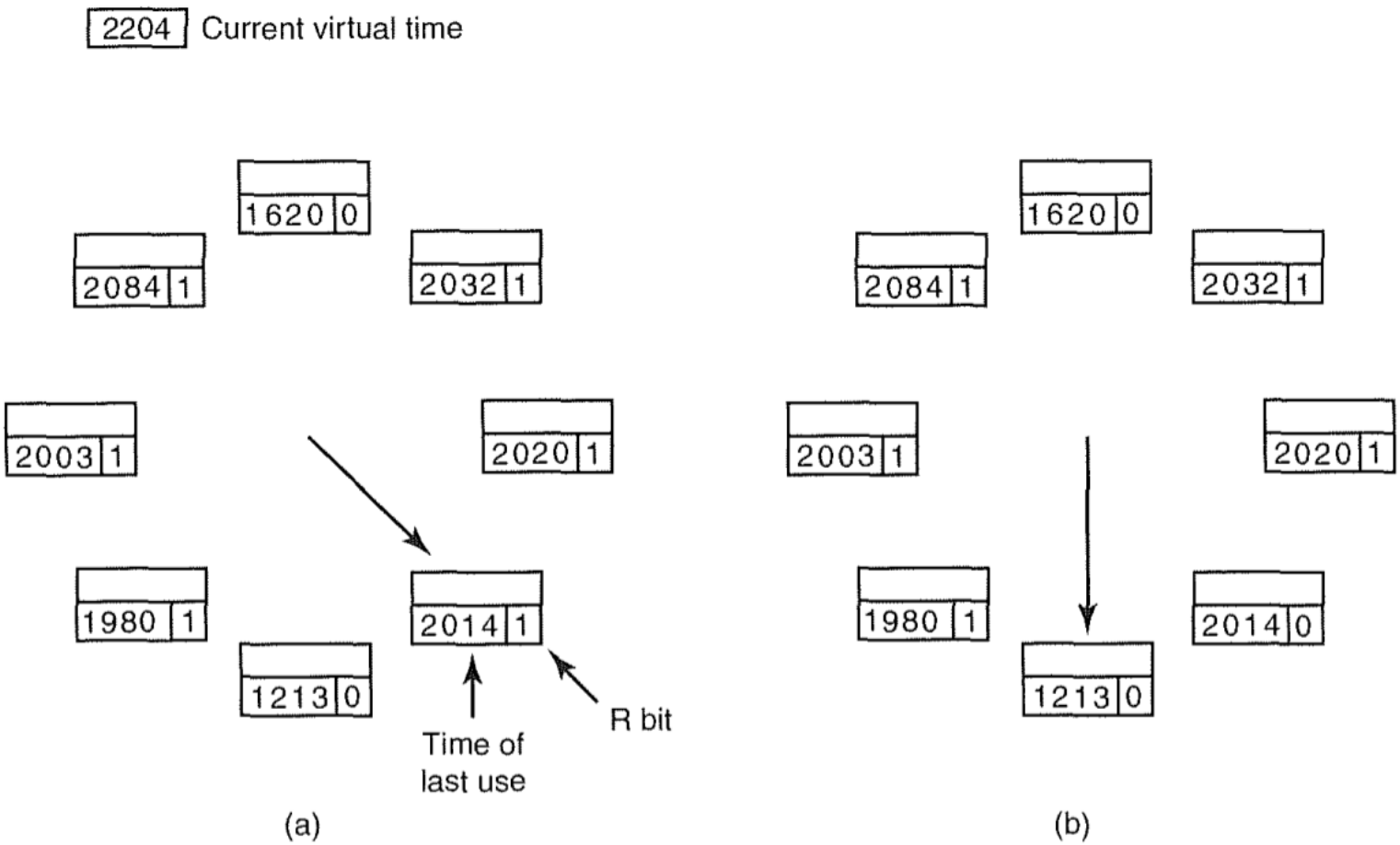
Figure 3-19. The working set is the set of pages used by the k most recent memory references. The function $w(k, t)$ is the size of the working set at time t.

工作集(working set)页面置换算法



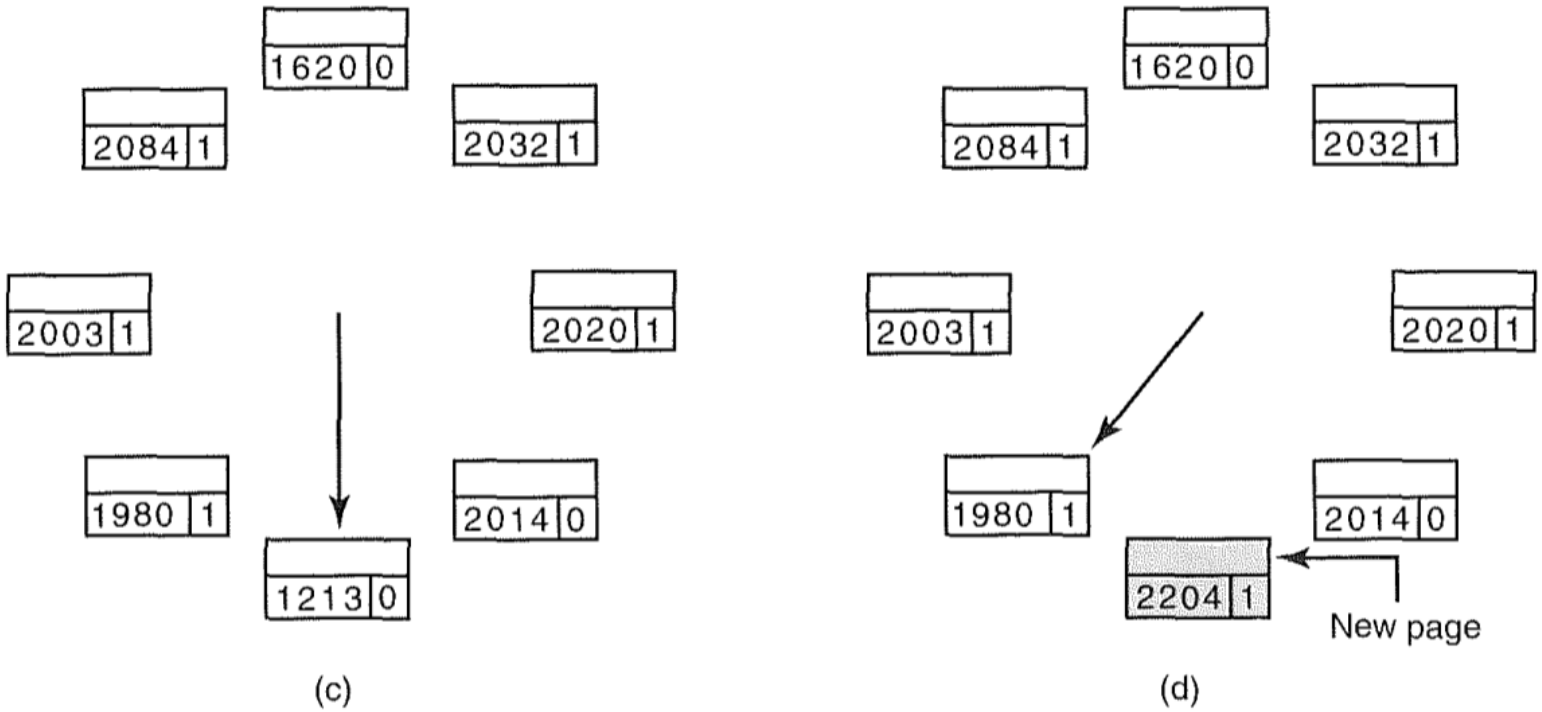
思考：如果找不到不在工作集的页面，怎么办?

WSClock页面置换算法



如果当前页R=1，将其设为0，然后考察下一页

WSClock页面置换算法



如果当前页R=0, M=0, 且其age足够大，则将其置换出物理内存

思考：如果M不为0?

分页技术：页面大小如何确定?

有以下几点考虑：

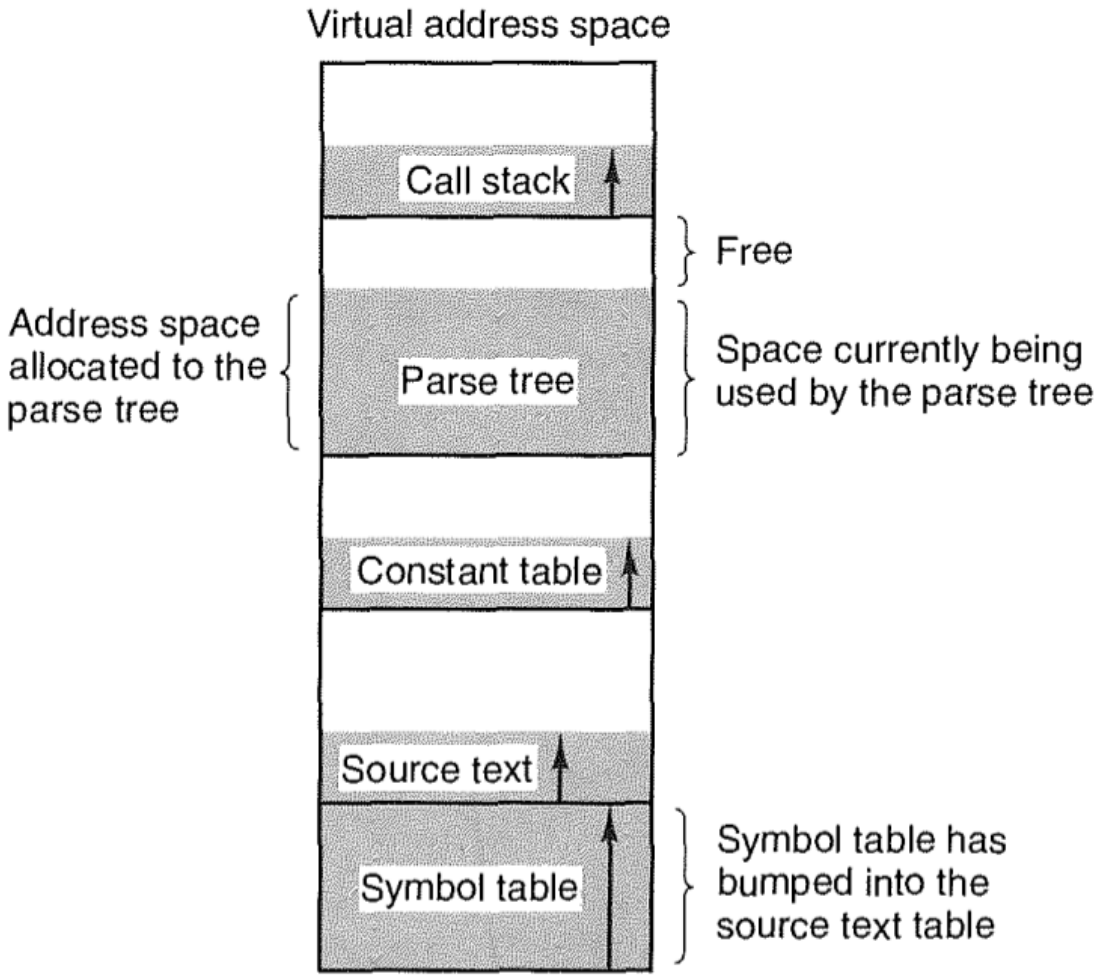
1. 页面太大? 导致最后一页浪费太多（半页）
2. 页面太小，导致页表规模很大
3. 额外开销计算

$$overhead = e \cdot \frac{s}{p} + \frac{p}{2}$$

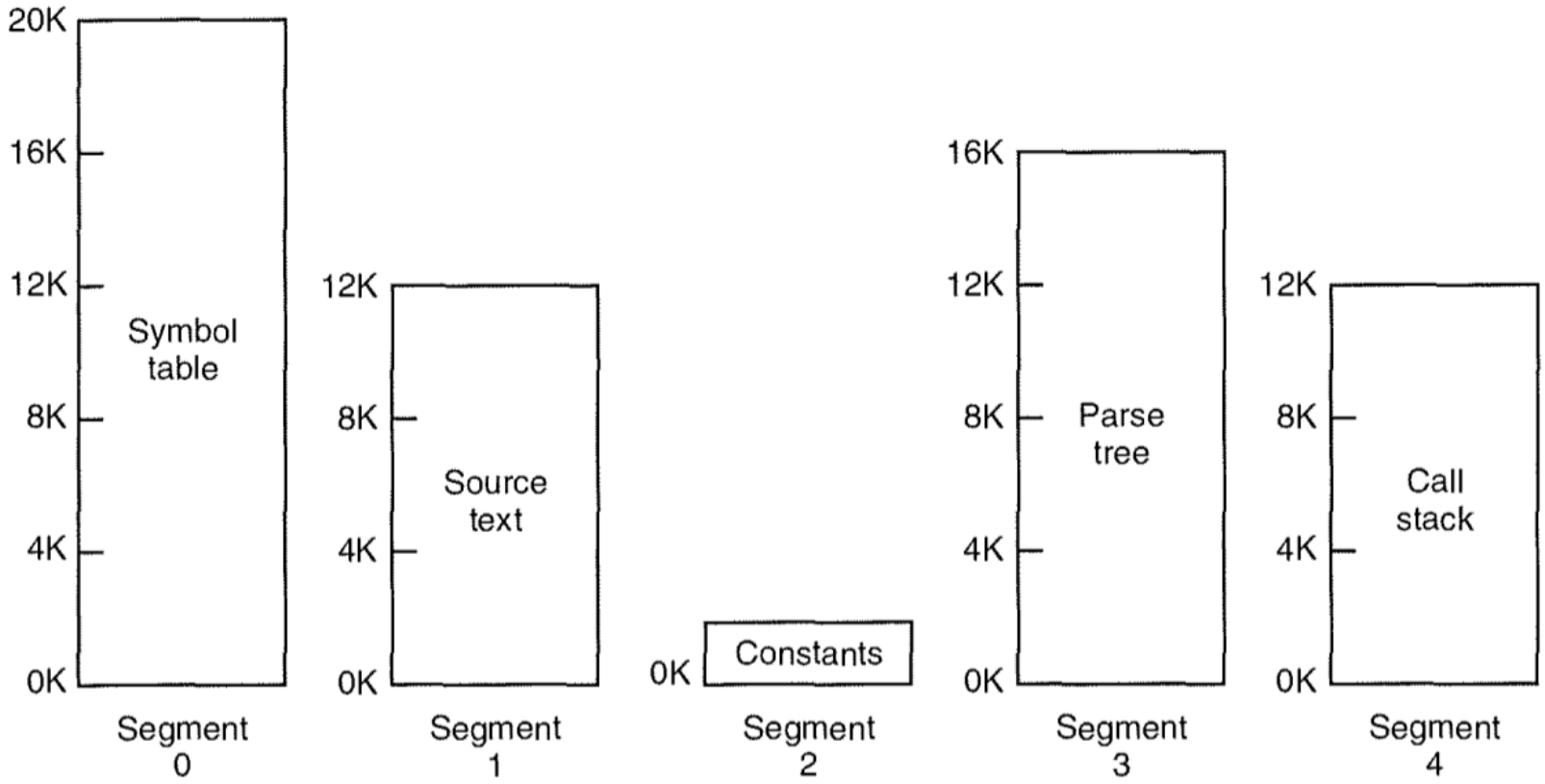
其中，p为页面大小，s为进程平均大小（字节），e为每个页表项的大小（字节）

容易算出，当$p = \sqrt{2s \cdot e}$时，额外开销最少

单纯分页技术的缺点–分段技术的引入



单纯分页技术的缺点–分段技术的引入



每个段(segment)是独立的地址空间，互不干涉，可自如伸缩

分段技术

- 程序员（编译器）可见
- 每个段可以对应子函数、栈、数组、其它类型变量中的一种(但一般不是多种)
- 分段以后，更有利于保护（代码段–执行、数据段–读写）
- 分段以后，更便于在进程间共享代码与数据

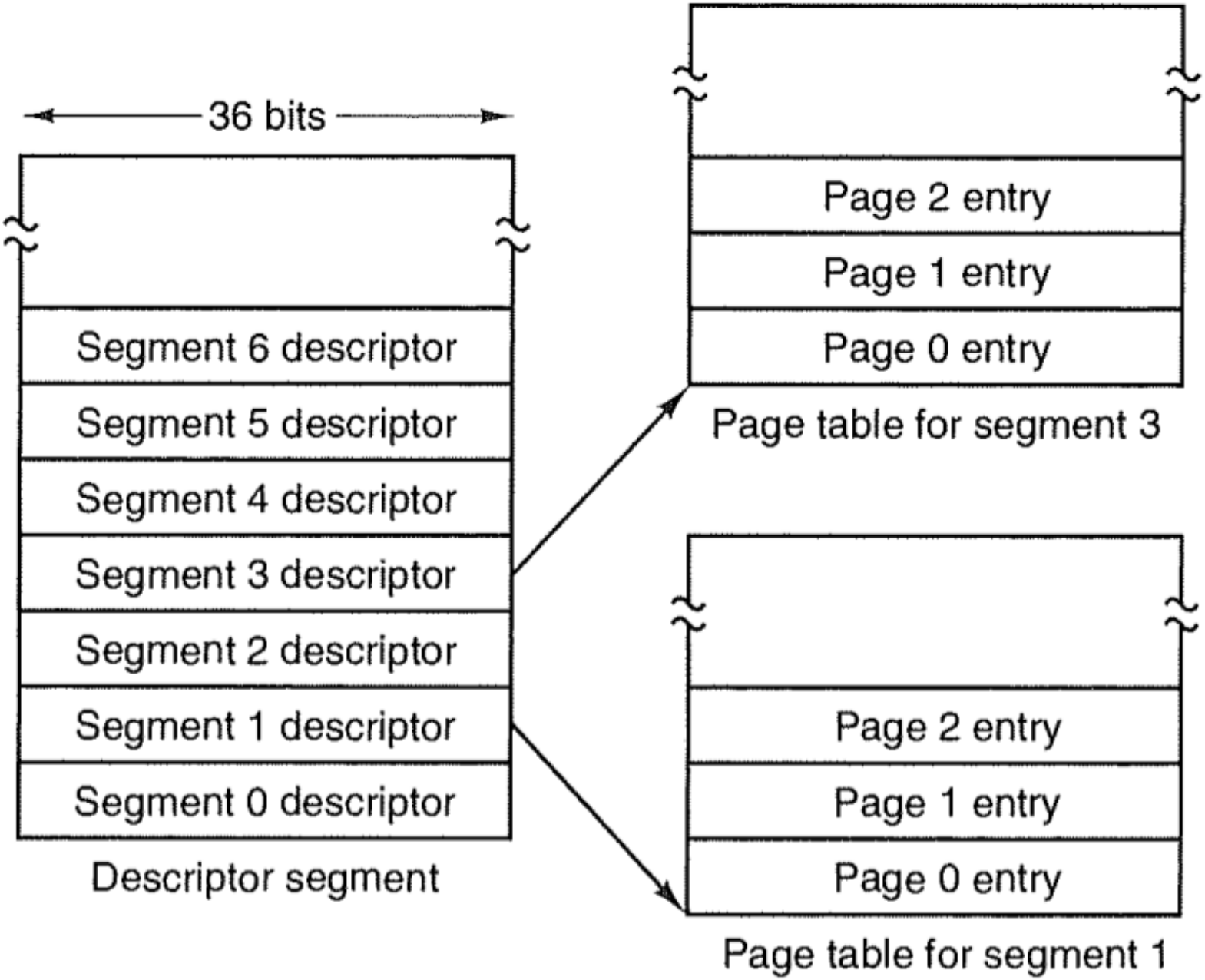
分段与分页技术的比较

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

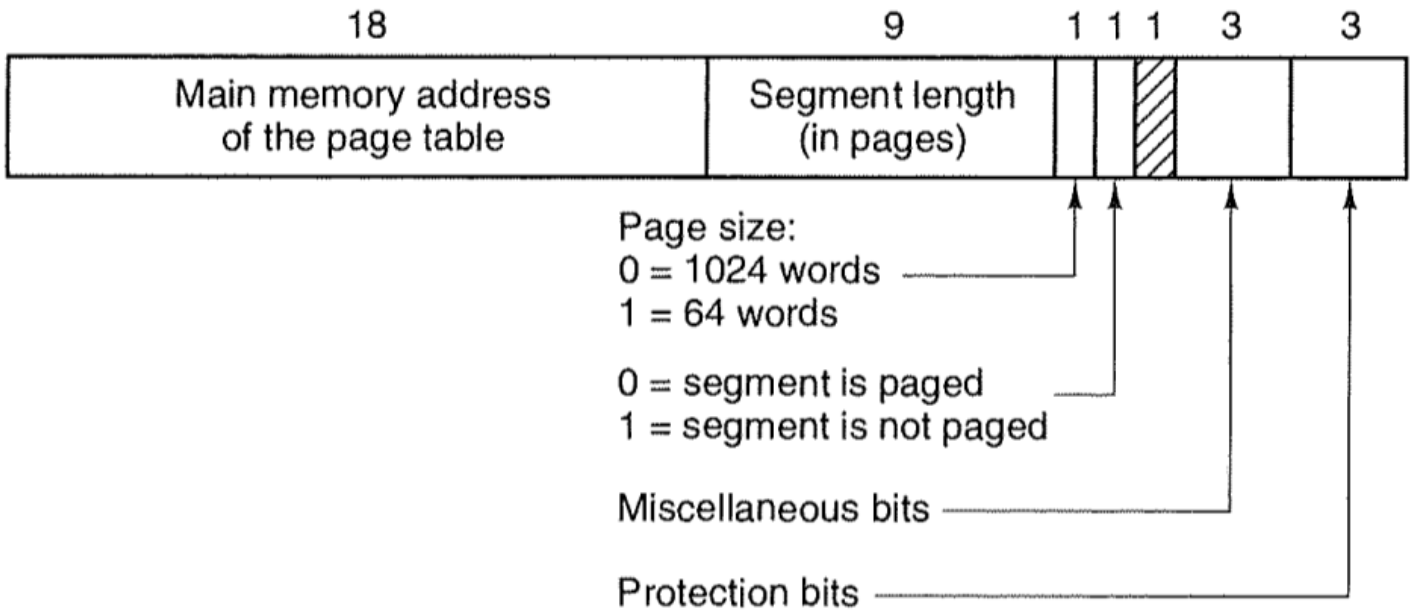
分段与分页技术相结合— 融合二者优点

如果某个段(segment)特别大，无法全放入内存，怎么办?
答案:针对每个段，采用分页技术。即只把每段中部分页面放入物理内存

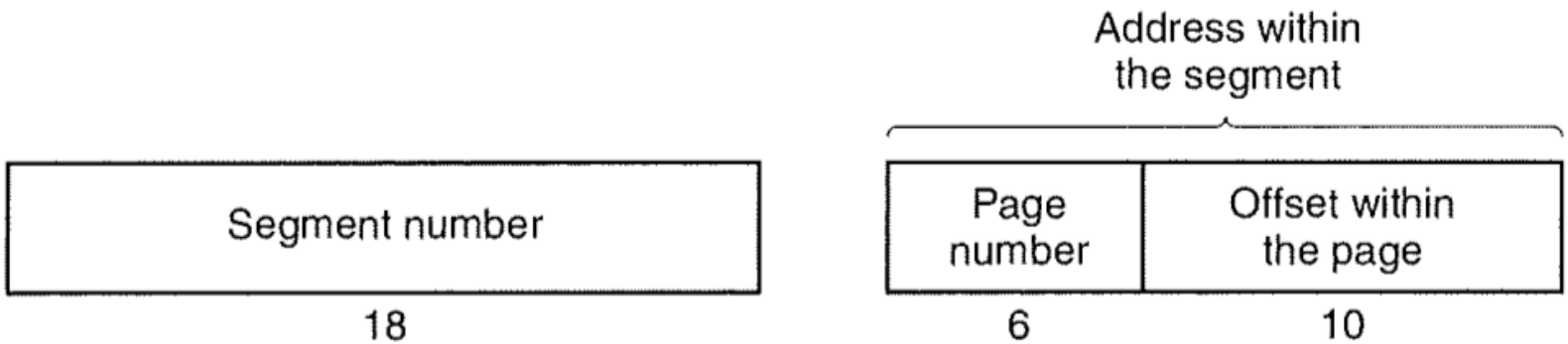
MULTICS的分段分页技术: 段表与页表



MULTICS的分段分页技术: 段表项

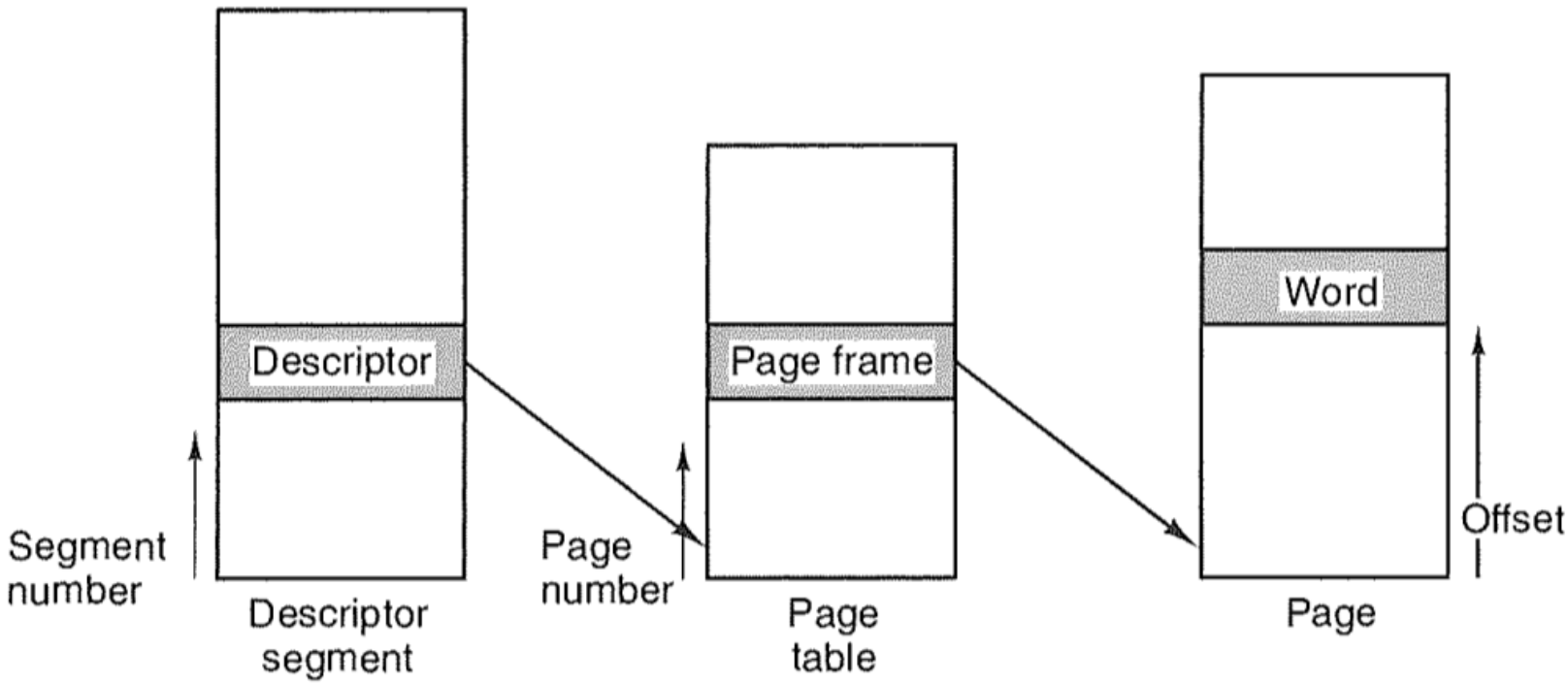


MULTICS的分段分页技术: 虚拟地址格式



思考：MULTICS可以支持多少个段?

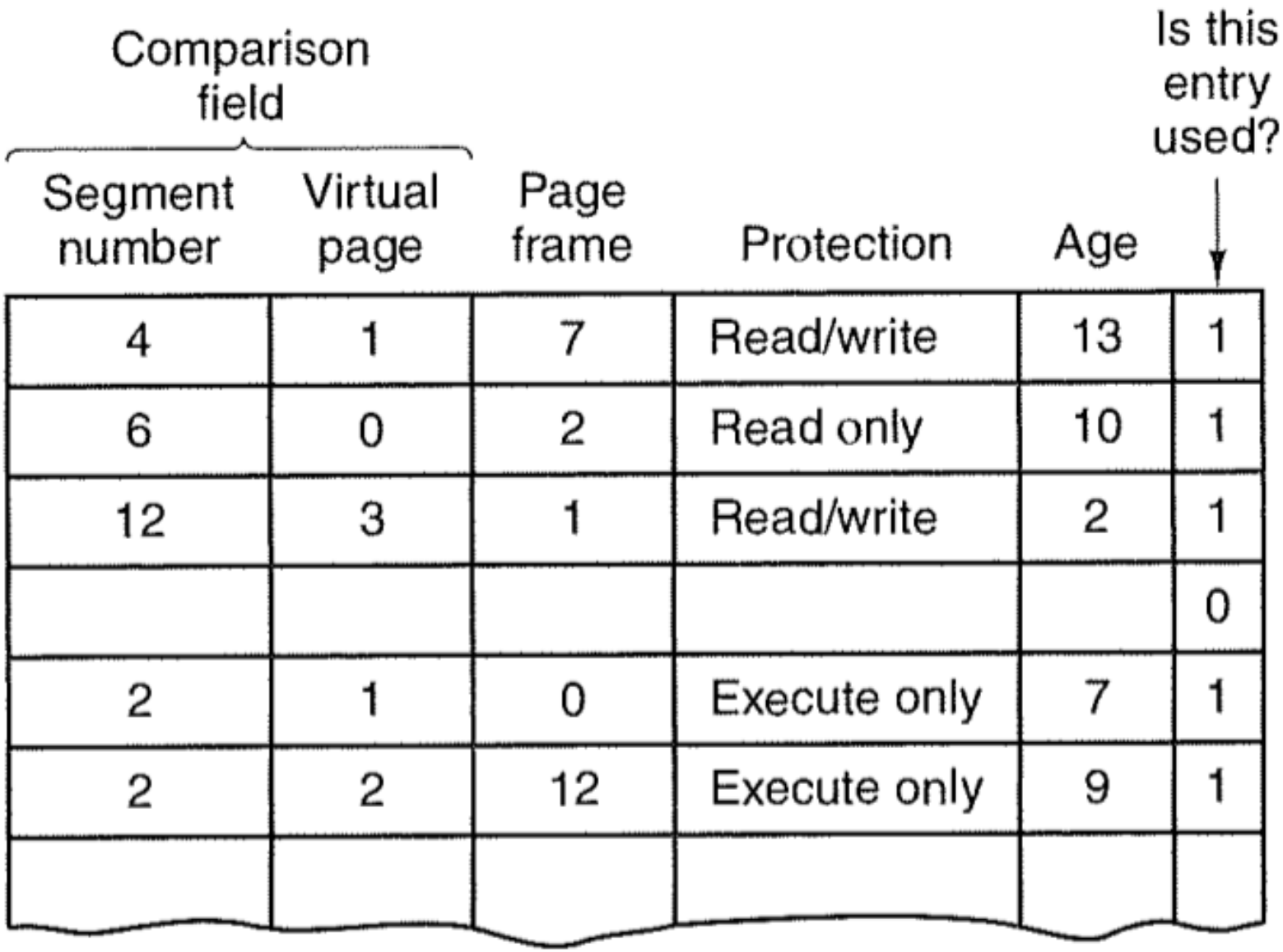
MULTICS的分段分页技术: 虚拟地址到物理地址的映射



MULTICS的分段分页技术: 虚拟地址到物理地址的映射

问题：如果每次访问内存都需要做这样的映射，则系统效率会很差，怎么办?

MULTICS的分段分页技术: TLB



Comparison field: Segment number	Comparison field: Virtual page	Page frame	Protection	Age	Is this entry used?
4	1	7	Read/write	13	1
6	0	2	Read only	10	1
12	3	1	Read/write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

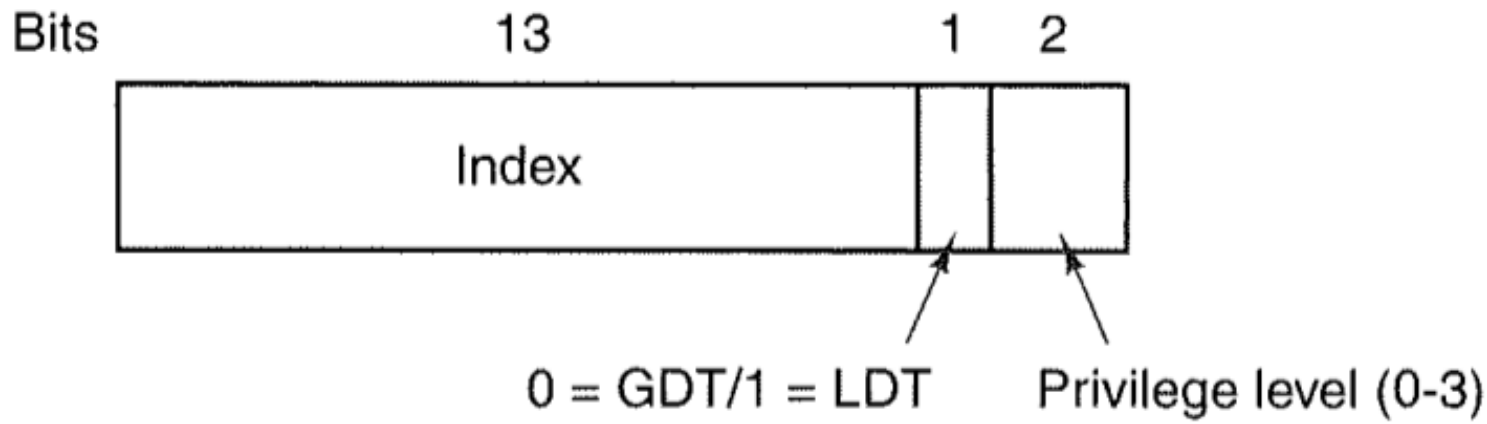
思考：最后两列的用途

Intel平台上的分段分页技术: GDT与LDT

Intel平台上，所有段表分成两类：

1. GDT (Global Descriptor Table) – 系统段(OS), 仅1个
2. LDT (Local Descriptor Table) – 每个进程都有自己的LDT

Intel平台上的分段分页技术: 段选择符(Segment Selector)

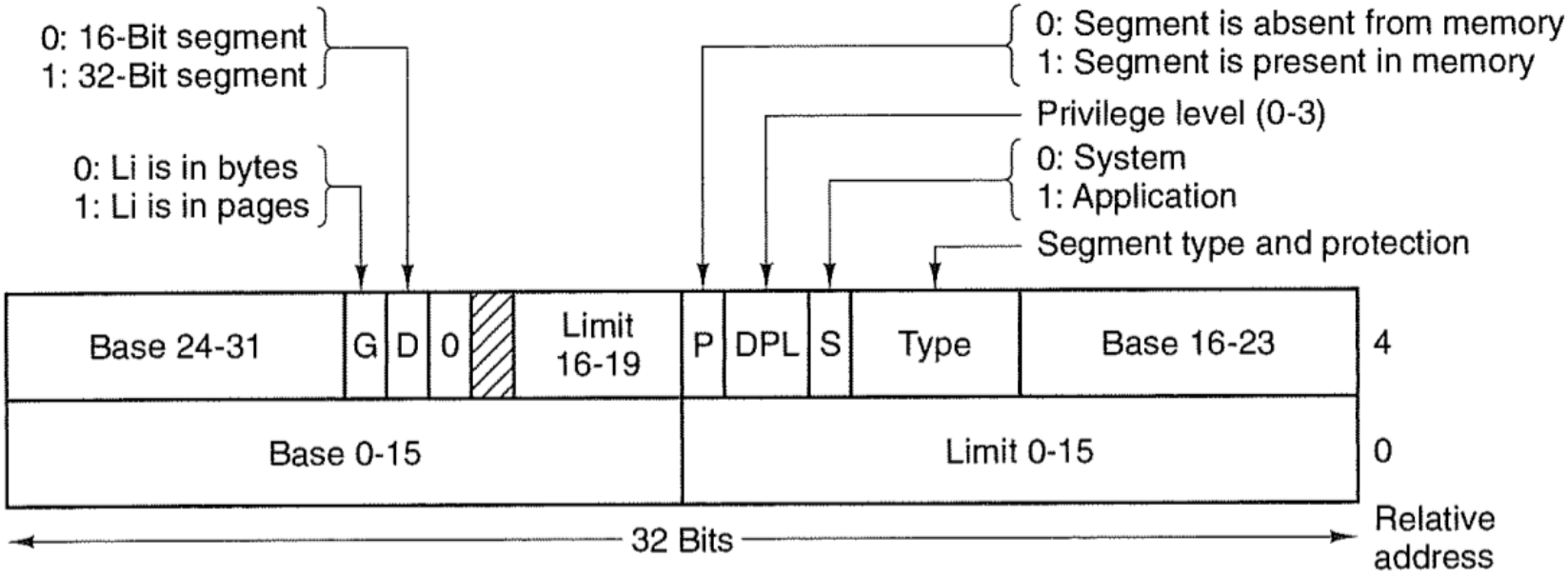


CS寄存器 存放代码段的段选择符

DS寄存器 存放数据段的段选择符

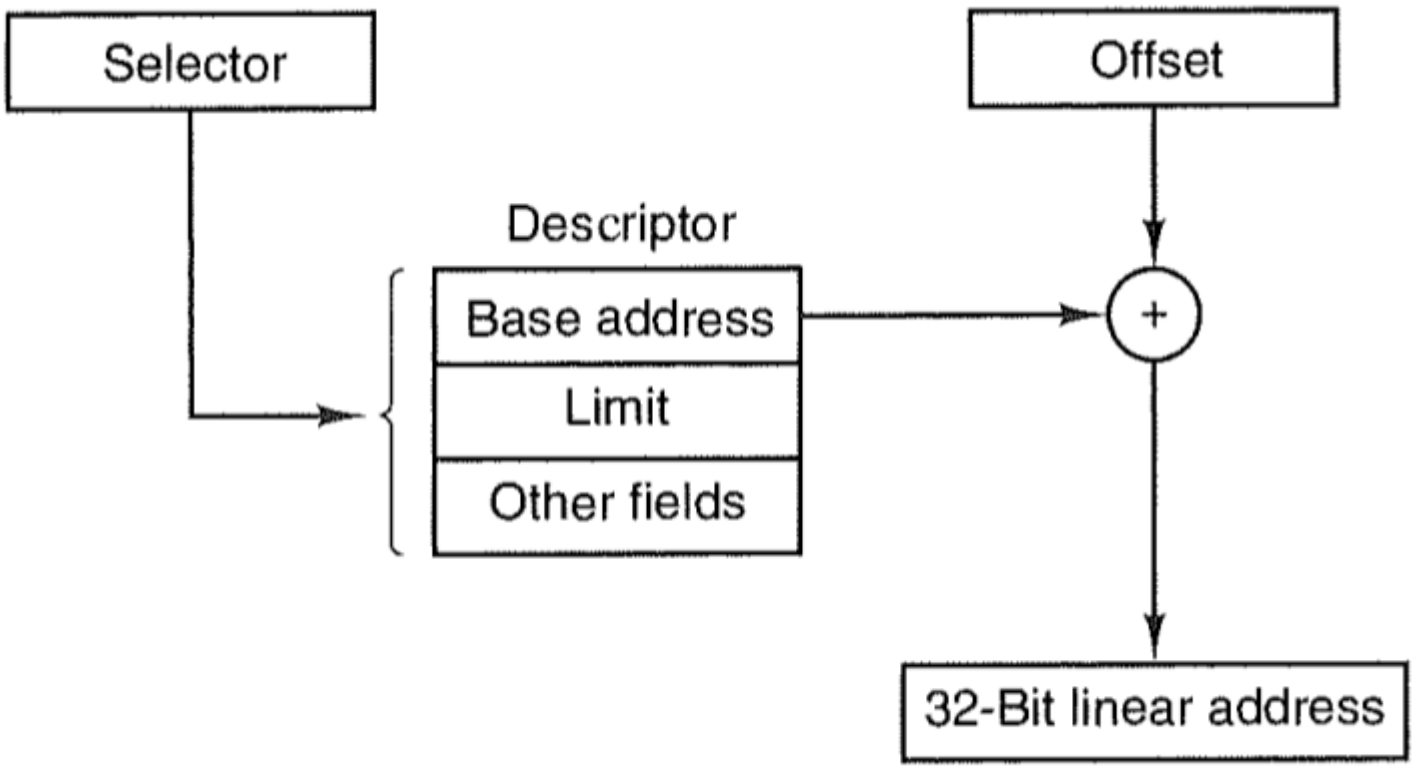
其他 四个段寄存器

Intel平台上的分段分页技术: 段描述符(Segment Descriptor) – 段表项



三个关键字段：BASE，LIMIT，G

Intel平台上的分段分页技术: 从（段选择符，偏移）到线性地址的映射



Intel平台上的分段分页技术: 从线性地址到物理地址的映射