第四章　存储管理

翁楚良

https://chuliangweng.github.io

2023 春　ECNU

虚拟存储管理

- 分页技术
- 页表
- 关联存储器TLB
- 反置页表

反置页表

- 物理存储器的每个页框对应一个页表项，而不是虚地址空间中的每个虚页对应一个页表项。
- 例：在具有64位虚地址、4K大小页面、32M RAM的系统上，一个逆向页表只需要8192个表项，每个表项记录了相应的物理页面中所保存的是哪一个进程的哪一个虚拟页面
- 优点：节省大量为保存页表所需要内存空间
- 缺点：查找过程复杂
  - 当进程n引用虚页p时，硬件不能再用p作为索引查页表得到物理地址，取而代之的是在整个逆向页表中查找表项(n,p)

第四章 提纲

- 4.1 基本的内存管理
- 4.2 交换技术
- 4.3 虚拟存储管理
- 4.4 页面替换算法
- 4.5 页式存储管理的设计问题
- 4.6 段式存储管理
- 4.7 MINIX3进程管理器概述
- 4.8 MINIX3进程管理器实现

动机

- 当发生缺页中断时，需从磁盘上调入相应的页面，然而内存已满，需要选取内存中的页面，将其换出，并装入新页面。
  - 被换出的页面已被修改，需写回
  - 被换出的页面未修改，直接抛弃
- 如何从众多的页面中选取被置换的页面？
  - 为此提出了各种算法
  - 可以应于缓存块的置换、Web缓冲区的更新等

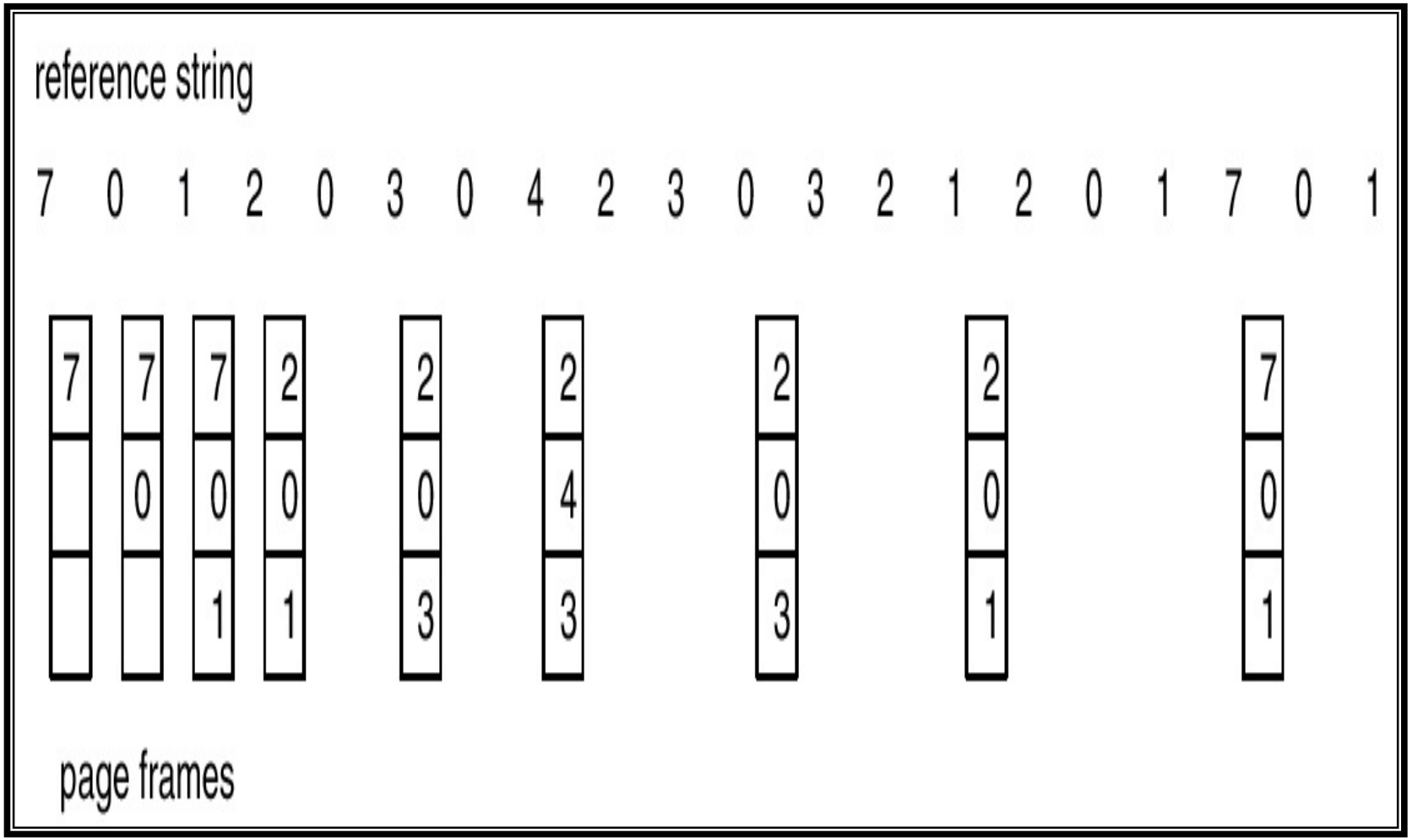
页面替换算法

- 最优页面置换算法
- 未最近使用页面置换算法
- 先进先出页面置换算法
- 第二次机会页面置换算法
- 时钟页面置换算法
- 最近最久未使用页面置换算法

The Optimal Page Replacement Algorithm

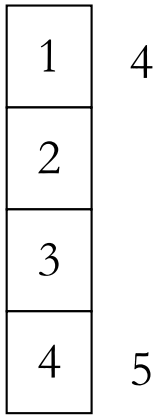
- The optimal policy selects for replacement the page for which the time to the next reference is the longest:
  - produces the fewest number of page faults.
  - impossible to implement (need to know the future) but serves as a standard to compare with the other algorithms we shall study.

The Optimal Page Replacement Algorithm



The Optimal Page Replacement Algorithm

- 4 frames example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

- How do you know this? You don't!
- Used for measuring how well your algorithm performs.

页面替换算法

- 最优页面置换算法
- 未最近使用页面置换算法
- 先进先出页面置换算法
- 第二次机会页面置换算法
- 时钟页面置换算法
- 最近最久未使用页面置换算法

The Not Recently Used Page Replacement Algorithm

- R is set whenever the page is referenced (read or written). M is set when the page is written to (i.e., modified).
- When a page fault occurs, the operating system inspects all the pages and divides them into four categories based on the current values of their R and M bits:
  - Class 0: not referenced, not modified.
  - Class 1: not referenced, modified.
  - Class 2: referenced, not modified.
  - Class 3: referenced, modified.
- The NRU (Not Recently Used) algorithm removes a page at random from the lowest numbered nonempty class

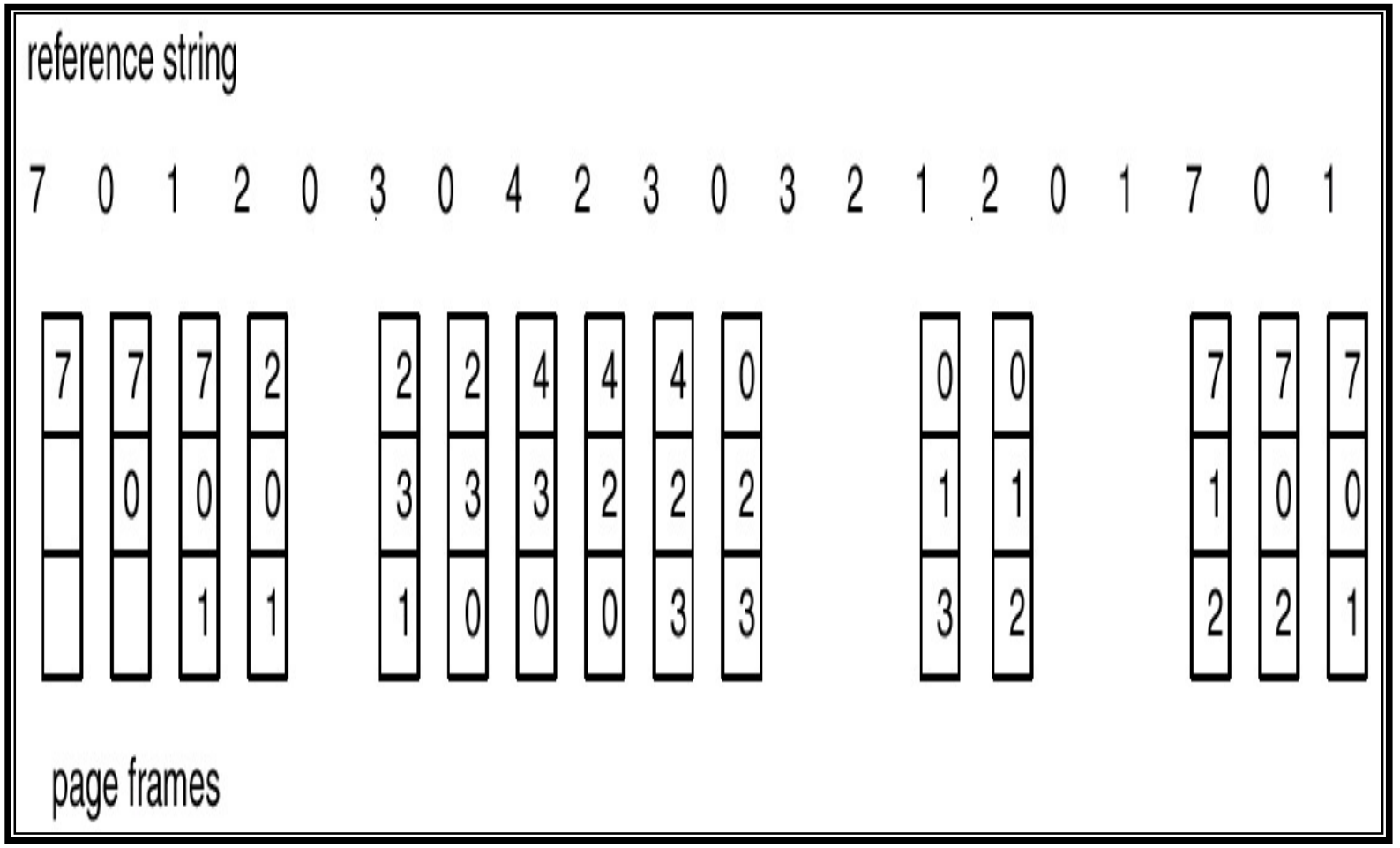
页面替换算法

- 最优页面置换算法
- 未最近使用页面置换算法
- 先进先出页面置换算法
- 第二次机会页面置换算法
- 时钟页面置换算法
- 最近最久未使用页面置换算法

FIFO Page Replacement Algorithm

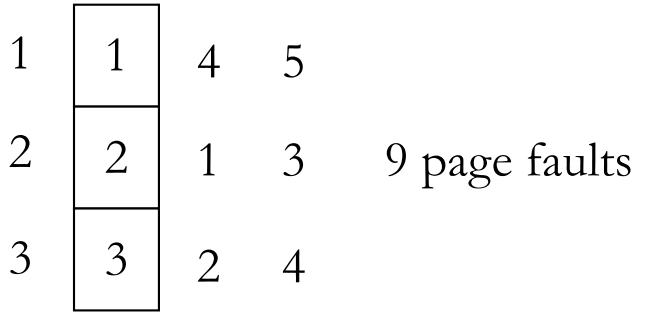
- Treats page frames allocated to a process as a circular buffer:
  - When the buffer is full, the oldest page is replaced. Hence first-in, first-out:
    - A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO.
  - Simple to implement:
    - requires only a pointer that circles through the page frames of the process.

FIFO Page Replacement Algorithm

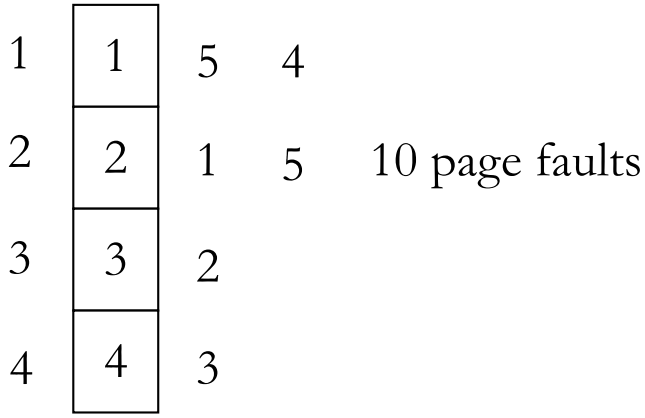


FIFO Page Replacement Algorithm

- 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

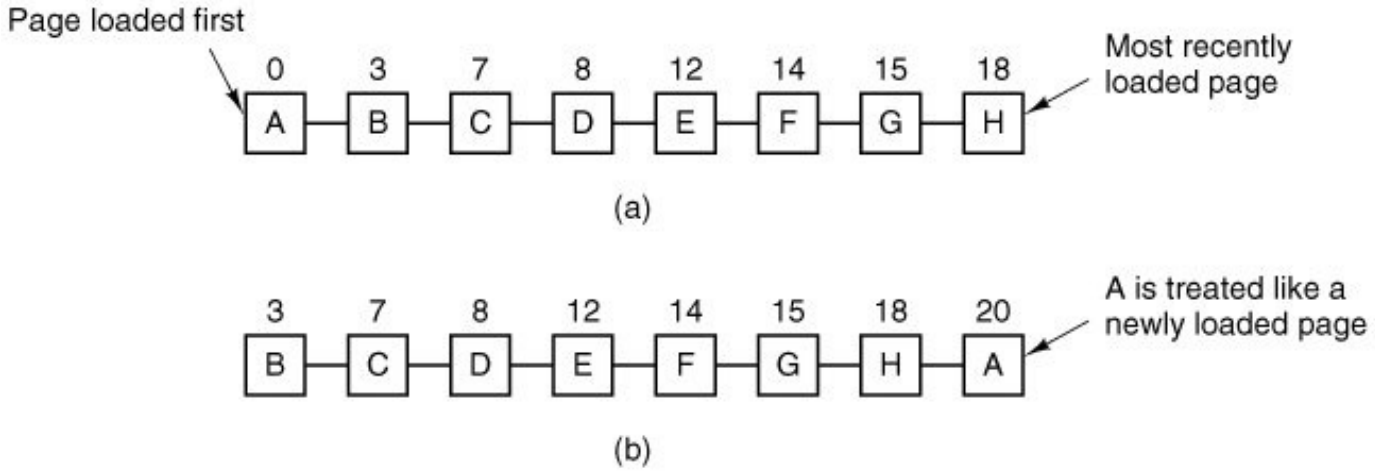


页面替换算法

- 最优页面置换算法
- 未最近使用页面置换算法
- 先进先出页面置换算法
- 第二次机会页面置换算法
- 时钟页面置换算法
- 最近最久未使用页面置换算法

The Second Chance Page Replacement Algorithm

- A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest page.
- If it is 0, the page is both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it had just arrived in memory.



页面替换算法

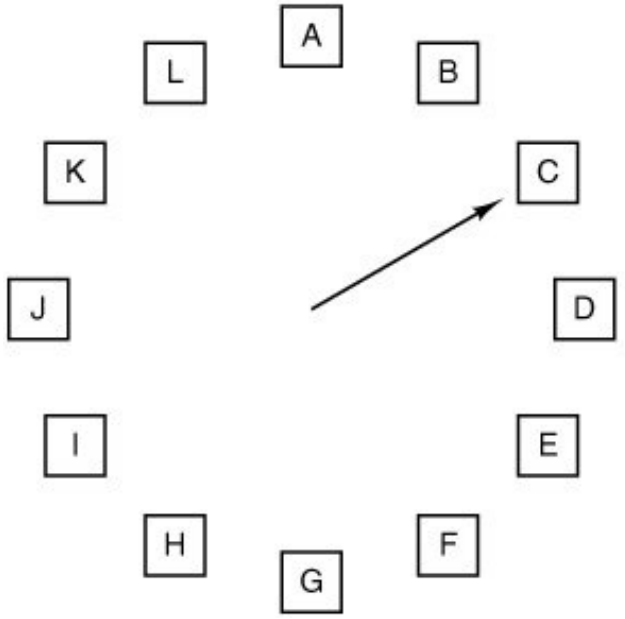
- 最优页面置换算法
- 未最近使用页面置换算法
- 先进先出页面置换算法
- 第二次机会页面置换算法
- 时钟页面置换算法
- 最近最久未使用页面置换算法

The Clock Page Replacement Algorithm

- The set of frames candidate for replacement is considered as a circular buffer.
- When a page fault occurs, the page being pointed to by the hand is inspected.

If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position.

If R is 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0.



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

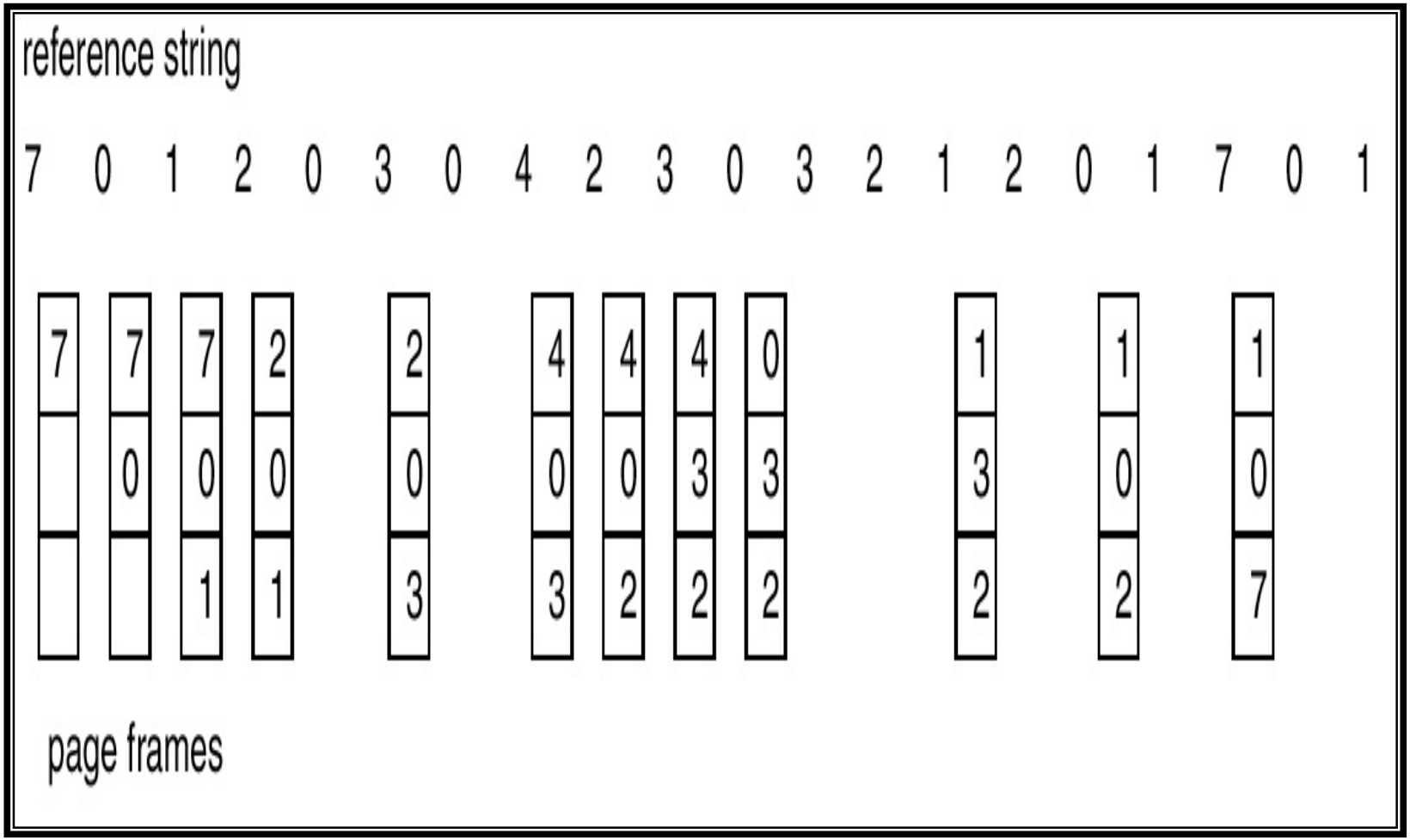
页面替换算法

- 最优页面置换算法
- 未最近使用页面置换算法
- 先进先出页面置换算法
- 第二次机会页面置换算法
- 时钟页面置换算法
- 最近最久未使用页面置换算法

The Least Recently Used (LRU) Page Replacement Algorithm

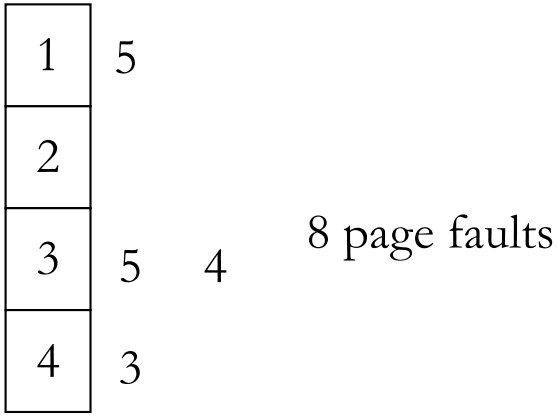
- Replaces the page that has not been referenced for the longest time:
  - By the principle of locality, this should be the page least likely to be referenced in the near future.
  - performs nearly as well as the optimal policy.

LRU Page Replacement



LRU Page Replacement

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

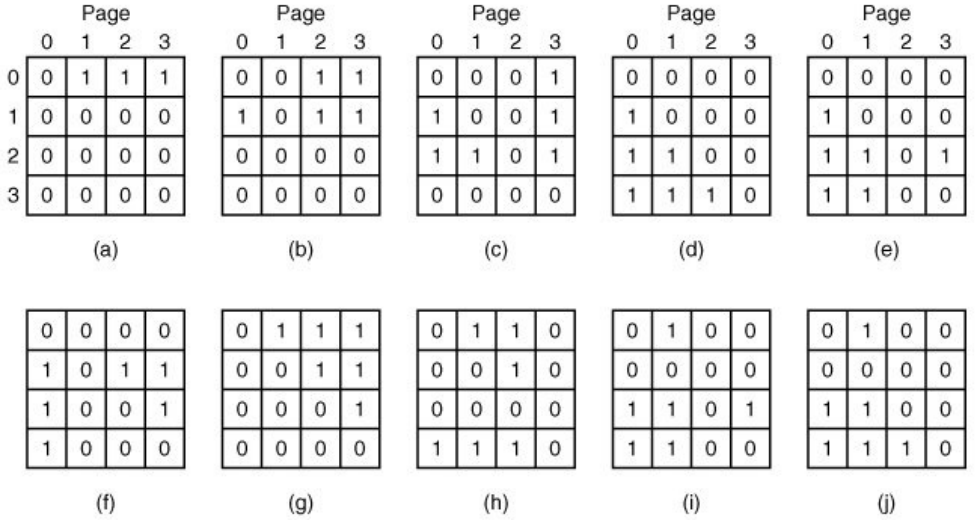


LRU Implementations

- Counter implementation:
  - Every page entry has a counter; every time a page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be changed, look at the counters to determine which are to change.

- Matrix method:
  - For a machine with n page frames, the LRU hardware can maintain a matrix of n × n bits, initially all zero.
  - Whenever page frame k is referenced, the hardware first sets all the bits of row k to 1, then sets all the bits of column k to 0
  - At any instant, the row whose binary value is lowest is the least recently used

pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.



LRU Approximation methods

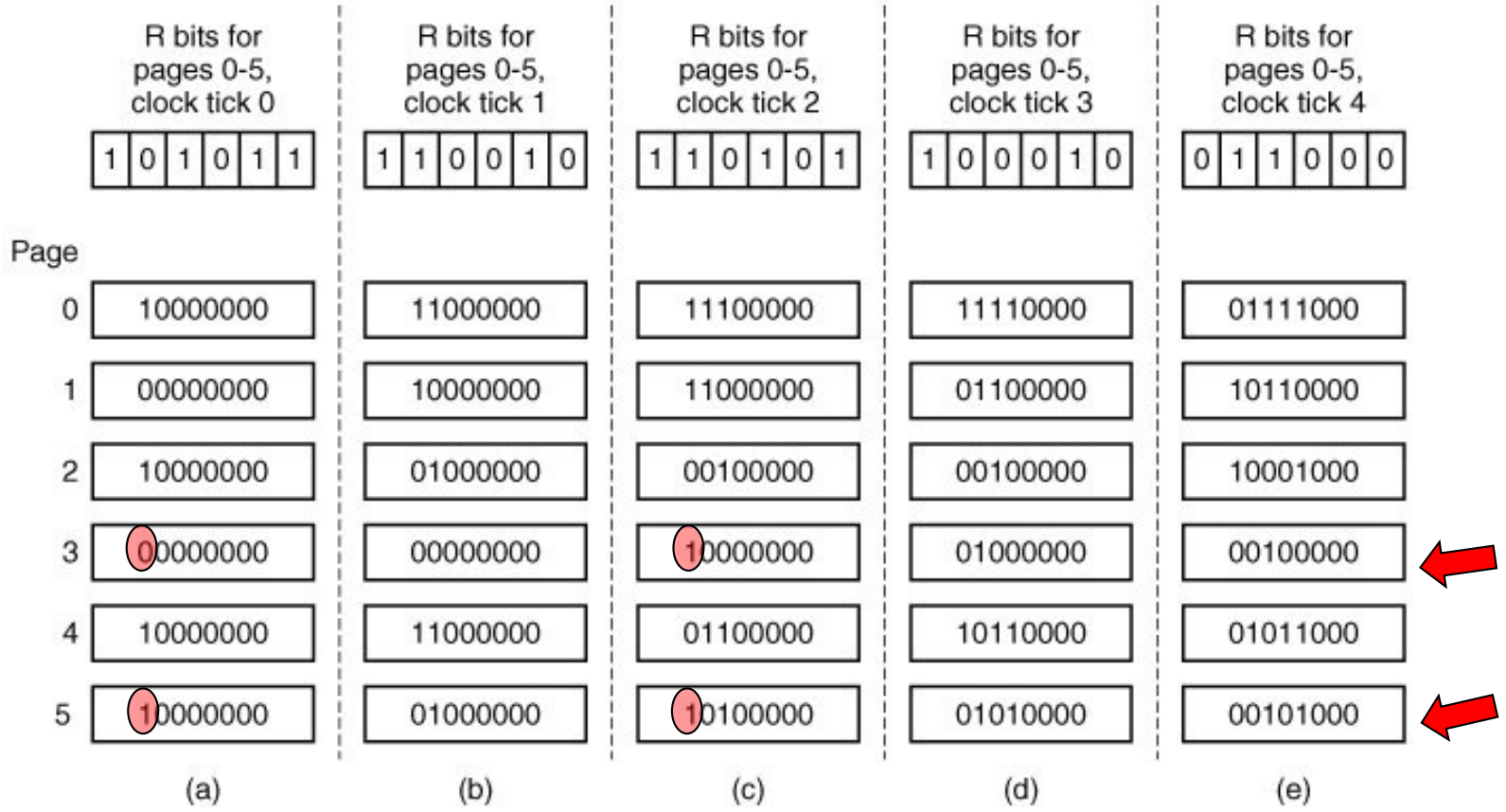
- NFU (Not Frequently Used) algorithm
  - It requires a software counter associated with each page, initially zero.
  - At each clock interrupt, the operating system scans all the pages in memory. For each page, the R bit, which is 0 or 1, is added to the counter.
  - When a page fault occurs, the page with the lowest counter is chosen for replacement.

LRU Approximation methods

- aging algorithm
  - It is a small modification to NFU.
  - The modification has two parts.
    - First, the counters are each shifted right 1 bit before the R bit is added in.
    - Second, the R bit is added to the leftmost, rather than the rightmost bit.
  - When a page fault occurs, the page whose counter is the lowest is removed

LRU Approximation methods

- aging algorithm



第四章 提纲

- 4.1 基本的内存管理
- 4.2 交换技术
- 4.3 虚拟存储管理
- 4.4 页面替换算法
- 4.5 页式存储管理的设计问题
- 4.6 段式存储管理
- 4.7 MINIX3进程管理器概述
- 4.8 MINIX3进程管理器实现

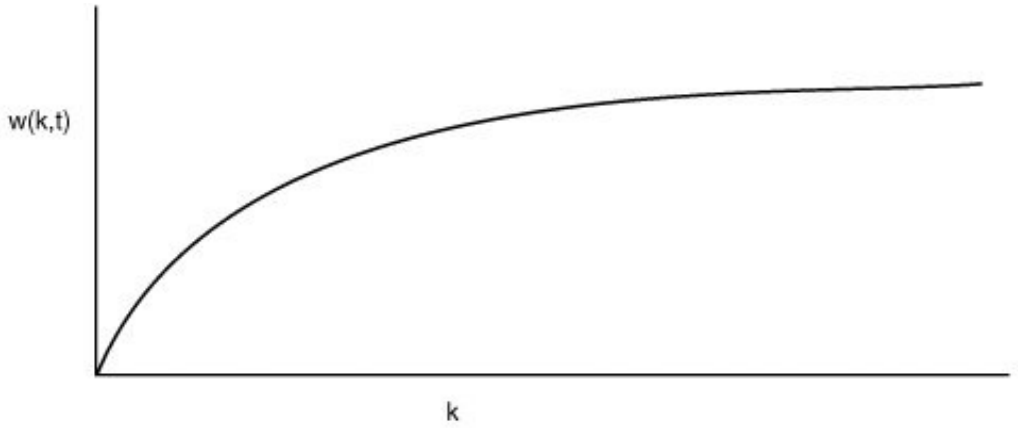
页式存储管理的设计问题

- 工作集模型
- 分配策略
- 页面大小
- 虚拟存储接口

访存的局部性与工作集

- 访存的局部性：在进程运行的任何阶段，它都只访问它的页面中较小的一部分
- 工作集：一个进程当前使用的页的集合
- 抖动：分配给进程的物理页面数太小，无法包含其工作集，频繁地在内存和外存间换页
- 工作集模型：页式存储管理系统跟踪进程的工作集，并保证在进程运行以前它的工作集就已经在内存中了。在进程运行之前预先装入页面也叫做预先调页。

w=w(t,k)
t: 时间
k: 访问次数
w: 当前时刻t之前的k次访问中所涉及到的页面的集合

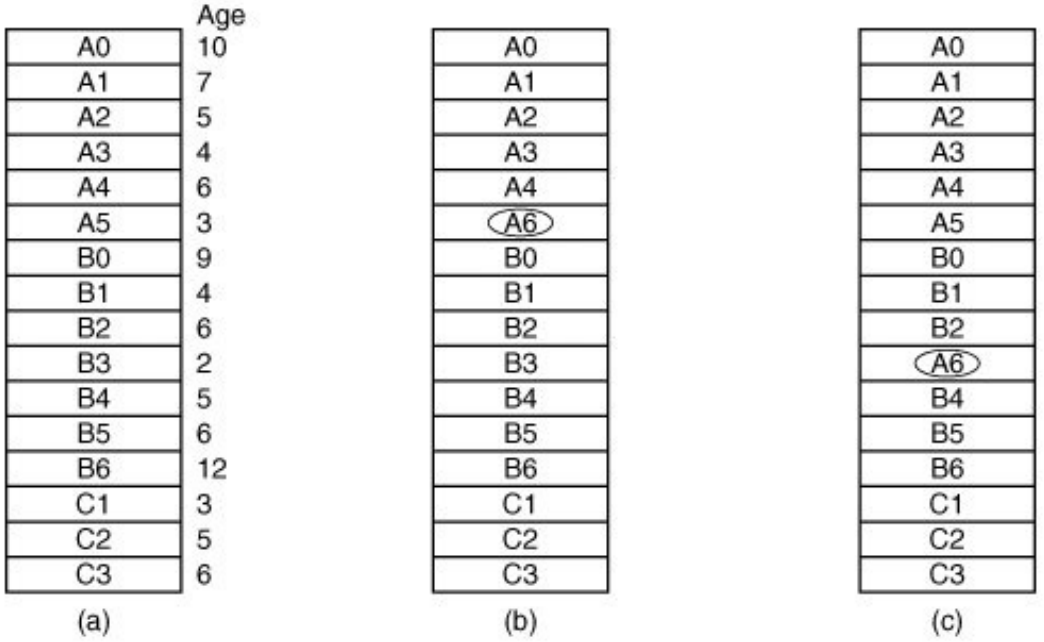


页式存储管理的设计问题

- 工作集模型
- 分配策略
- 页面大小
- 虚拟存储接口

分配策略

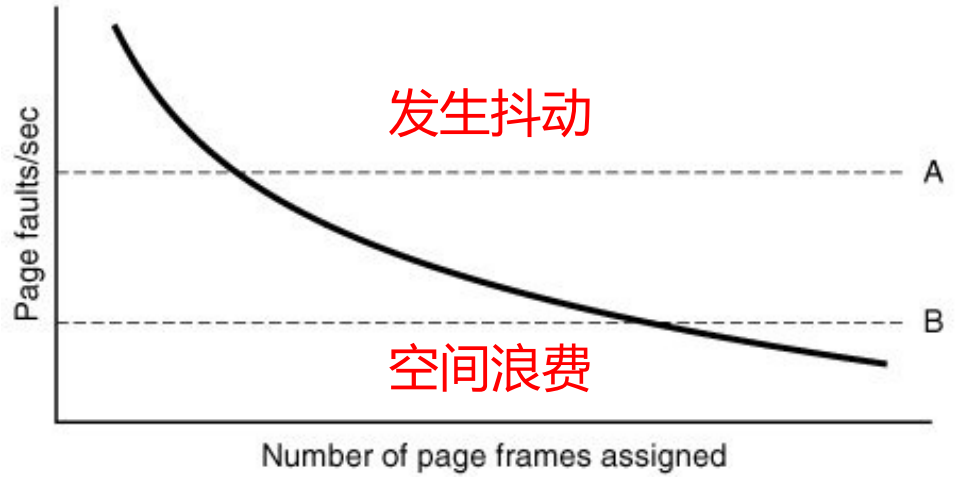
- 页面置换算法作用的范围不同，对应不同的分配策略
  - 局部页面置换算法：在进程所分配的页面范围内选取将被置换的页面
    - 每个进程分配固定大小的内存空间
  - 全局页面置换算法：在内存中所有的页面范围内选取被置换的页面
    - 所有进程动态共享系统的物理页面，分配给每个进程的页面数动态变化的



物理页面的分配

- 阻止抖动、避免内存空间的浪费
- 缺页率算法
  - 缺页率：一秒钟内出现的缺页数(统计)
  - 定义上下界，使得进程的缺页率在其范围之内

负载控制：系统内运行的进程过多，无法使所有进程的缺页率都低于A，则需要将一些进程换出至外存(交换)



页式存储管理的设计问题

- 工作集模型
- 分配策略
- 页面大小
- 虚拟存储接口

页面大小

- 页面大小需要权衡多方互相矛盾的因素
  - 内碎片、空间利用率
    - 从统计的规律看，内碎片的大小一般是半个页面；页面越小、内碎片也会越小
    - 例，内存被分配n段，页面大小为p, 则总内碎片大小为$np/2$
  - 页表项数、页表装入时间
    - 同一程序，页面大小越小，需要的页表数会越多
    - 传送不同大小页面所花的时间相差不多，同一程序，页面越小、页面数会越多、故时间会越长

页面大小

- 理论分析
  - 假设平均进程大小是s个字节，页面大小是p个字节，每个页表项需要e个字节，那么进程需要的页数大约是s/p，占用了se/p个字节的页表空间，由于内碎片在最后一页浪费的内存是p/2。因此，由页表和内碎片损失造成的全部开销是：开销 = se/p + p/2
  - 最优值一定在中间某个地方，通过对p求导并令其等于零，得到方程：-se/p2 + 1/2 = 0
  - 从这个方程得出最优页面大小的公式（只考虑碎片浪费和页表所需的内存）：$p = \sqrt{2se}$

页式存储管理的设计问题

- 工作集模型
- 分配策略
- 页面大小
- 虚拟存储接口

虚拟存储接口

- 通常虚拟存储器对进程和程序员是透明的，即所能看到的全部是在一个带有较小的物理存储器之上一个大的虚地址空间
- 允许程序员对内存映射进行某些控制，可以实现两个或多个进程共享同一段内存空间，即页面共享
- 页面共享可以用来实现高性能的消息传递
- 分布式共享存储器
  - 允许在网络上的多个进程共享一组页面，这些页面可以组成一个共享的线性地址空间

第四章 提纲

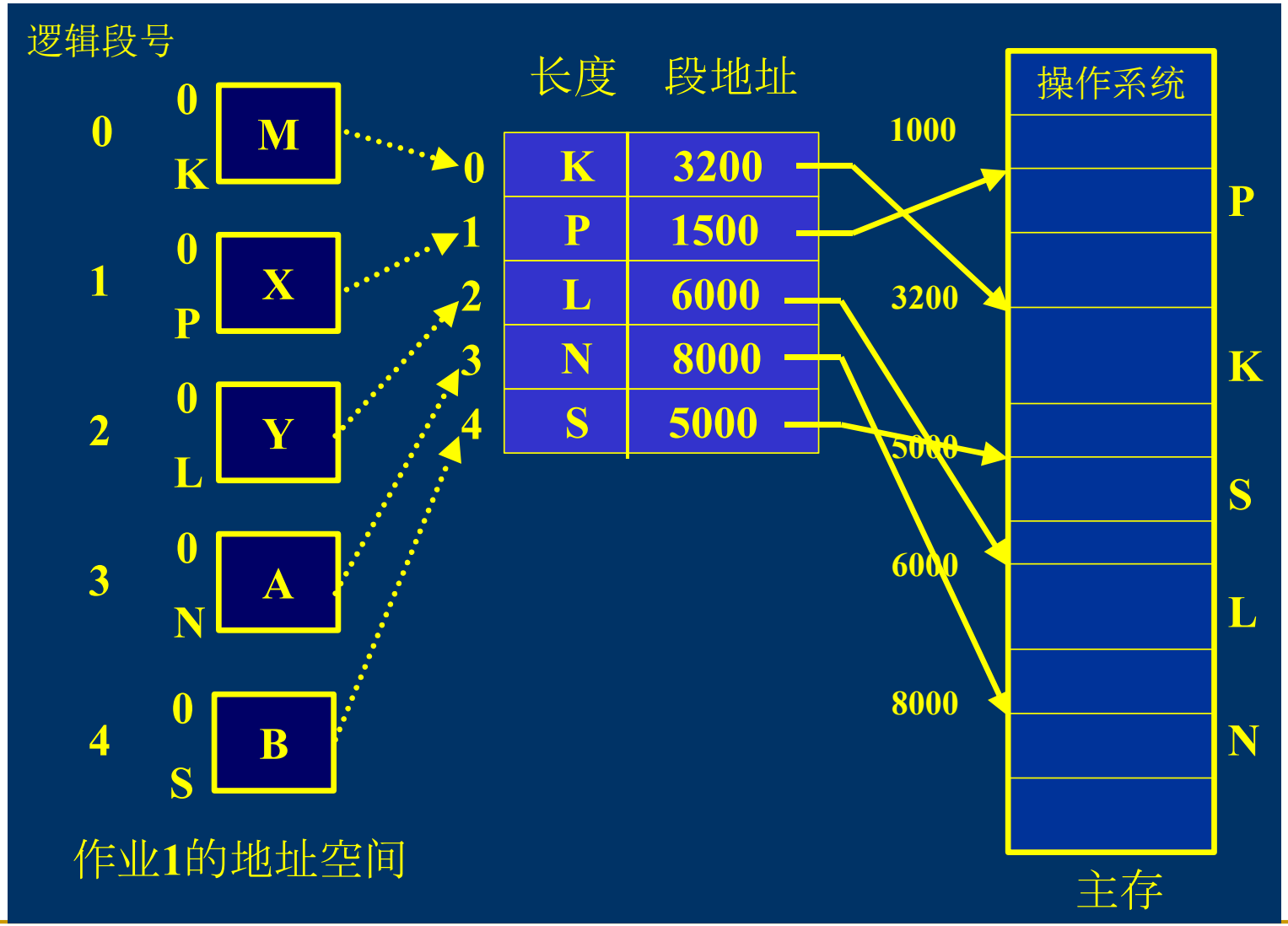
- 4.1 基本的内存管理
- 4.2 交换技术
- 4.3 虚拟存储管理
- 4.4 页面替换算法
- 4.5 页式存储管理的设计问题
- 4.6 段式存储管理
- 4.7 MINIX3进程管理器概述
- 4.8 MINIX3进程管理器实现

段式存储管理

- 页式管理是把内存视为一维线性空间；而段式管理是把内存视为二维空间

- 将程序的地址空间划分为若干个段(segment)，程序加载时，分配其所需的所有段(内存分区)，这些段不必连续；物理内存的管理采用动态分区。

例

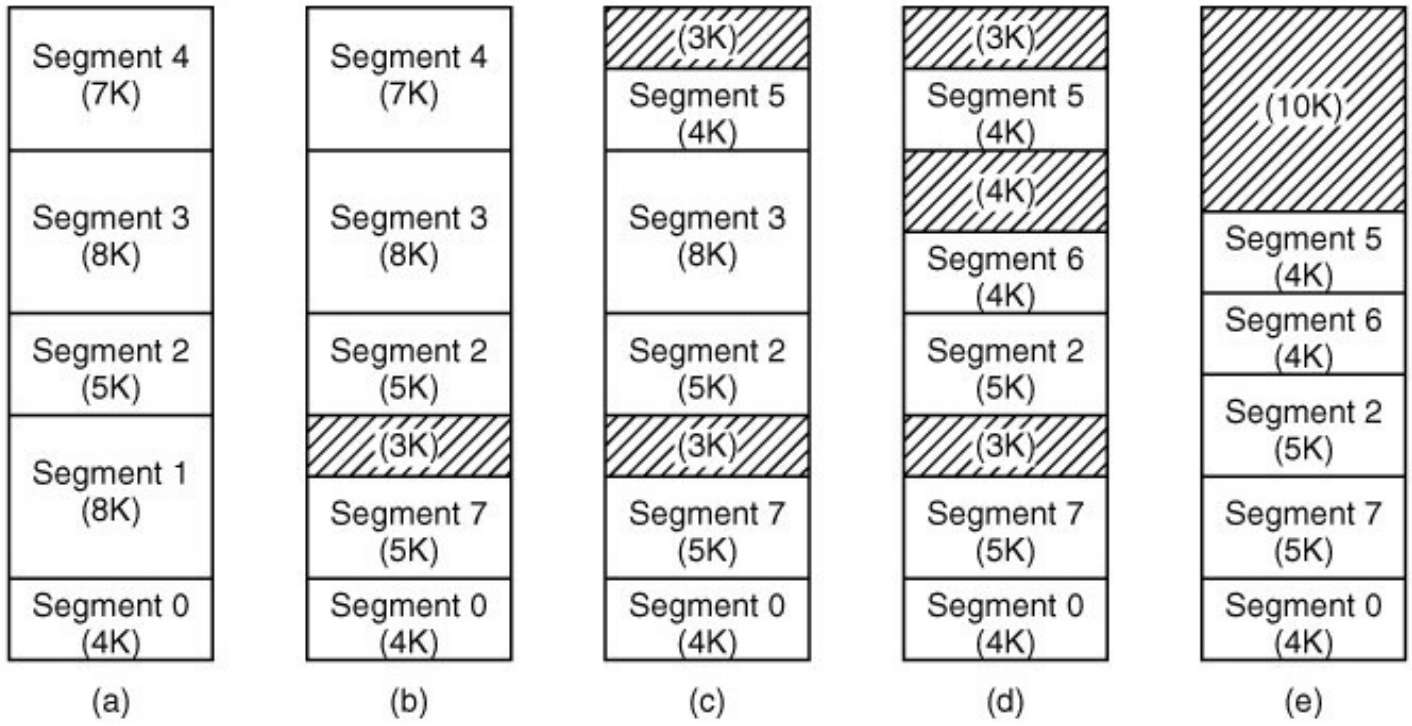

逻辑段号
0
K
M
1
P
X
2
L
Y
3
N
A
4
S
B
作业1的地址空间
长度
段地址
0
1
2
3
4
K 3200
P 1500
L 6000
N 8000
S 5000
操作系统
1000
3200
5000
6000
8000
P
K
S
L
N
主存

段式存储管理

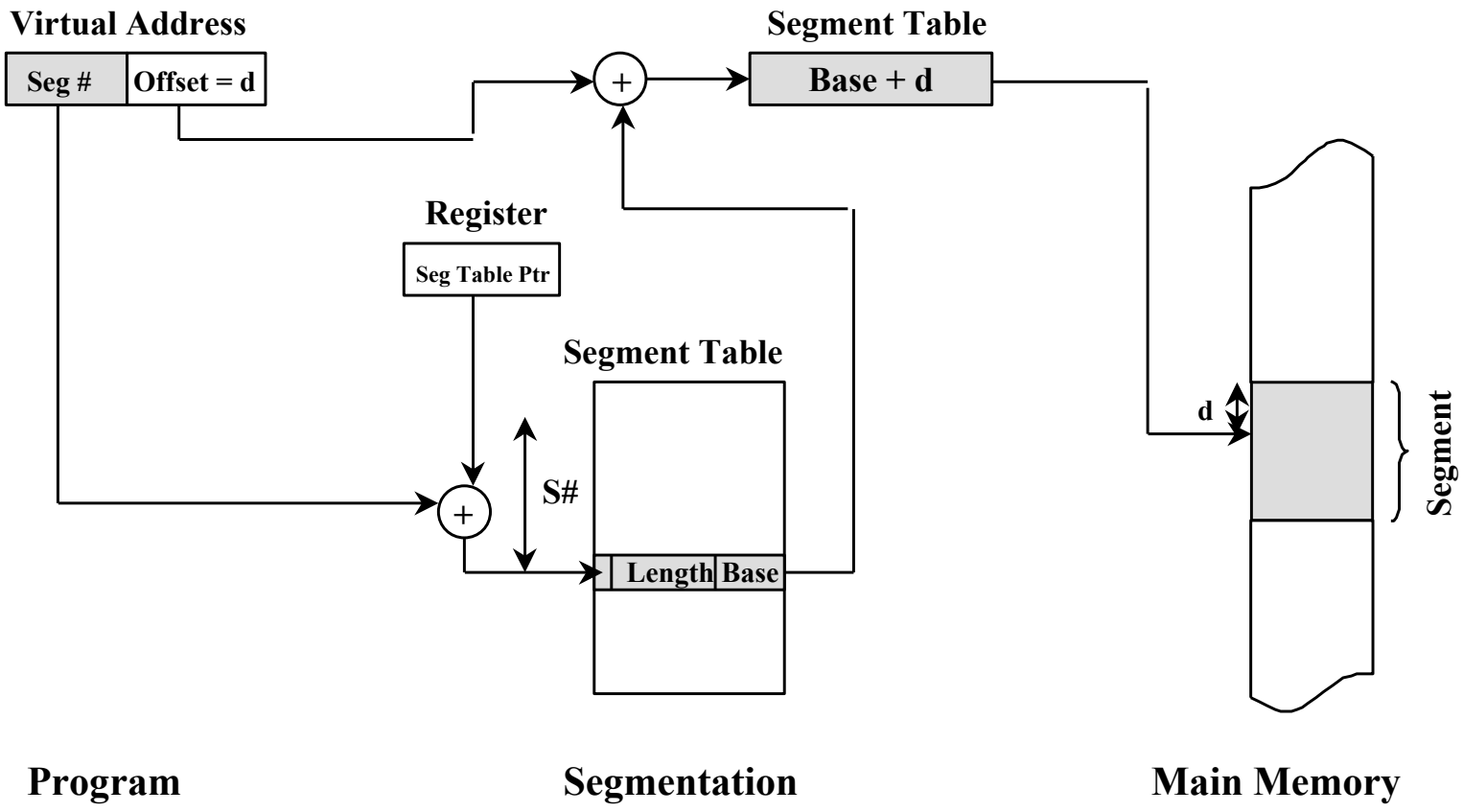
- 程序通过分段(segmentation)划分为多个模块，如代码段、数据段、共享段。
  - 可以分别编写和编译
  - 可以针对不同类型的段采取不同的保护
  - 可以按段为单位来进行共享，包括通过动态链接进行代码共享
- 优点：
  - 没有内碎片。
  - 便于改变进程占用空间的大小。
  - 易于实现代码和数据共享，如共享库
- 引入新的问题：
  - 存在外碎片，需要通过内存压缩来消除。

外碎片

- 通过内存压缩来消除外碎片



段式管理的地址变换


Virtual Address
Seg #
Offset = d
Segment Table
Base + d
Register
Seg Table Ptr
Segment Table
S#
Length
Base
d
Segment
Program
Segmentation
Main Memory

例：段式地址变换