第四章　存储管理

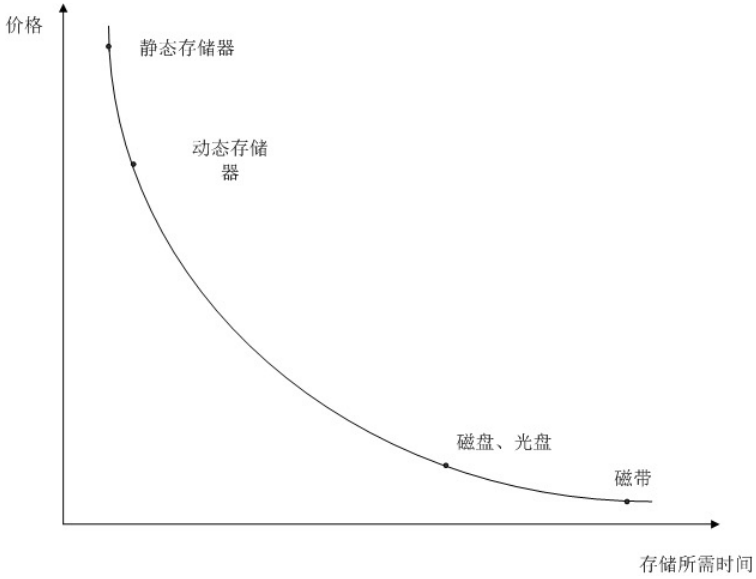
翁楚良

https://chuliangweng.github.io

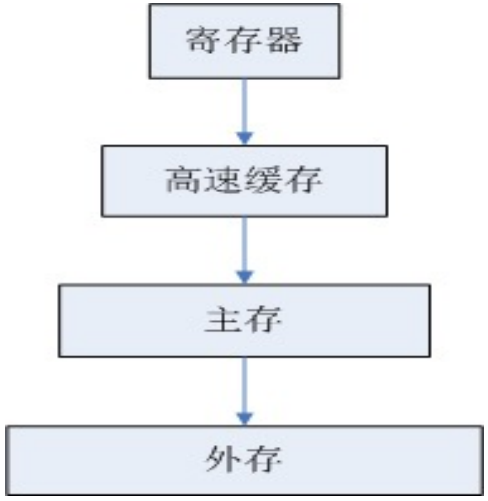
2023 春　ECNU

层次化存储体系结构

- 计算机的存储体系
  - 少量的非常快速、昂贵、易变的的高速缓存（cache）
  - 若干兆字节的中等速度、中等价格、易变的主存储器（RAM）
  - 数百兆或数千兆字节的低速、廉价、不易变的磁盘(外存)组成



- 操作系统的工作就是协调这些存储器的使用，其中管理存储器的部分程序称为存储管理器
  - 记录存储使用状况
  - 分配、回收存储资源
  - 数据的装入与写回



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

存储管理系统分类

- 在运行期间，进程需要在内存和磁盘之间换进换出的系统（交换和分页）和不需要换进换出的系统。

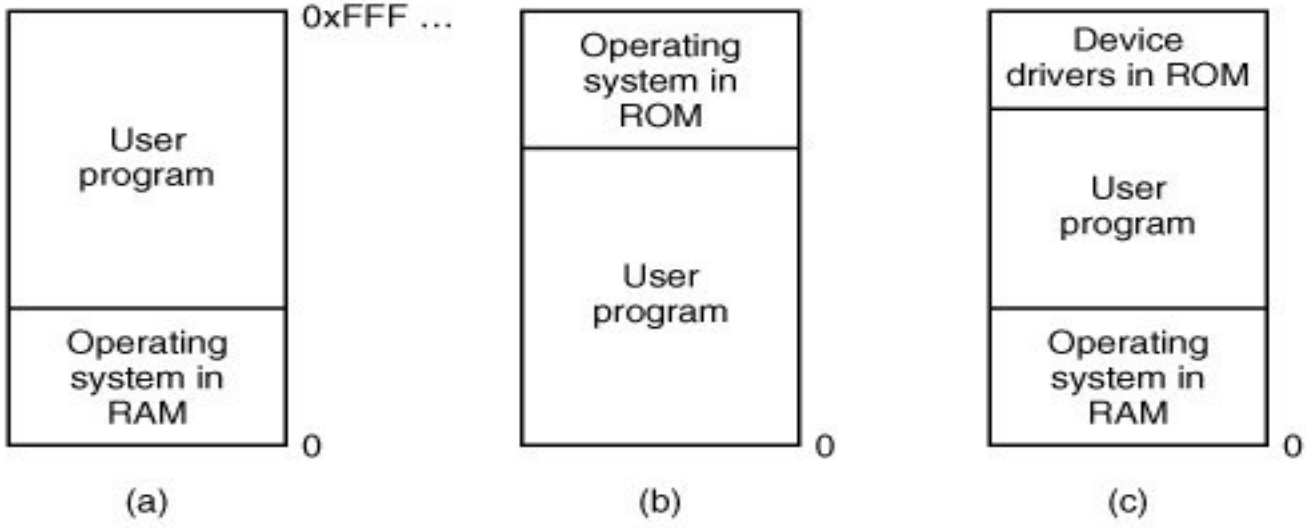
- 不需要换进换出的系统
  - 特点：进程被调入运行后，它将始终位于内存中，直至运行结束
    - 没有交换和分页的单道程序
    - 固定分区的多道程序

基本的存储管理

- 单道程序存储管理
- 多道程序存储管理
- 重定位和存储保护

没有交换或分页的单道程序

- 同一时刻只运行一道程序，应用程序和操作系统共享存储器
- 相应地，同一时刻只能有一个进程在存储器中运行。
  - 一旦用户输入了一个命令，操作系统就把需要的程序从磁盘拷贝到存储器中并执行它；在进程运行结束后，操作系统显示出一个提示符并等待新的命令。当收到新的命令时它把新的程序装入存储器，覆盖掉原来的程序。



早期的大型机小型机　　嵌入式系统　　早期PC机

基本的存储管理

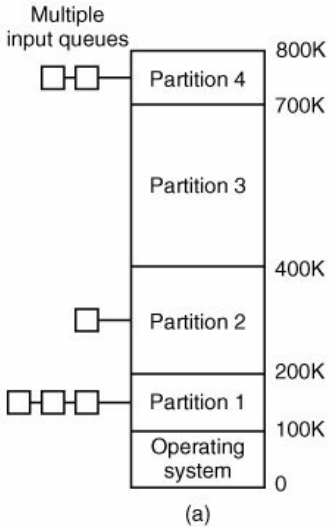
- 单道程序存储管理
- 多道程序存储管理
- 重定位和存储保护

多道程序(固定分区的系统)

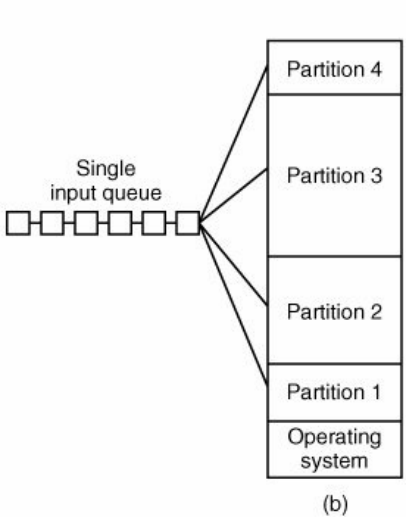
- 将内存划分为n个分区（可能不相等），分区的划分可以在系统启动时手工完成
  - 每个分区分别有一个运行队列
    - 当一个作业到达时，可以把它放到能够容纳它的最小的分区的输入队列中
  - 各分区共享同一个输入队列

IBM大型机的OS/360

由操作员在早晨设置好随后就不能再被改变的固定分区的系统



各分区有自己独立的输入队列



仅有单个输入队列的固定内存分区

基本的存储管理

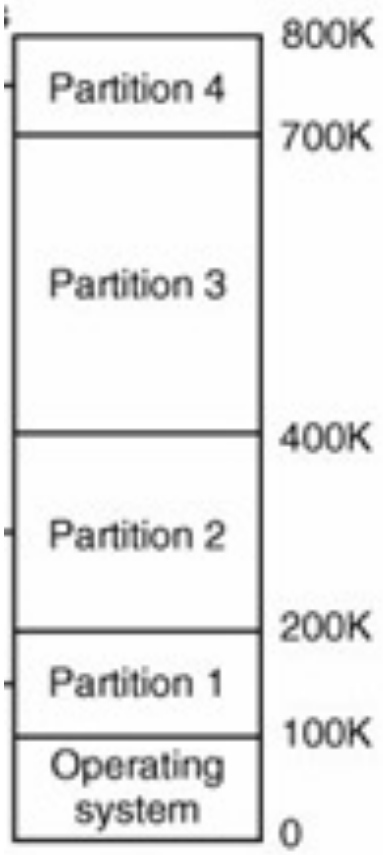
- 单道程序存储管理
- 多道程序存储管理
- 重定位和存储保护

重定位和保护

多道程序技术引发的问题???

- 重定位
  - 假设程序的第一条指令是调用在链接器产生的二进制文件中起始地址为100的一个过程。
  - 如果程序被装入分区1，这条指令跳转的目的地址将是绝对地址100，这会造成混乱，因为该地址在操作系统的内部。其实真正应该被调用的地址是100K+100。
  - 如果程序被装入分区2，它就应该去调用200K+100
- 保护
  - 在装入时重定位并没有解决保护问题，一个恶意的程序总可以生成一条新指令去访问任何它想访问的地址
  - 每个内存块分配4位的保护码，PSW中包含一个4位的密钥，若运行进程试图对保护码不同于PSW中密钥的主存进行访问，则由硬件引起一个陷入

一个既针对重定位又针对保护问题的解决方法是在机器中设置两个专门的寄存器，称为基址和界限寄存器



第四章 提纲

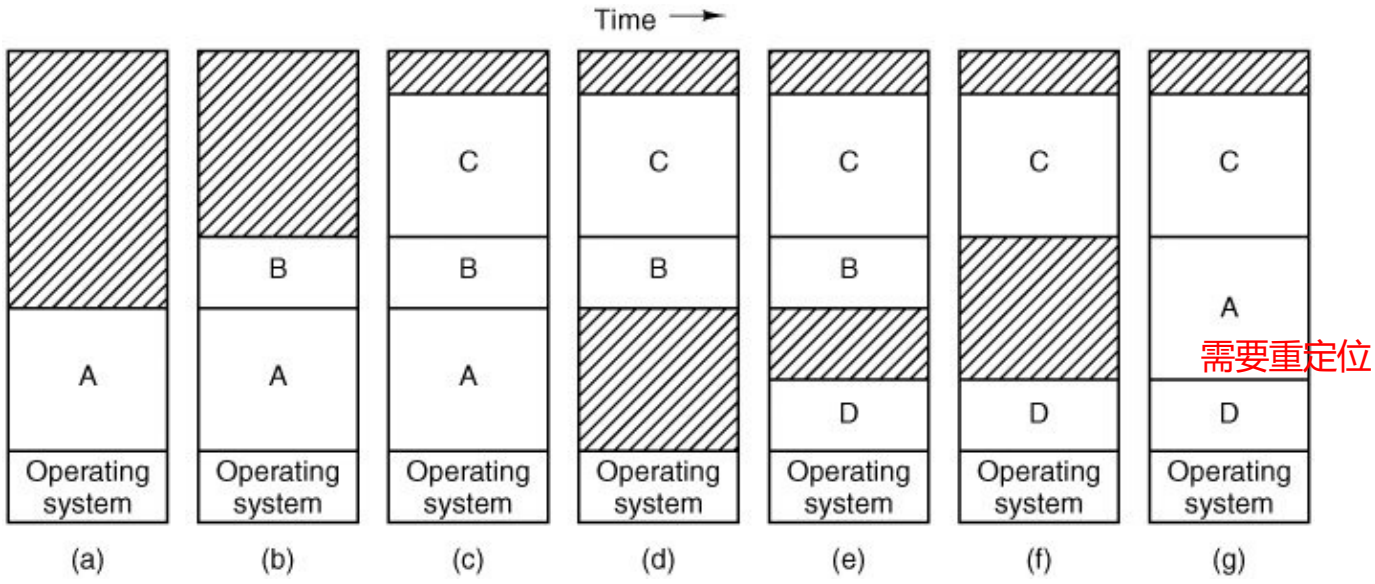
- 4.1 基本的内存管理
- 4.2 交换技术
- 4.3 虚拟存储管理
- 4.4 页面替换算法
- 4.5 页式存储管理的设计问题
- 4.6 段式存储管理
- 4.7 MINIX3进程管理器概述
- 4.8 MINIX3进程管理器实现

交换技术

虚拟存储器（virtual memory），使进程在只有一部分在主存的情况下也能运行。

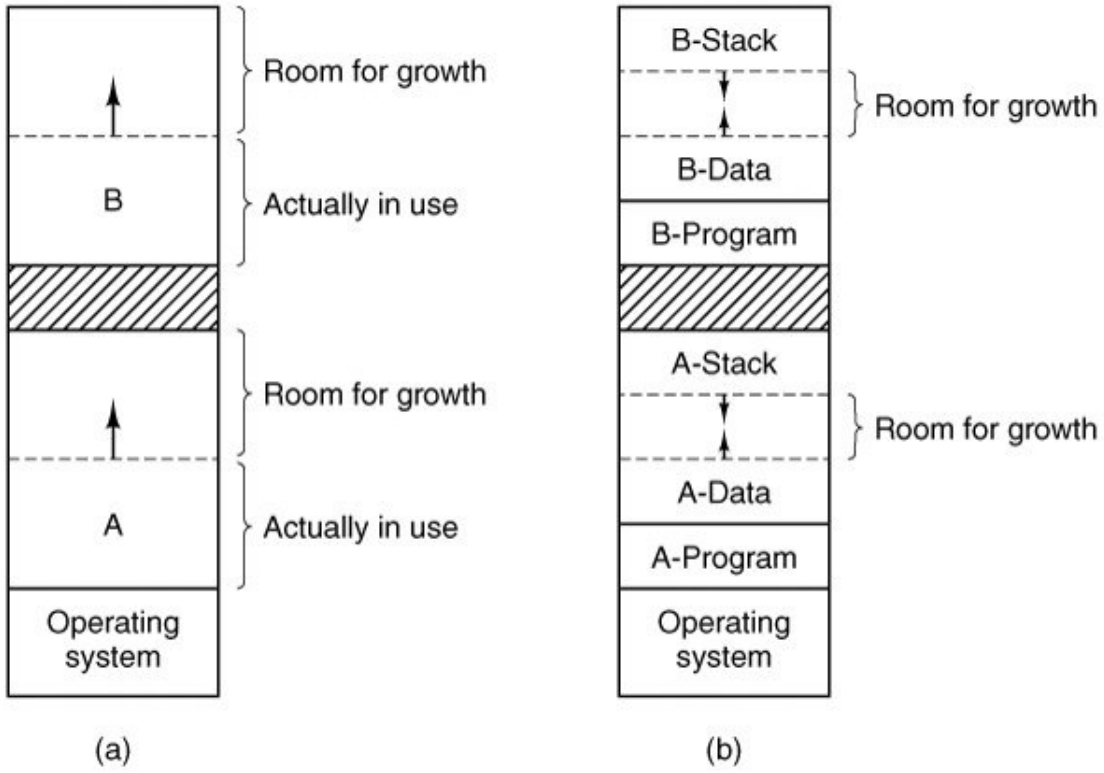
- 交换技术（swapping）
  - 把各个进程完整地调入主存，运行一段时间，再放回到磁盘上，过段时间再调入运行。

内存整理（memory compaction）
当交换在主存中生成了多个空洞时，可以把所有的进程向下移动至相互靠紧，从而把这些空洞结合成一大块



支持可变内存策略

- 进程分配内存固定，需要扩大内存时需将其移动到内存中一个足够大的空洞中
- 在进程被换进或移动时为其分配一点额外的内存

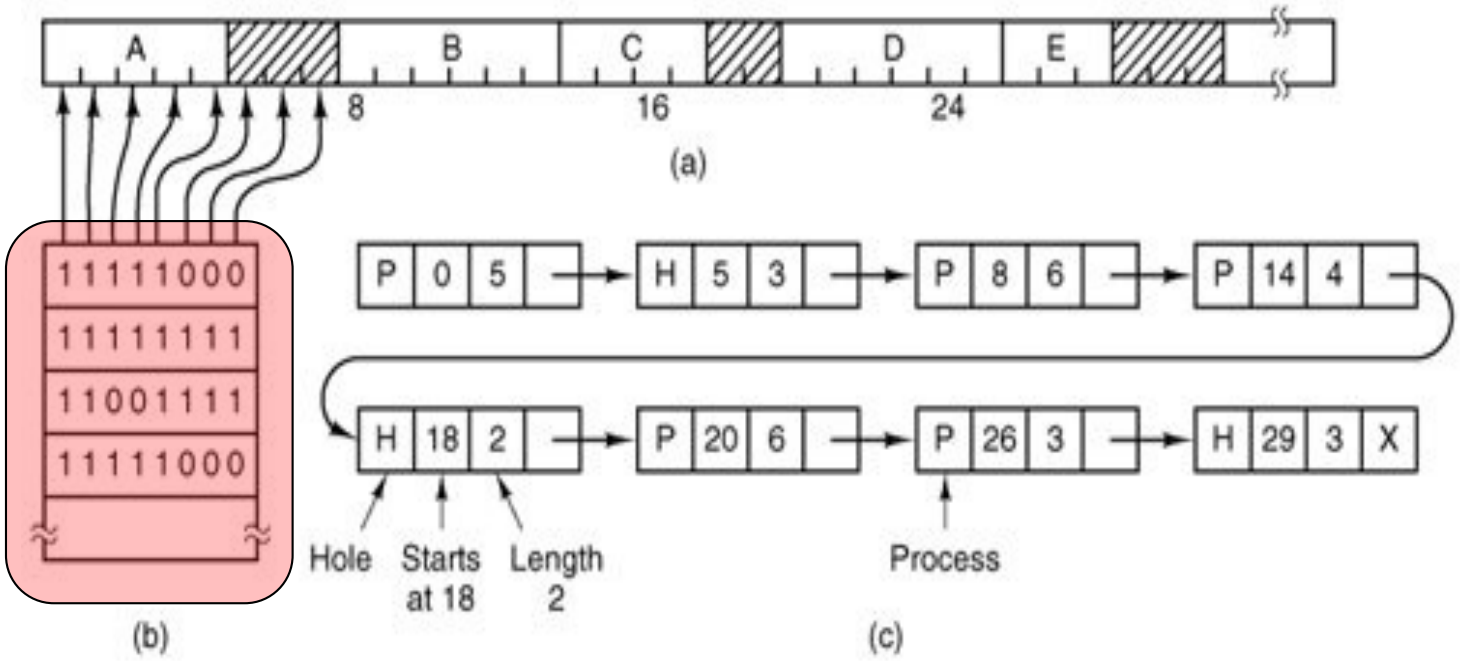


交换技术

- 进程被换入换出，内存动态分配，操作系统需要记录内存的使用情况
  - 基于位图的存储管理
  - 基于链表的存储管理

使用位图的内存管理

- 内存被划分为可能小到几个字或大到几千字节的分配单位，每个分配单位对应于位图中的一位，0表示空闲，1表示占用（或者反过来）。



位图的大小仅仅取决于内存和分配单位的大小

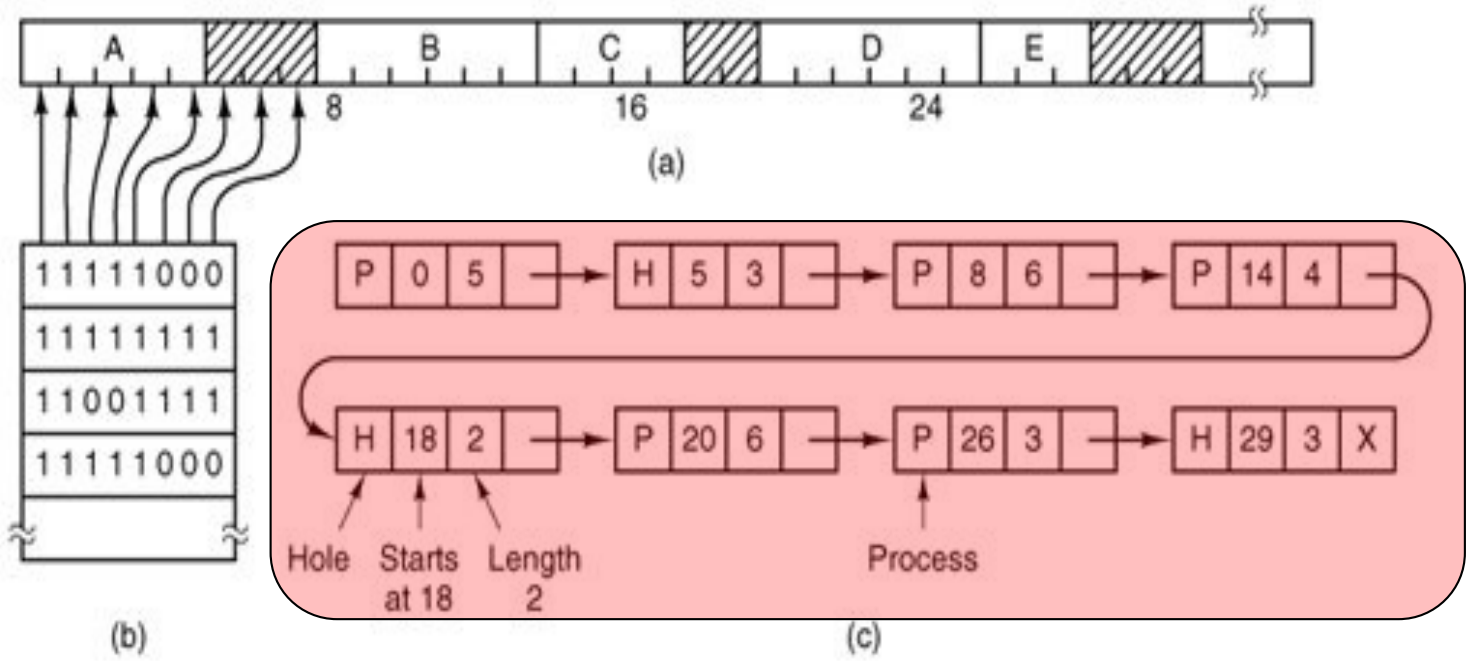
缺点：在位图中查找指定长度的连续0串是一个缓慢的操作

交换技术

- 进程被换入换出，内存动态分配，操作系统需要记录内存的使用情况
  - 基于位图的存储管理
  - 基于链表的存储管理

使用链表的内存管理

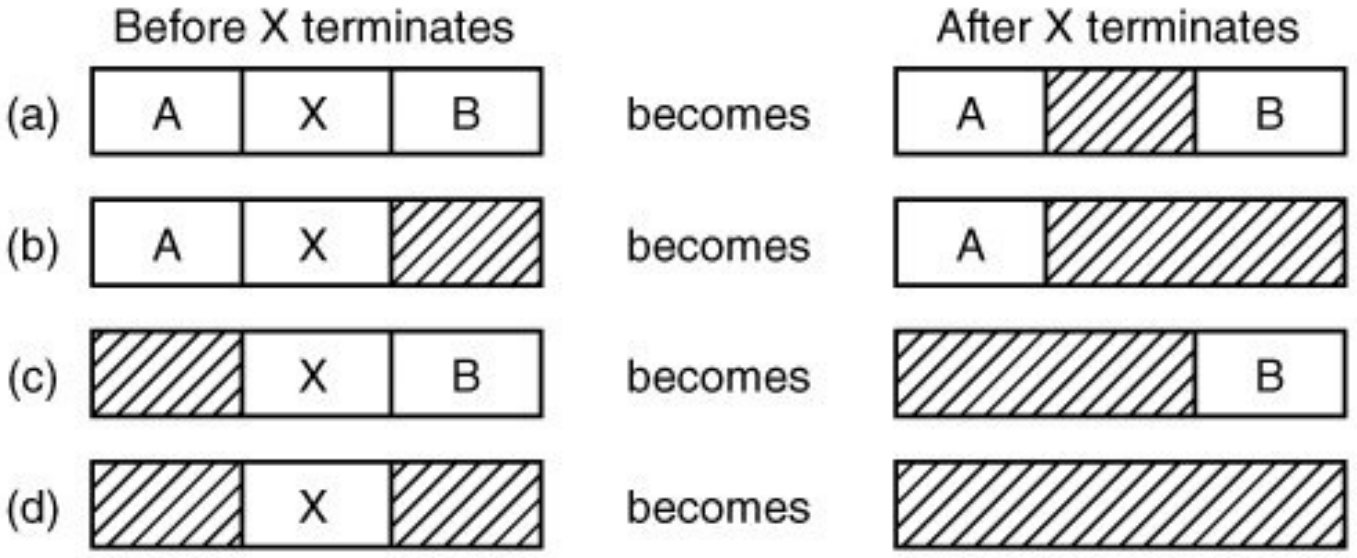
- 跟踪内存使用的另一个方法是维持一个已分配和空闲的内存段的链表



链表中的每一个表项都包含下列内容：指明是空洞(H)还是进程(P)的标志、开始地址、长度、和指向下一个表项的指针。

内存释放

- 一个要结束的进程一般有两个邻居（除非它是在内存的最低端或最高端），他们可能是进程也可能是空洞，这导致了图4-6所示的四种组合



内存分配算法

- 首次适配算法
  - 存储管理器沿着内存段链表搜索直到找到一个足够大的空洞，除非空洞大小和要分配的空间大小刚好一样，否则的话这个空洞将被分为两部分，一部分供进程使用，另一部分是未用的内存
- 下次适配
  - 每次找到合适的空洞时都记住当时的位置，在下次寻找空洞时从上次结束的地方开始搜索，而不是每次都从头开始
- 最佳适配算法
  - 试图找出最接近实际需要的大小的空洞，而不是把一个以后可能会用到的大空洞先使用
  - 最佳适配算法每次被调用时都要搜索整个链表，因此会比首次适配算法慢
- 最坏匹配算法
  - 在每次分配时，总是将最大的那个空闲区切去一部分，分配给请求者

内存分配算法

- 如果进程和空洞使用不同的链表，空洞链表可以按照大小排序以提高最佳适配的速度。
- 在最佳适配算法搜索由小到大排列的空洞链表时，当它找到一个合适的空洞时它就知道这个空洞是能容纳这个作业的最小的空洞，因此是最佳的，不需要象在单个链表的情况那样继续进行搜索。
- 当空洞链表按大小排序时，首次适配与最佳适配一样快，而下次适配则毫无意义。

快速适配法

- 为一些经常被用到长度的空洞设立单独的链表。
- 例，一个n个项的表，这个表的第一个项是指向长度为4K的空洞的链表的表头的指针，第二个项是指向长度为8K的空洞的链表的指针，第三个项指向长度12K的空洞链表，等等。
- 快速适配算法寻找一个指定大小的空洞是十分迅速的，但在一个进程结束或被换出时寻找它的邻接块以查看是否可以合并是非常费时间的

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

虚拟存储器

- 基本思想：操作系统把程序当前使用的那些部分保留在存储器中，而把其他部分保存在磁盘上

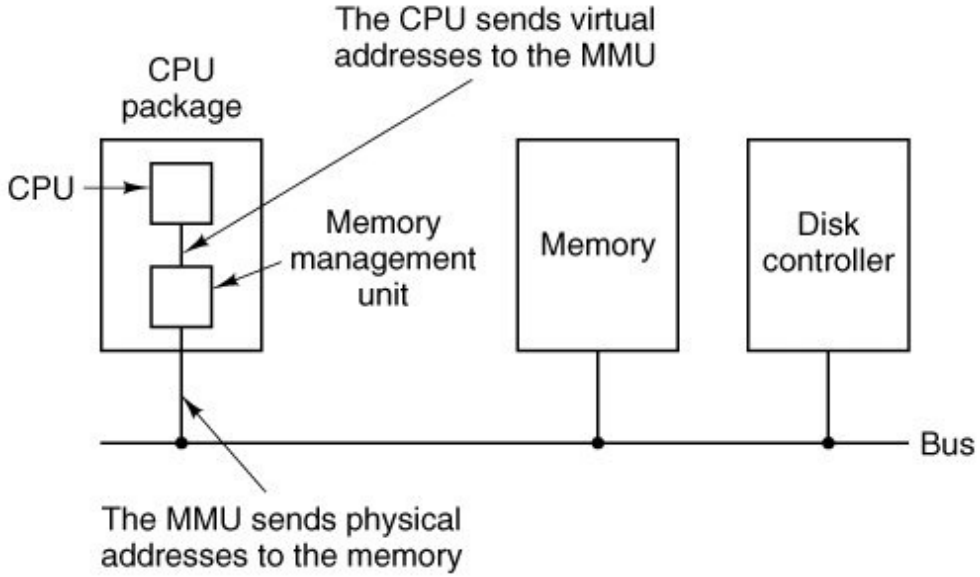
- 例
  - 一个16M的程序，通过仔细地选择在各个时刻将哪4M内容保留在内存中，并在需要时在内存和磁盘间交换程序的片段，那么就可以在一个4M的机器上运行。

虚拟存储管理

- 分页技术
- 页表
- 关联存储器TLB
- 反置页表

分页技术

- 虚地址空间被划分成称为页面（pages）的单位，在物理存储器中对应的单位称为页框（page frames），页和页框总是同样大小的。
- 由程序产生的地址被称为虚地址（virtual addresses），他们构成了一个虚地址空间（virtual address space）。
  - 在没有虚拟存储器的计算机上，虚地址被直接送到内存总线上，使具有同样地址的物理存储器字被读写
  - 在使用虚拟存储器的情况下，虚地址不是被直接送到内存总线上，而是送到存储管理单元(MMU)，它由一个或一组芯片组成，其功能是把虚地址映射为物理地址，

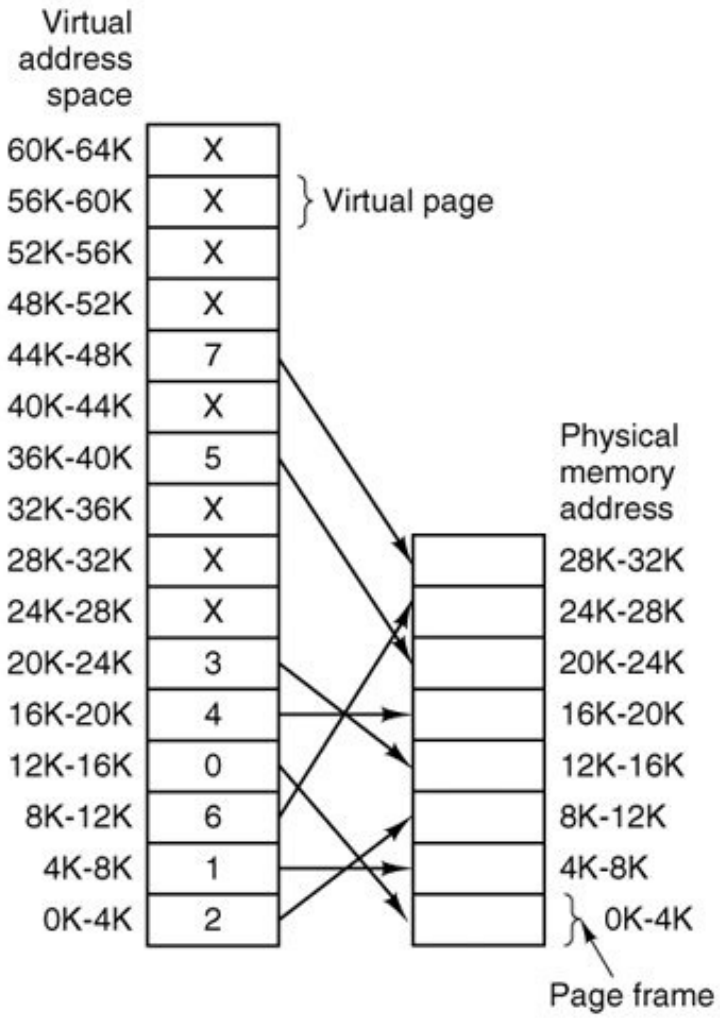


例

- 一台可以生成16位地址的计算机，地址变化范围从0到64K，这些地址是虚地址。
- 该台计算机只有32K的物理存储器，因此虽然可以编写64K的程序，他们却不能被完全调入内存运行。

MOVE REG, 0
MOVE REG, 8192

MOVE REG, 8192
MOVE REG, 24576

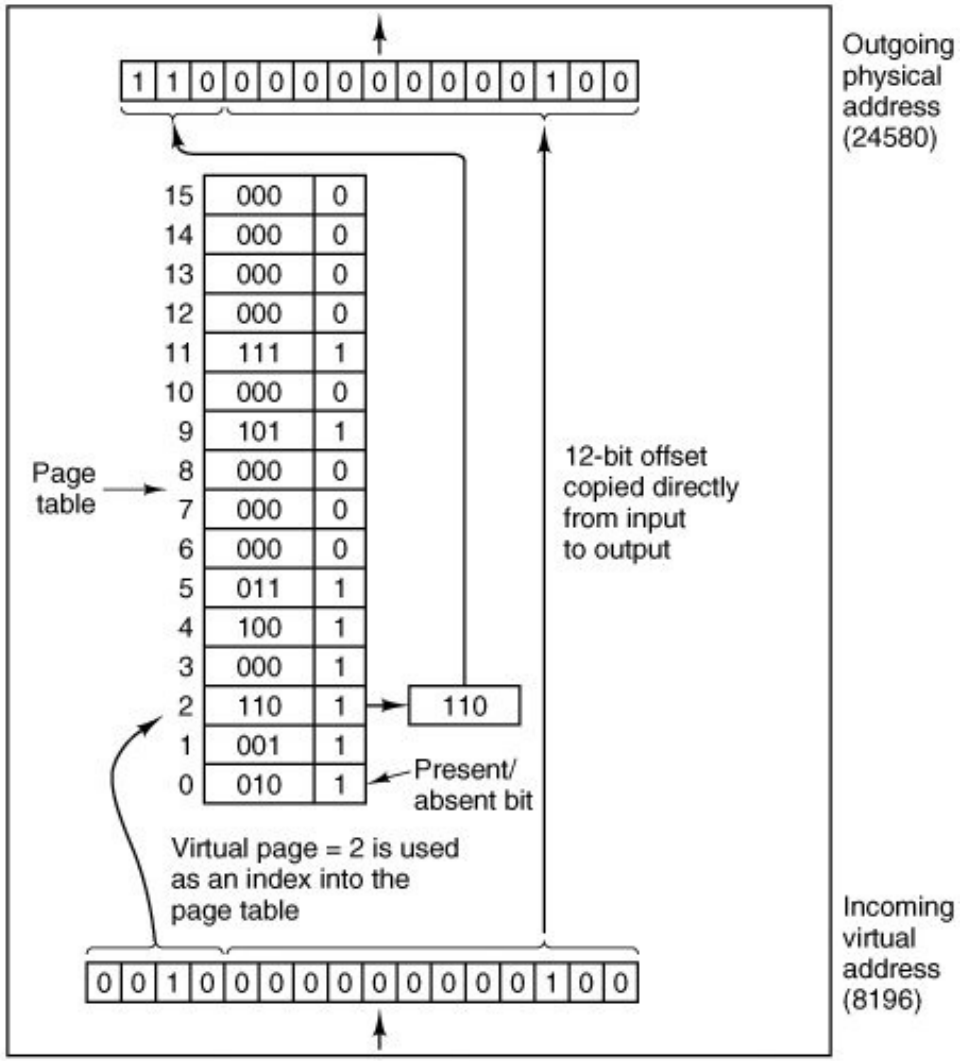


缺页故障

- 当访问未有映射的虚拟页，会引发陷入，这个陷入称为缺页故障
  - 操作系统找到一个很少使用的页框并把它的内容写入磁盘，随后把需引用的页取到刚才释放的页框中，修改映射，然后重新启动引起陷入的指令。

- 例，假设操作系统决定放弃页框1，那么它将把虚页8装入物理地址4K，并对MMU作两处修改：
  - 首先，它要标记虚页1为未映射，以使以后任何对虚地址4K到8K的访问都引起陷入；随后把虚页8对应表项的叉号改为1，
  - 因此在引起陷入的指令重新启动时，它将把虚地址32780映射为物理地址4108。

MMU内部结构



虚地址8196（二进制0010000000000100）

虚拟存储管理

- 分页技术
- 页表
- 关联存储器TLB
- 反置页表

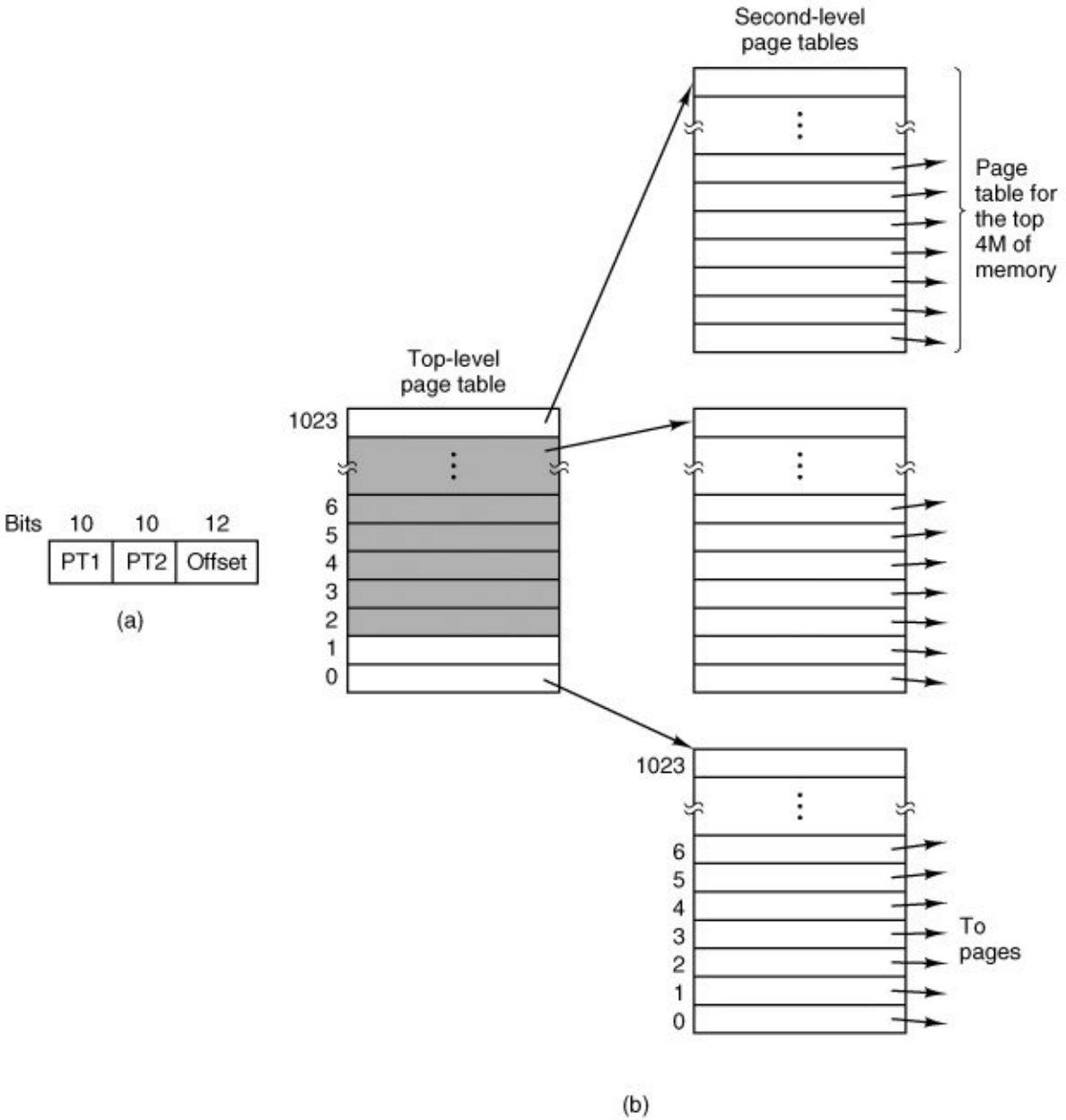
页表

- 虚地址被分成虚页号（高位）和偏移（低位）两部分，虚页号被用做页表的索引以找到该虚页对应的页表项，从页表项中可以找到页框号（如果有的话）。随后页框号被拼接到偏移的高位端，形成送往内存的物理地址。
- 从数学角度而言，页表是一个函数，它的参数是虚页号，结果是物理页框号。通过这个函数可以把虚地址中的虚页域替换成页框域，从而形成物理地址。

页表需要解决的两个主要问题
(1) 页表可能会非常大
(2) 地址映射必须十分迅速

多级页表(例)

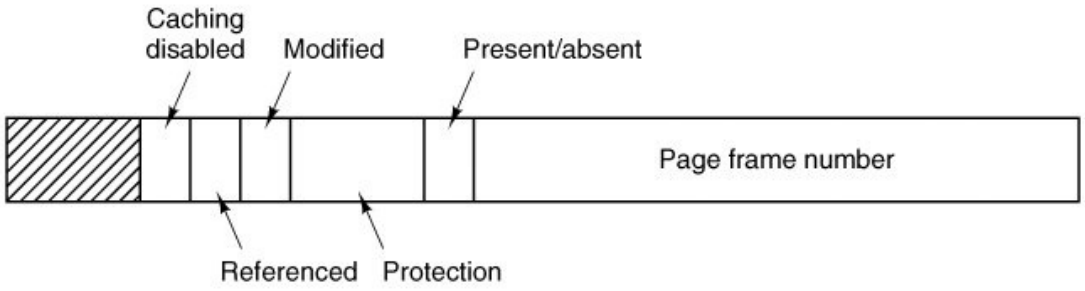
- 32位的虚地址划分为
  - 10位的PT1域
  - 10位的PT2域
  - 12位的偏移，相应地页长是4K，页面共有2^{20}个



多级页表优缺点

- 优点：避免将进程的所有页表项一直保存在内存中
- 缺点：需要多次访问内存，以查找页表
- 例：一个需要12兆字节的进程(地址空间大小为4GB)
  - 最低端是4兆程序正文，后面是4兆数据，顶端是4兆堆栈，在数据顶端上方和堆栈底端之间的上吉（千兆）字节的空洞根本没有使用
  - 虽然地址空间包含100万个页面(4GB/4KB=100万)，实际上只需要四个页表：顶级页表，0到4M、4M到8M和顶端4M的二级页表
  - 顶级页表中有1021个表项的Present/absent位都被设为0，使得访问他们时强制产生一个页面故障
    - 如果这种情况发生了，操作系统将注意到进程在试图访问一个不期望被访问的地址并采取适当的行动

页表项的结构



- 页框号，即物理页面号
- 有效位
  - 这一位是1时这个表项是有效的可以被使用，如果是0，表示这个表项对应的虚页现在不在内存中，访问这一位为0的页会引起一个页面故障。
- 保护位指出这个页允许什么样的访问。
  - 在最简单的形式下这个域只有一位，0表示读写，1表示只读。一个更先进的安排是使用三位，各位分别指出是否允许读、写、执行这个页。
- 修改位和访问位跟踪页的使用
  - 在一个页被写入时硬件自动设置修改位，此位在操作系统重新分配页框时是非常有用的，如果一个页已经被修改过（即它是“脏”的），则必须把它写回磁盘，否则只用简单地把它丢弃就可以了，因为它在磁盘上的拷贝仍然是有效的。此位有时也被称为脏(dirty)位，因为这反映了该页的状态。
  - 访问位在该页被引用时设置，不管是读还是写。它的值被用来帮助操作系统在发生页面故障时选择淘汰的页，不在使用的页要比在使用的页更适合于被淘汰。
- 禁止缓存位
  - 这个特性对那些映射到设备寄存器而不是常规内存的页面是非常重要的。

虚拟存储管理

- 分页技术
- 页表
- 关联存储器TLB
- 反置页表

TLB

- 在现代计算机系统中，一小部分页表的拷贝被保存在处理器芯片中，这部分页表是由最近访问页的页表项组成，这小部分页表是保存在转换后备缓冲缓存(TLB)中(通常在MMU中)。

当一个虚地址被送到MMU翻译时，硬件首先把它和TLB中的所有条目同时（并行地）进行比较。

如果找到了并且这个访问没有违反保护位，它的页框号将直接从TLB中取出而不用去查页表

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

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

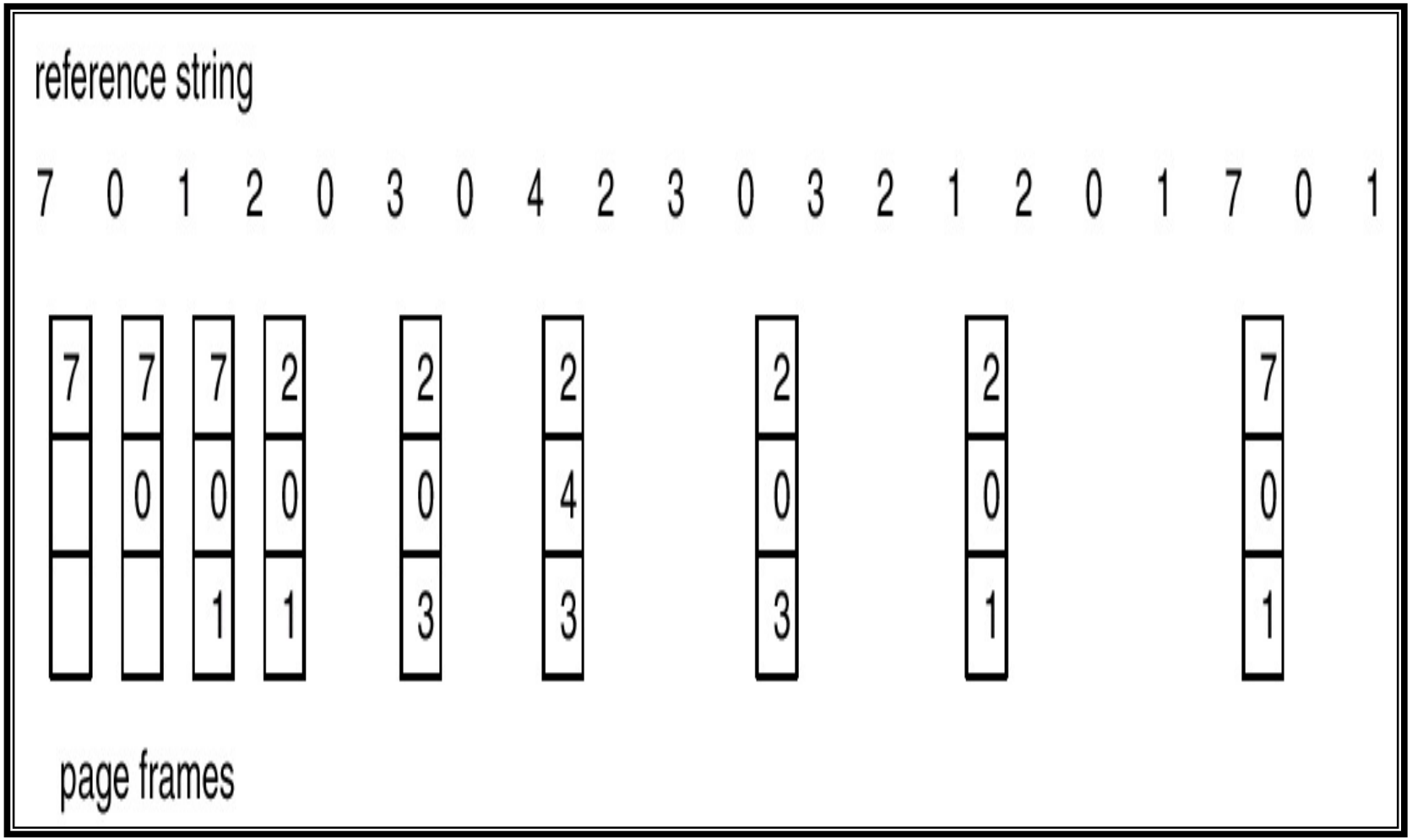
页面替换算法

- 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

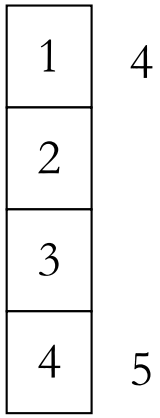
- The optimal policy selects for replacement the page for which the time to the next reference is the longest:
  - produces the fewest number of page faults.
  - impossible to implement (need to know the future) but serves as a standard to compare with the other algorithms we shall study.

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

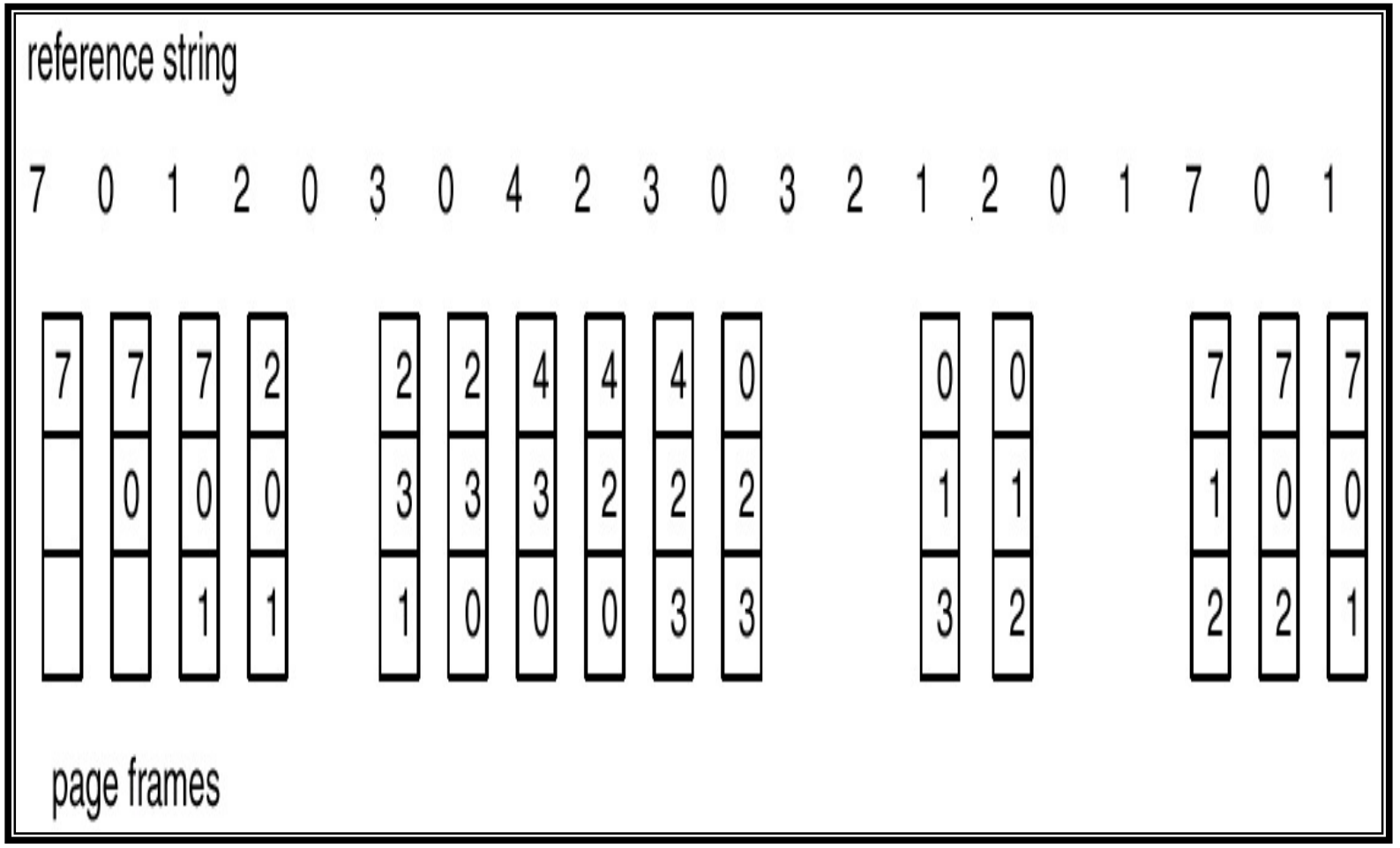
页面替换算法

- 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

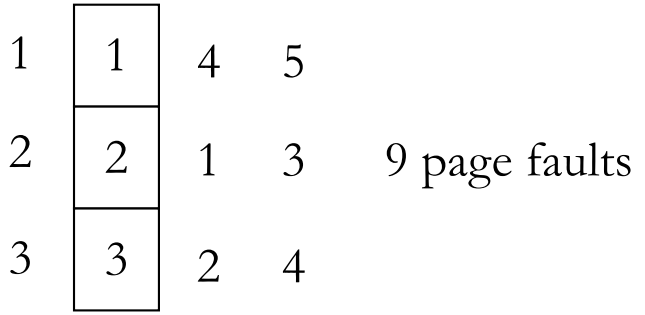
- Treats page frames allocated to a process as a circular buffer:
  - When the buffer is full, the oldest page is replaced. Hence first-in, first-out:
    - A frequently used page is often the oldest, so it will be repeatedly paged out by FIFO.
  - Simple to implement:
    - requires only a pointer that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

FIFO Page Replacement Algorithm

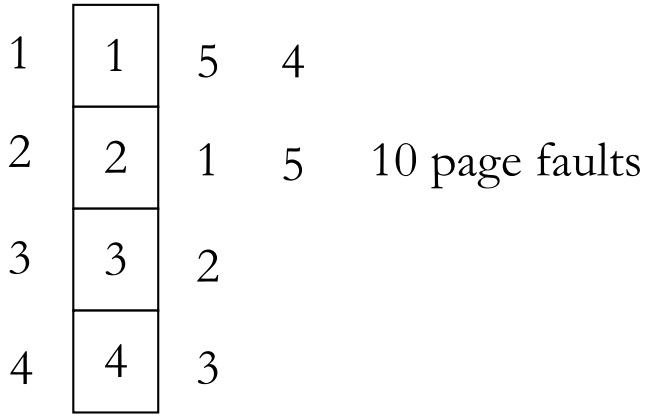


FIFO Page Replacement Algorithm

- 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

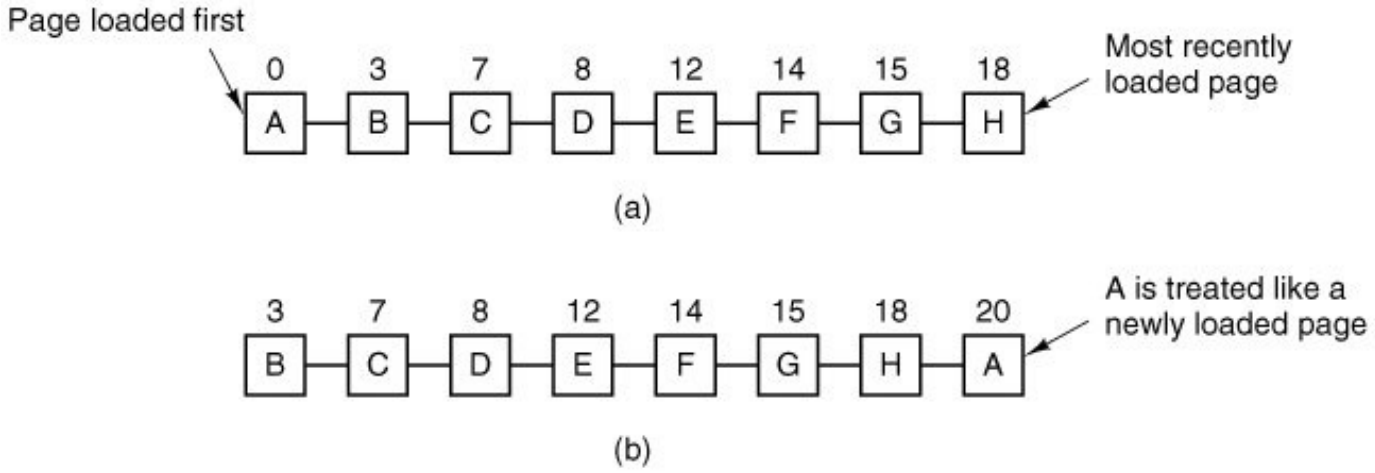


页面替换算法

- 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

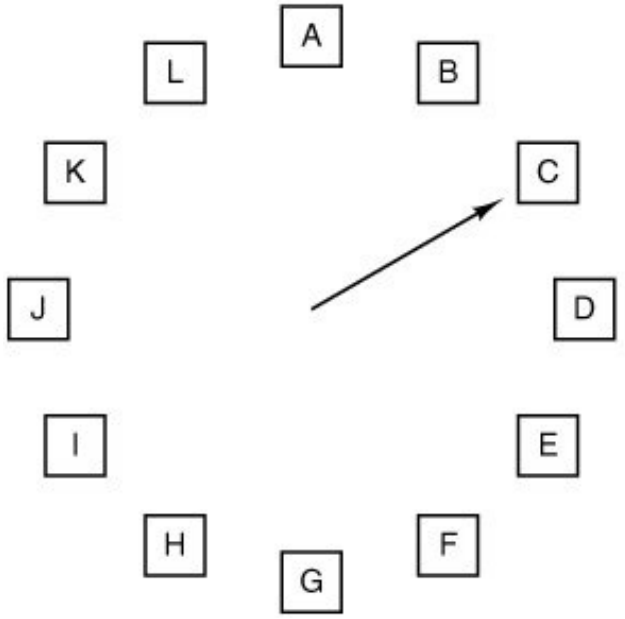
- 最优页面置换算法
- 未最近使用页面置换算法
- 先进先出页面置换算法
- 第二次机会页面置换算法
- 时钟页面置换算法
- 最近最久未使用页面置换算法

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

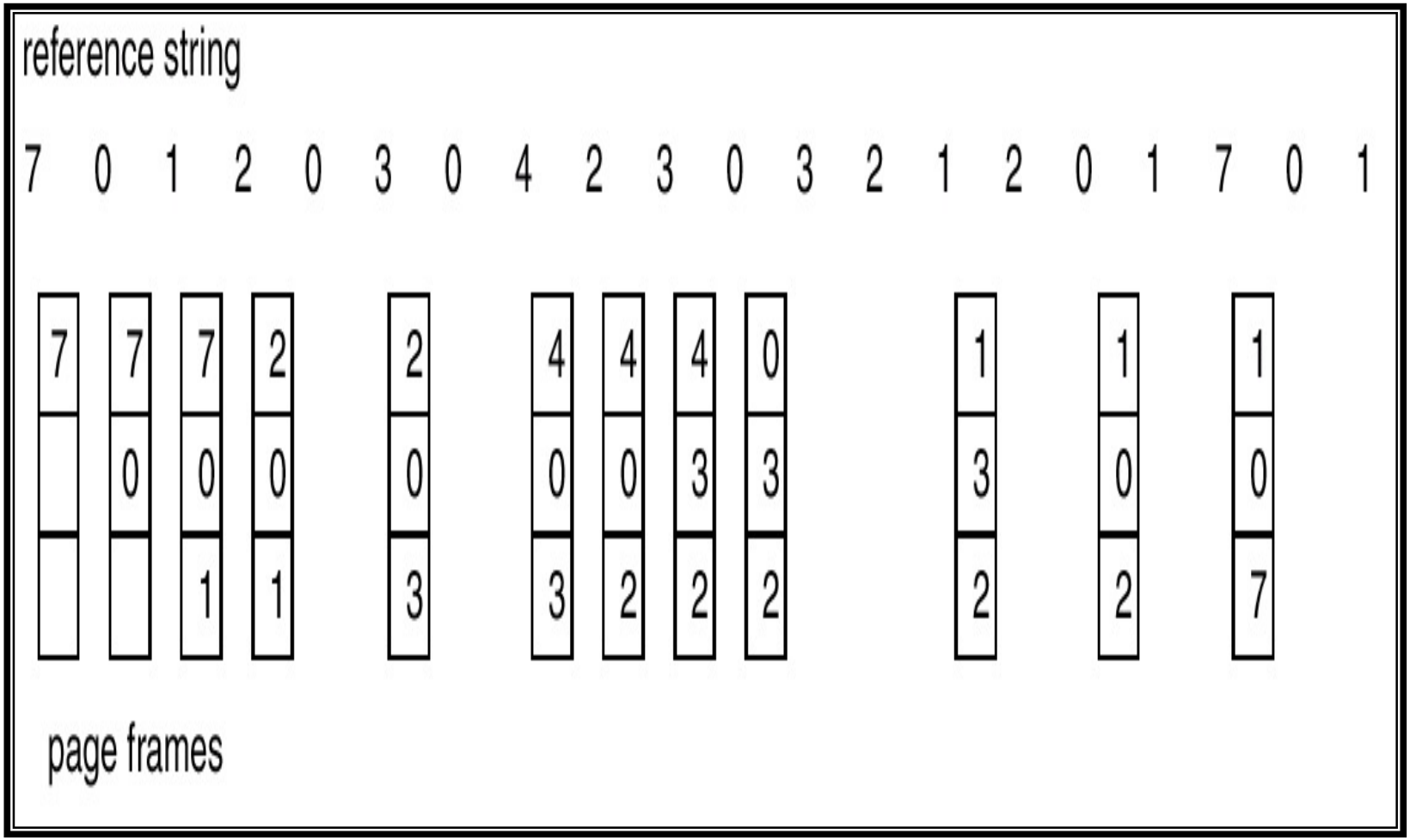
页面替换算法

- 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

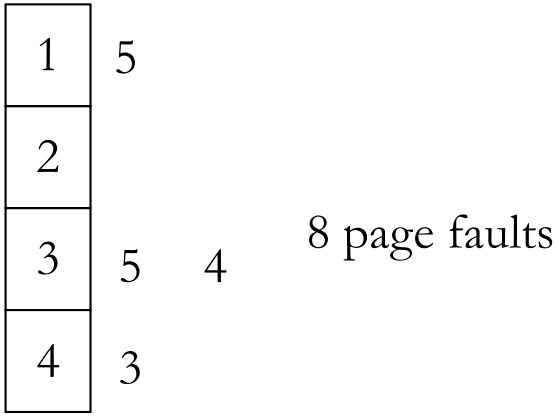
- Replaces the page that has not been referenced for the longest time:
  - By the principle of locality, this should be the page least likely to be referenced in the near future.
  - performs nearly as well as the optimal policy.

LRU Page Replacement



LRU Page Replacement

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

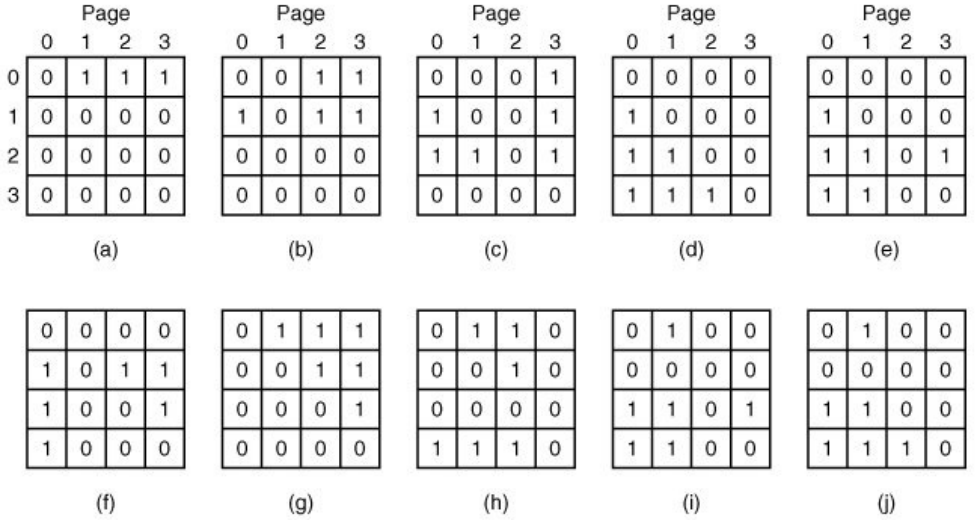


LRU Implementations

- 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

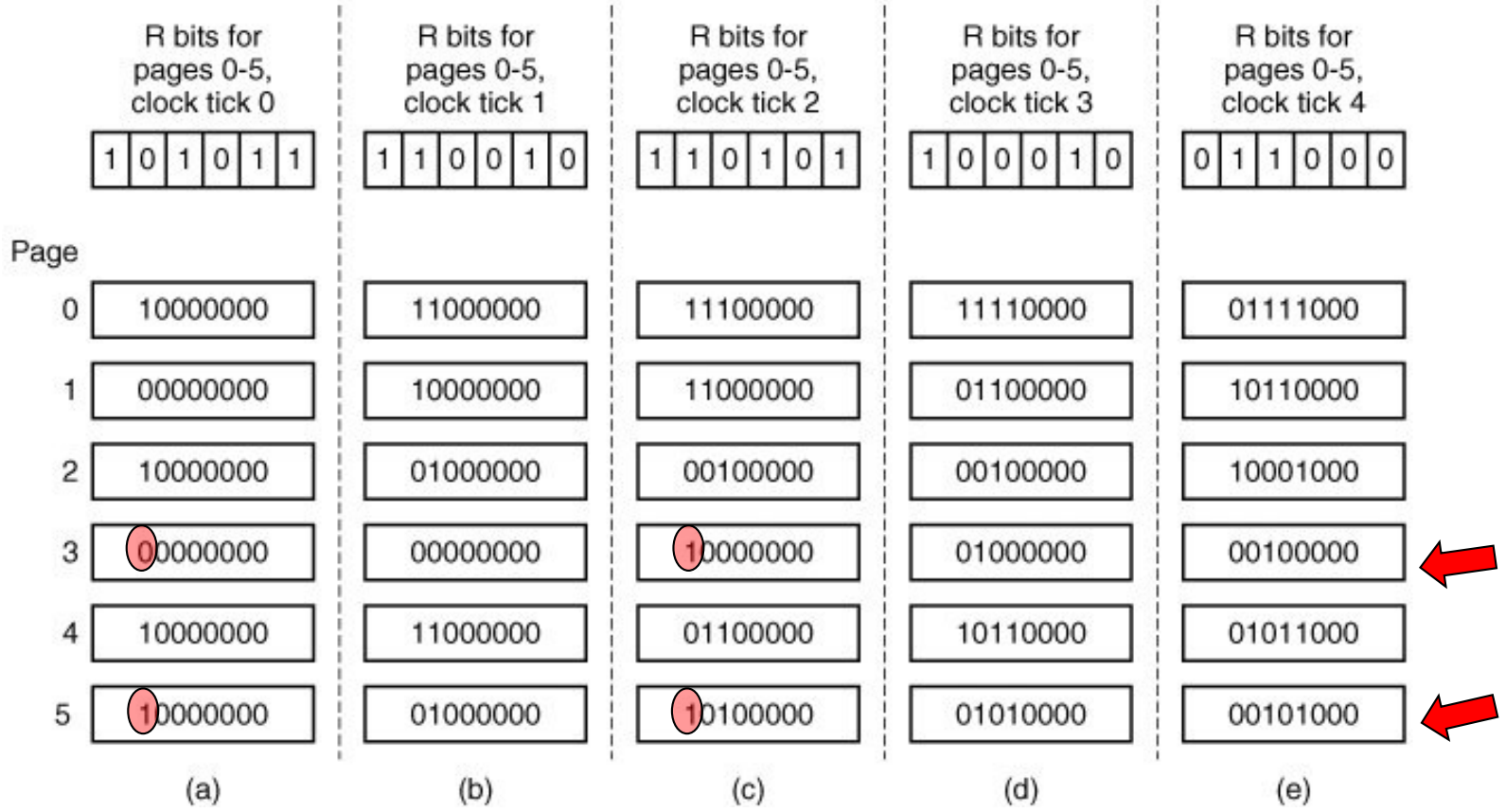
- NFU (Not Frequently Used) algorithm
  - It requires a software counter associated with each page, initially zero.
  - At each clock interrupt, the operating system scans all the pages in memory. For each page, the R bit, which is 0 or 1, is added to the counter.
  - When a page fault occurs, the page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

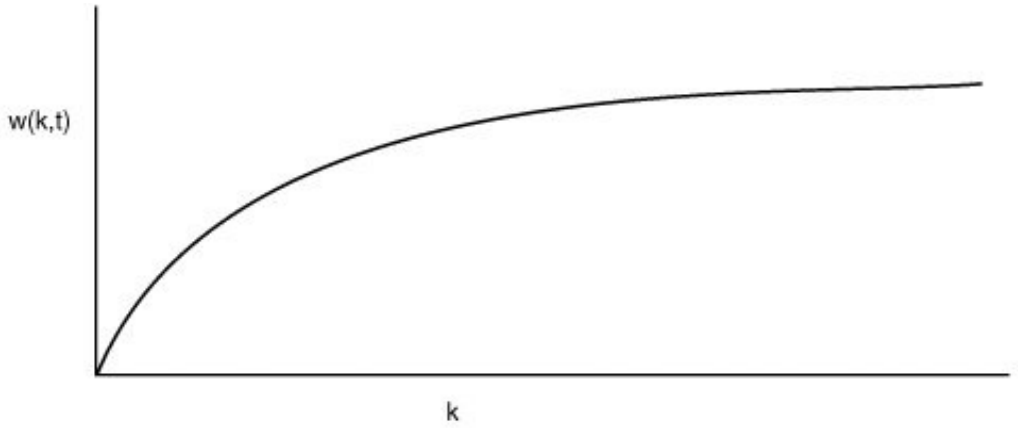
页式存储管理的设计问题

- 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

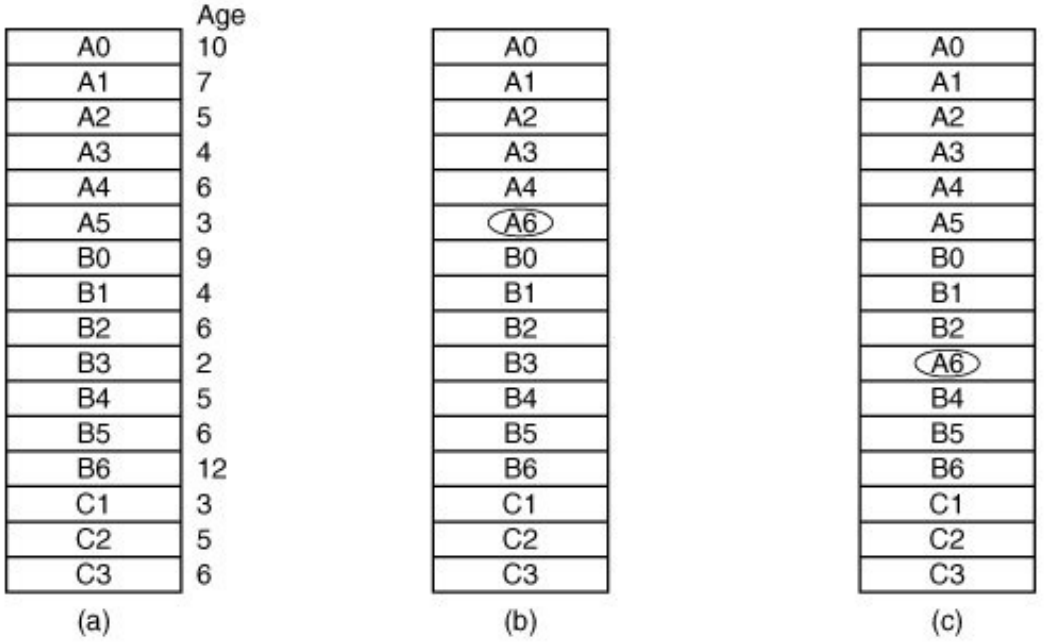


页式存储管理的设计问题

- 工作集模型
- 分配策略
- 页面大小
- 虚拟存储接口

分配策略

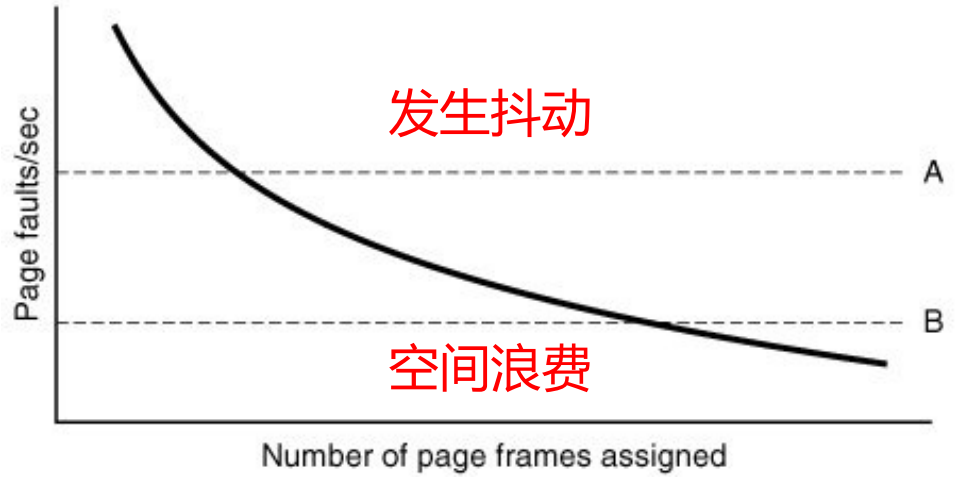
- 页面置换算法作用的范围不同，对应不同的分配策略
  - 局部页面置换算法：在进程所分配的页面范围内选取将被置换的页面
    - 每个进程分配固定大小的内存空间
  - 全局页面置换算法：在内存中所有的页面范围内选取被置换的页面
    - 所有进程动态共享系统的物理页面，分配给每个进程的页面数动态变化的



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

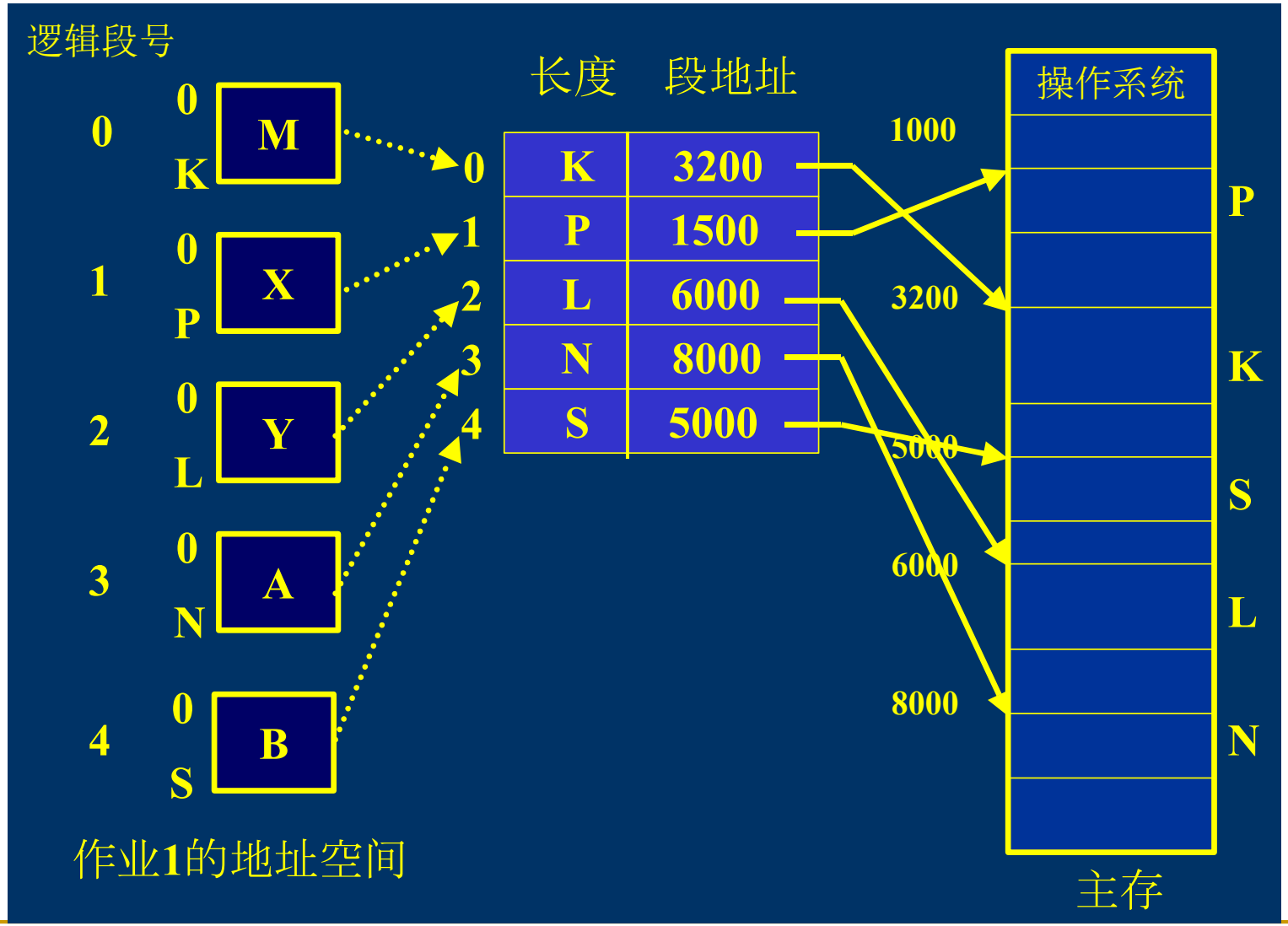
- 4.1 基本的内存管理
- 4.2 交换技术
- 4.3 虚拟存储管理
- 4.4 页面替换算法
- 4.5 页式存储管理的设计问题
- 4.6 段式存储管理
- 4.7 MINIX3进程管理器概述
- 4.8 MINIX3进程管理器实现

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

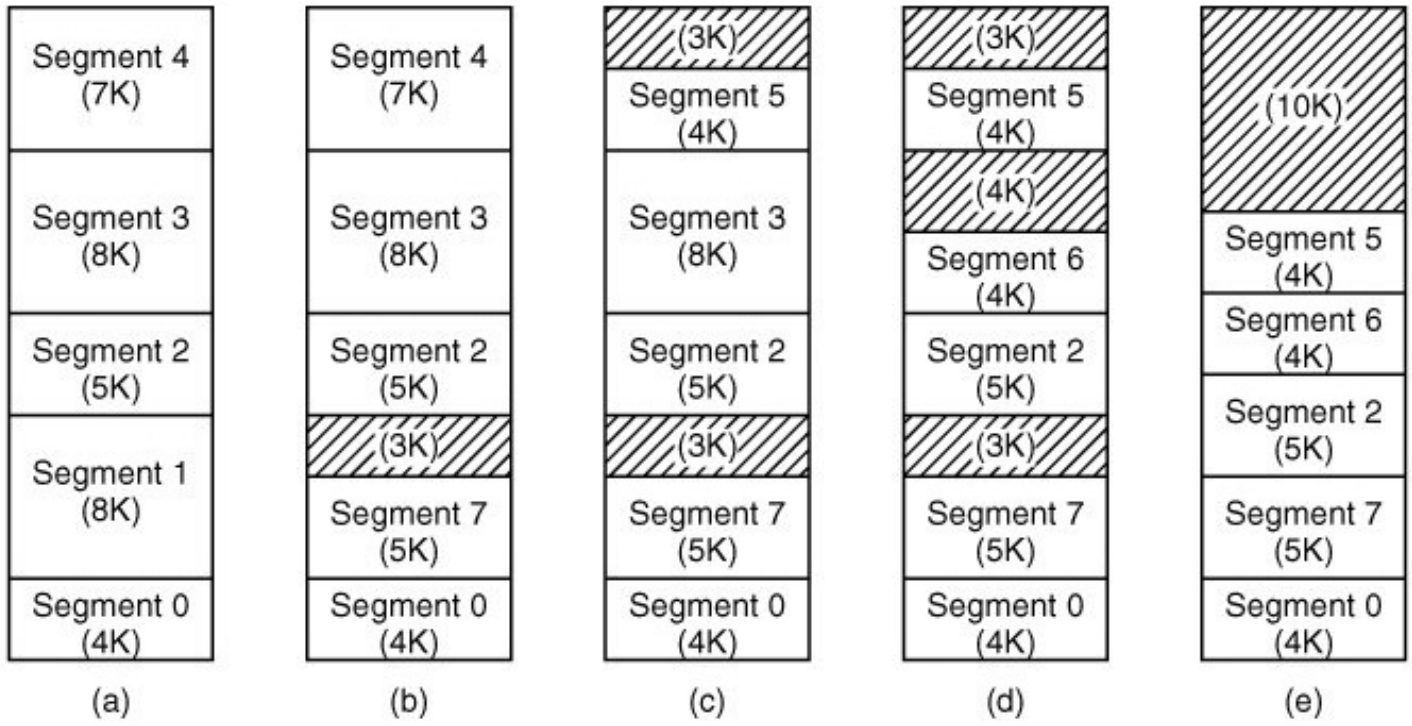

逻辑段号
0
K
M
1
P
X
2
L
Y
3
N
A
4
S
B
作业1的地址空间
长度
段地址
0
1
2
3
4
K 3200
P 1500
L 6000
N 8000
S 5000
操作系统
1000
3200
5000
6000
8000
P
K
S
L
N
主存

段式存储管理

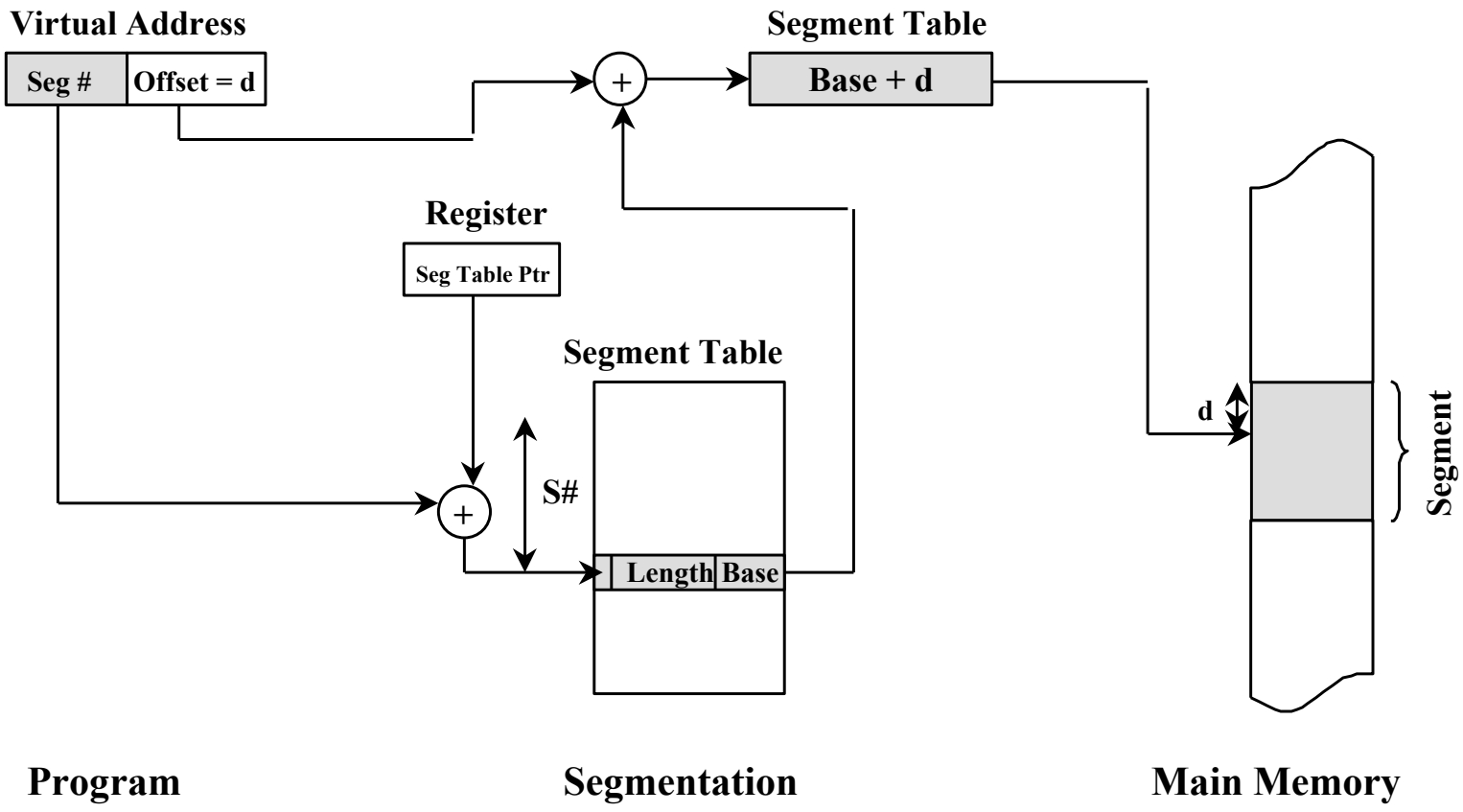
- 程序通过分段(segmentation)划分为多个模块，如代码段、数据段、共享段。
  - 可以分别编写和编译
  - 可以针对不同类型的段采取不同的保护
  - 可以按段为单位来进行共享，包括通过动态链接进行代码共享
- 优点：
  - 没有内碎片。
  - 便于改变进程占用空间的大小。
  - 易于实现代码和数据共享，如共享库
- 引入新的问题：
  - 存在外碎片，需要通过内存压缩来消除。

外碎片

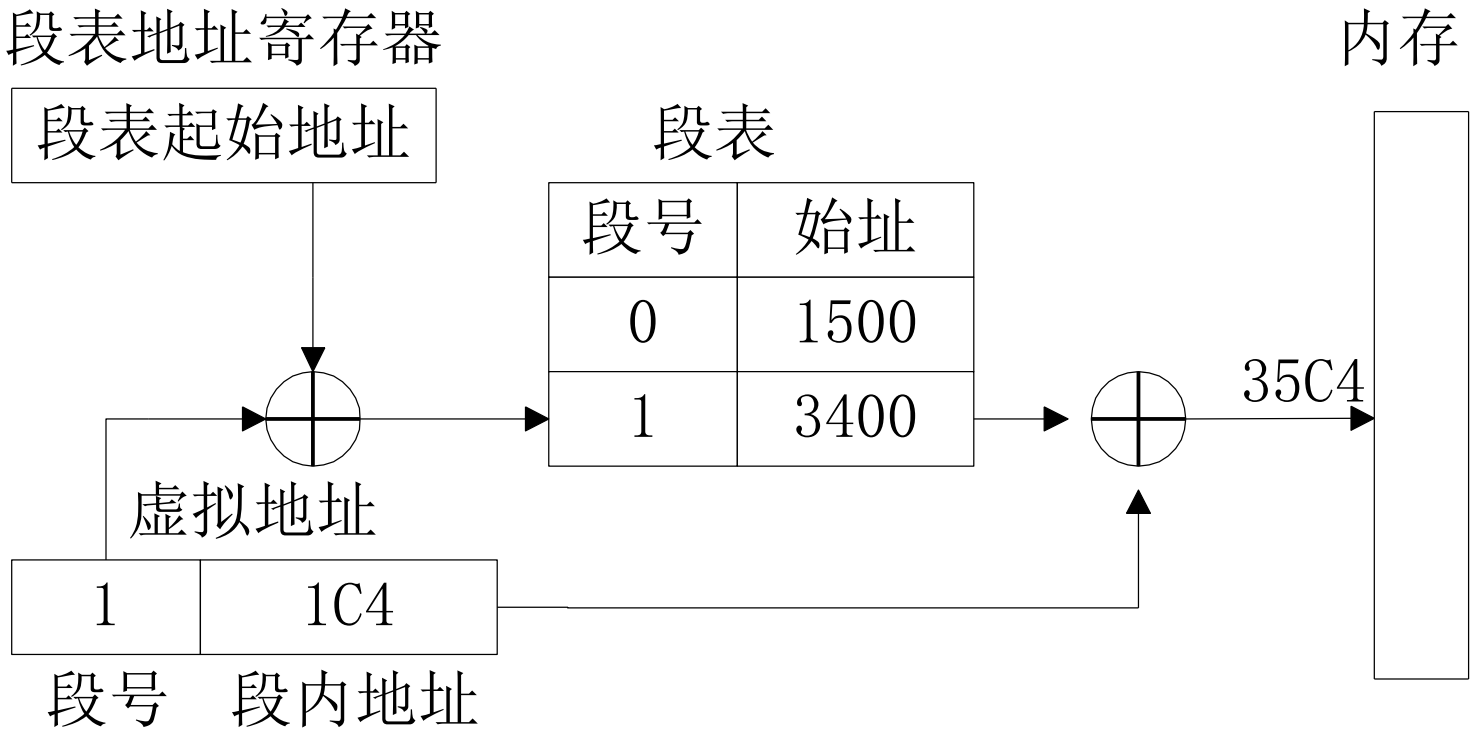
- 通过内存压缩来消除外碎片



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



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

MINIX3进程管理器概述

- MINIX3不支持页式存储管理、提供了交换所需的相关代码
- 进程管理器：负责处理与进程管理相关的系统调用，包括存储管理
  - 进程管理
  - 存储管理 (存储管理器)
- 存储管理器保存着一张按照内存地址排列的空洞列表，当由于执行系统调用FORK或EXEC需要内存时，系统将用最先匹配算法对空洞列表进行搜索找出一个足够大的空洞。
- 一旦一个程序被装入内存，它将一直保持在原来的位置直到运行结束，它不会被换出或移动到内存的其他位置去，为它分配的空间也不会增长或缩小。

策略与机制分离

- 哪个进程应该被放在内存中哪个位置的决定（策略）是由存储管理器作出的
- 而具体的为进程设置内存映像（机制）的操作是由在内核中的系统任务完成的
- 这个划分使得修改存储管理策略（算法等）比较容易实现，不需要修改操作系统底层。

4.7 MINIX3进程管理器概述

- 内存布局
- 消息处理
- 数据结构与算法
- 创建进程系统调用
- 信号处理系统调用
- 其它系统调用

组合的I和D空间(1)

- 进程所有的部分（代码、数据、和栈）共用一个内存块，它是作为一个整体来申请和释放。

- 在MINIX中有两种情况需要分配内存：
  - 在一个进程执行fork时，为子进程分配所需要的空间；
  - 在一个进程通过EXEC系统调用修改它的内存映象时，老的映象被作为空洞送到空闲表，需要为新的映象分配内存。

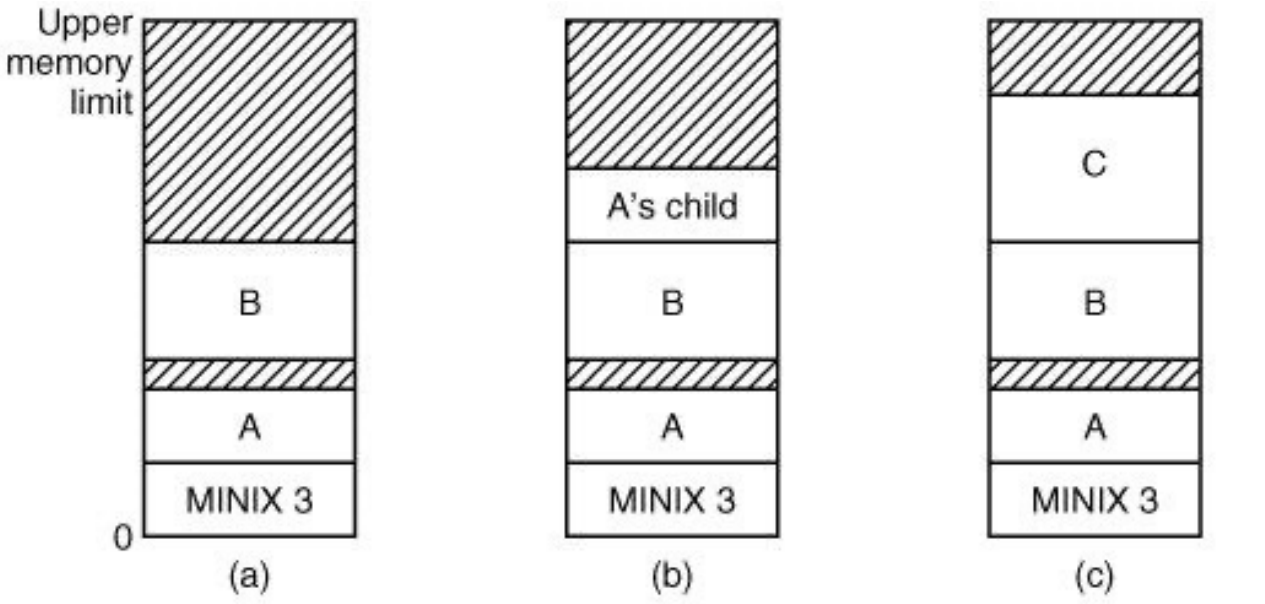
组合的I和D空间(2)

- 进程所有的部分（代码、数据、和堆栈）共用一个内存块，它是作为一个整体来申请和释放。

由fork或exec系统调用引起而分配内存时，一定数量的内存将给予新进程。

在fork情况下，分配的数量和父进程的相同

在exec情况下，内存管理器将分配被执行文件的头部所指明的数量。



独立的I和D空间(默认)

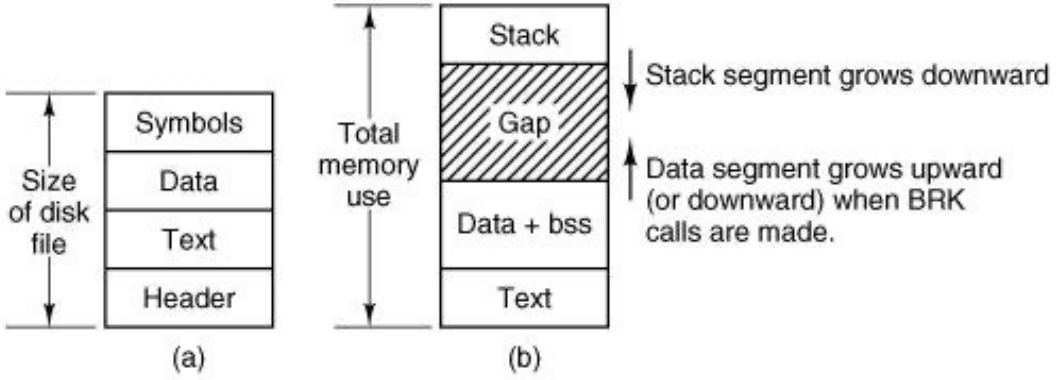
- 当这样的一个进程FORK时，只需要分配为新进程做一个堆栈段和数据段拷贝所需数量的内存。
- 父进程和子进程将共享已经由父进程使用的执行代码，即共享代码
- 当一个这样的进程执行EXEC时，系统将查找进程表看是否有另外一个进程已经在使用需要的执行代码，如果找到了就只为数据和堆栈分配新内存，已经在内存的代码段将被共享。
- 在一个进程结束时它总是要释放它的数据和堆栈占用的内存，但是只有在搜索了进程表并发现没有其他进程在使用代码段后，才释放代码段所占用的内存。
  - 如果一个进程在启动时装入了自己的代码段，而在结束时它的代码正在被一个或多个其他进程共享，那么这个进程启动时分配的内存就会比结束时释放的多。

程序文件及内存布局

- 磁盘文件的头部包含了进程映象各部分的大小以及总的大小的信息。
- 在具有给定的I和D空间的程序头部，有一个域指出代码和数据部分的总长度，这些部分被直接拷贝到内存映象中。
  - 映象中的数据部分增扩了头部bss域指出的数量，扩大的部分被清0，用于未初始化的静态数据。
  - 总共分配的内存数量是由文件头中的total域说明。

若程序(a.out) 的数据和栈段增长所需要的内存总共最多是10K，可以使用下面的命令：chmem = 10240 a.out

这个命令将修改文件头，使得在exec时候存储管理器将为这个程序分配内存空间为：代码段长度+数据段长度+10240字节。



数据段的界限只有通过BRK系统调用才能修改，这个操作涉及的只是当初分配给进程的内存区，操作系统不会分配额外的内存。

程序文件及内存布局

- 对于组合I和D空间，假如一个程序有4K代码段、2K的数据与bss、和1K堆栈，若文件头中说明的需要分配的内存总量是40K(total)，那么在堆栈段和数据段之间的未用内存将是33K。

- 对于使用独立的I和D段的程序(由文件头中连接器设置的一位指出)，文件头中的total域只对结合的数据段和堆栈段有用。一个有4K正文、2K数据、1K堆栈，total域为64K的程序将被分配68K的空间(4K指令空间，64K数据空间)，留出61K空间供数据段和堆栈在运行时使用。

4.7 MINIX3进程管理器概述

- 内存布局
- 消息处理
- 数据结构与算法
- 创建进程系统调用
- 信号处理系统调用
- 其它系统调用

消息处理

- 象MINIX所有其他部分一样，进程管理器是消息驱动的。在系统初始化完成之后，进程管理器就进入它的主循环，包括等待消息、执行消息中包含的请求、和发送应答
  - 内核与系统服务器之间的高优先级通信，采用系统通知消息
  - 来源于用户进程所启动的系统调用(如fork, exit, brk, reboot, ……)

消息

- 来自于用户进程系统调用的消息

Message type	Input parameters	Reply value
fork	(none)	Child's PID, (to child: 0)
exit	Exit status	(No reply if successful)
wait	(none)	Status
waitpid	Process identifier and flags	Status
brk	New size	New size
exec	Pointer to initial stack	(No reply if successful)
kill	Process identifier and signal	Status
alarm	Number of seconds to wait	Residual time
pause	(none)	(No reply if successful)
sigaction	Signal number, action, old action	Status
sigsuspend	Signal mask	(No reply if successful)
sigpending	(none)	Status
sigprocmask	How, set, old set	Status
sigreturn	Context	Status
getuid	(none)	Uid, effective uid
getgid	(none)	Gid, effective gid
getpid	(none)	PID, parent PID
setuid	New uid	Status
setgid	New gid	Status
setsid	New sid	Process group
getpgrp	New gid	Process group
time	Pointer to place where current time	Status

call_vec表

- 用于消息处理的一个关键数据结构是在table.c中call_vec表。
- 它包含了指向处理不同类型消息的过程的指针。当一条消息来到进程管理器时，主循环抽出消息类型，并把它放在全局变量call_nr中，以其值作为call_vec的索引以找到相应的消息处理函数。

```
_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
    no_sys,     /*  0 = unused    */
    do_pm_exit, /*  1 = exit  */
    do_fork,/*  2 = fork  */
    no_sys,     /*  3 = read  */
    no_sys,     /*  4 = write */
    no_sys,     /*  5 = open  */
    no_sys,     /*  6 = close */
    do_waitpid, /*  7 = wait  */
    no_sys,     /*  8 = creat */
    no_sys,     /*  9 = link  */
    no_sys,     /* 10 = unlink    */
    do_waitpid, /* 11 = waitpid   */
    no_sys,     /* 12 = chdir */
    do_time,    /* 13 = time  */
    no_sys,     /* 14 = mknod */
    no_sys,     /* 15 = chmod */
    no_sys,     /* 16 = chown */
    do_brk,     /* 17 = break */
    no_sys,     /* 18 = stat  */
    no_sys,     /* 19 = lseek */
    do_getset,  /* 20 = getpid    */
    no_sys,     /* 21 = mount */
    no_sys,     /* 22 = umount    */
    do_getset,  /* 23 = setuid    */
    do_getset,  /* 24 = getuid    */
    do_stime,   /* 25 = stime */
    do_trace,   /* 26 = ptrace    */
    do_alarm,   /* 27 = alarm */
    no_sys,     /* 28 = fstat */
    do_pause,   /* 29 = pause */
    no_sys,     /* 30 = utime */
    no_sys,     /* 31 = (stty)    */
    no_sys,     /* 32 = (gtty)    */
    no_sys,     /* 33 = access    */
    no_sys,     /* 34 = (nice)    */
    no_sys,     /* 35 = (ftime)   */
    no_sys,     /* 36 = sync  */
    do_kill, /* 37 = kill  */
    no_sys,     /* 38 = rename    */
    no_sys,     /* 39 = mkdir */
    no_sys,     /* 40 = rmdir */
    no_sys,     /* 41 = dup   */
    no_sys,     /* 42 = pipe  */
    do_times,   /* 43 = times */
    no_sys,     /* 44 = (prof)    */
    no_sys,     /* 45 = unused    */
    do_getset,  /* 46 = setgid    */
    do_getset,  /* 47 = getgid    */
    no_sys,     /* 48 = (signal)*/
```

4.7 MINIX3进程管理器概述

- 内存布局
- 消息处理
- 数据结构与算法
- 创建进程系统调用
- 信号处理系统调用
- 其它系统调用

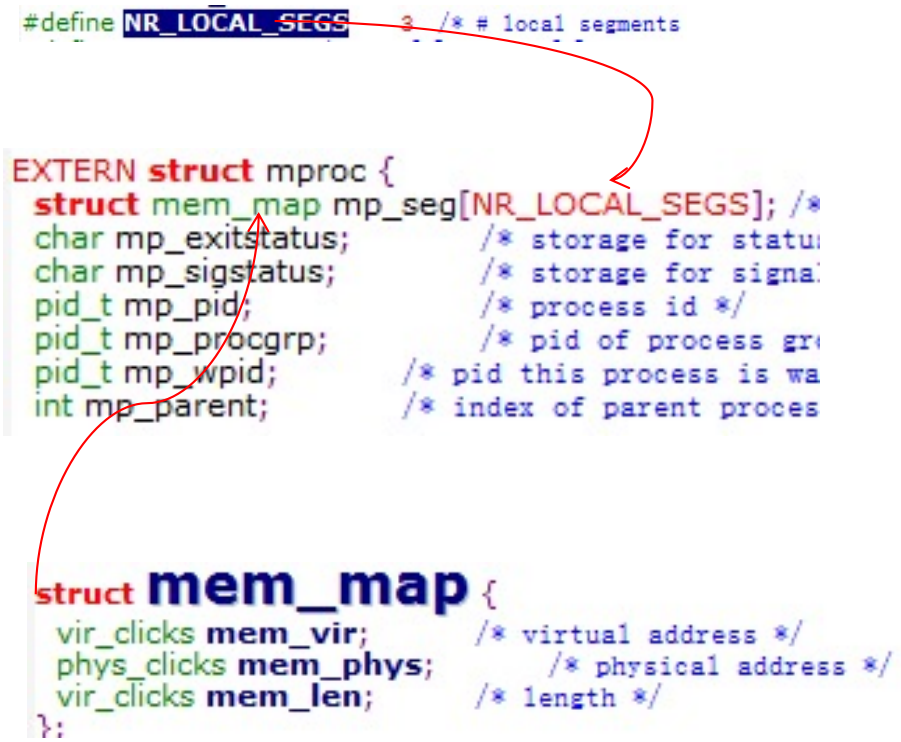
进程表

- 在MINIX3中，进程表(进程控制块)被分三部分，分别为内核、进程管理器和文件管理器所用
- 为了简单起见，表项是精确对应的，因此进程管理器的表项k和文件系统的表项k对应的是同一个进程。为了保持同步，在进程创建或结束时，这三个部分都要更新它们的表以反映新的情况。

Kernel	Process management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Real id
Current scheduling priority	Process ID	Effective UID
Maximum scheduling priority	Parent process	Real GID
Scheduling ticks left	Process group	Effective GID
Quantum size	Children's CPU time	Controlling tty
CPU time used	Real UID	Save area for read/write
Message queue pointers	Effective UID	System call parameters
Pending signal bits	Real GID	Various flag bits
Various flag bits	Effective GID	
Process name	File info for sharing text	
	Bitmaps for signals	
	Various flag bits	
	Process name	

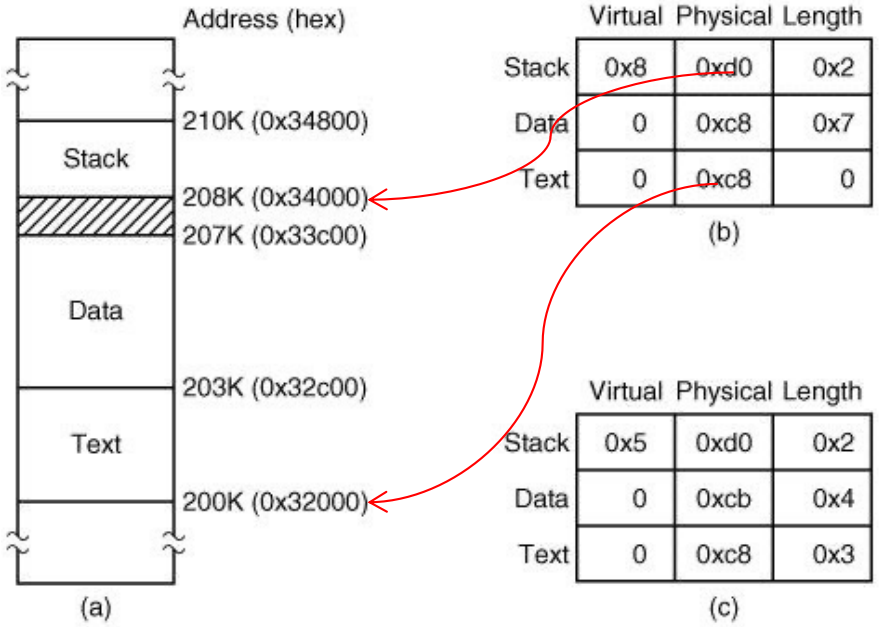
内存中的进程

- PM管理器的进程表叫做mproc，定义在mproc.h中，包含了与进程内存分配有关的全部域和一些附加的信息。
  - 最重要的域是mp_seg数组，它有三个表项，分别用于代码段、数据段、和堆栈段。
  - 各个表项是一个由虚地址、物理地址、和段长度组成的结构。
  - 他们都是用块（click）而不是字节来量度的。在MINIX3中是1024字节。



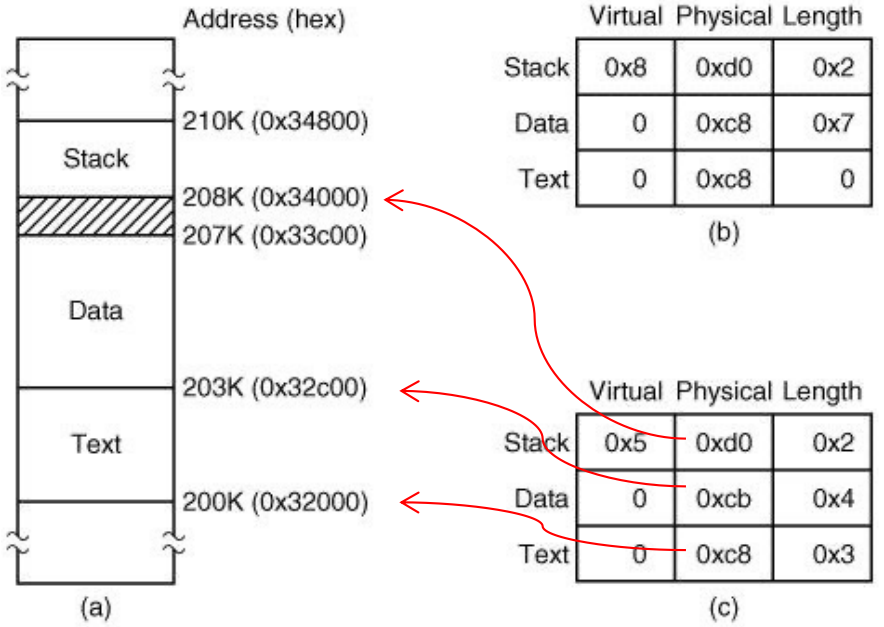
内存分配的记录(1)

- 该图中的一个进程有3K代码、4K数据、1K的空隙、和随后的2K堆栈，分配的全部内存是10K。
- 假设这个进程没有独立的I和D空间，在(b)中看到这三个段各自的虚地址、物理地址、和长度域。在这个模型中，代码段总是空的，数据段包含了代码和数据。
- 当进程引用虚地址0时，不管是跳转到它还是读它（即，在指令空间还是在数据空间），将使用物理地址0x32000（十进制是200K），这个地址位于0xc8个click。



内存分配的记录(2)

- 该图中的一个进程有3K代码、4K数据、1K的空隙、和随后的2K堆栈，分配的全部内存是10K。
- 内存布局在具有独立的I和D空间情况下的段表项如(c)所示。代码段和数据段的长度都不为零。
- mm_seg数组主要是用于把虚地址映射成物理地址。给定一个虚地址和它所属的空间，确定虚地址是否合法（即它是否落在一个段的内部），若合法再确定其对应的物理地址。



栈的虚拟起始地址与进程内存总量

- 栈的起始虚地址取决于分配给进程的内存总量。
- 如果为了提供更大的动态空间（在数据段和堆栈之间更大的间隙），则可以用chmem命令修改了文件头，那么在下一次文件执行时堆栈将从一个更高的虚地址开始。
- 如果堆栈增长了一块，那么堆栈的表项应该从三元组(0x8, 0xd0, 0x2)变成三元组 (0x7, 0xcf, 0x3)(以上页图为例)

数据段与堆栈段越界

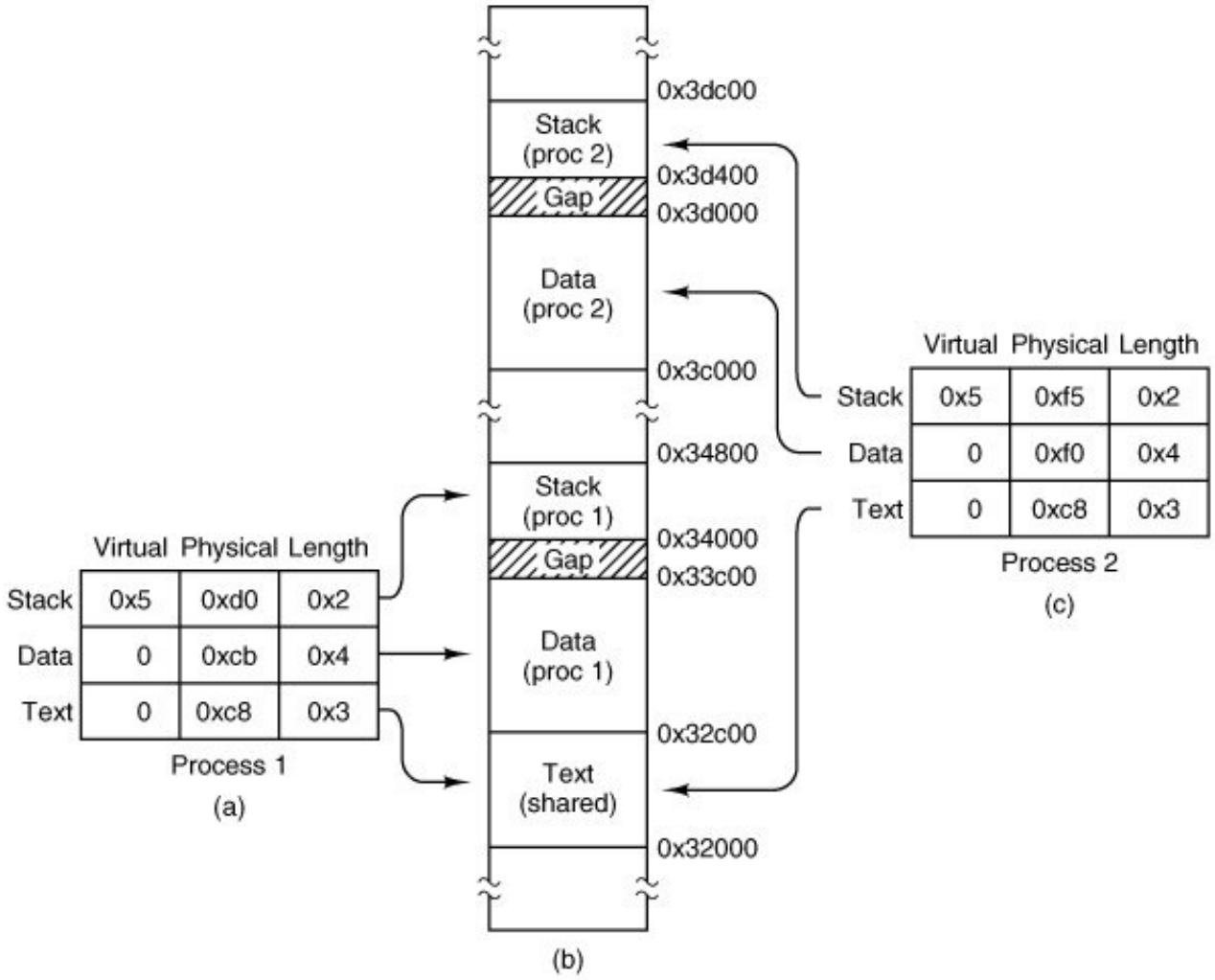
- 尽管处于保护方式的Intel硬件检测超出段界限的内存访问，但是在MINIX中数据段描述符和堆栈段描述符总是一样的。
- MINIX定义的数据和堆栈各自使用同一段(硬件定义的)的不同部分，两者都可以扩展进入位于他们之间的空隙中。
- 在这里，只有MINIX可以管理这些，因为对硬件来说间隙既是数据段也是堆栈段合法的一部分，CPU根本无法检测涉及空隙的错误。

共享代码段

- 一个进程的数据和堆栈区域的内容可能会随着进程的执行而改变，但代码不会改变。几个进程同时执行同一个程序拷贝的情况是很常见的。
- 当EXEC要装入一个程序时，它首先打开保存着该程序磁盘映像的文件并读入文件头。
- 如果进程使用的是独立的I和D空间，系统将搜索每个mproc表项，如果找到了一个正在执行相同程序的进程，那么就没有必要为新进程代码段分配内存，只要把新进程内存映射的mm_seg[T]初始化为指向已经装入代码段的位置。只需为数据段和堆栈段分配相应的内存空间。

例

- (a) The memory map of a separate I and D space process, as in the previous figure.
- (b) The layout in memory after a second process starts, executing the same program image with shared text.
- (c) The memory map of the second process.



空闲链表

```
PRIVATE struct hole {
  struct hole *h_next;      /* pointer to next entry on the list */
  phys_clicks h_base;       /* where does the hole begin? */
  phys_clicks h_len;        /* how big is the hole? */
} hole[NR_HOLES];
```

- 进程管理器另一个重要的数据结构是空闲链表，定义在alloc.c中，它按照内存地址递增的顺序列出了内存中的各个空洞。
- 在空闲链表上的主要操作是分配一块指定大小的内存和回收一块已经分配的内存。
  - 在分配内存时，首先从最低的地址开始搜索空洞表，直到找到一个足够大的空闲区（最先匹配法），并从空闲区中减去需要的空间
  - 在进程结束后，它的数据和堆栈内存区将归还给空闲链表。如果进程使用的是结合的I和D空间，它的全部内存都将被释放；如果进程使用的是独立的I和D空间并且搜索进程表发现没有其他进程在共享其代码，那么它的代码段也将被释放。
  - 对于每个被归还的内存区域，如果它的任何一侧或者两侧邻接的区域也是空闲的，则将他们合并，因此决不会出现邻接的空洞。

4.7 MINIX3进程管理器概述

- 内存布局
- 消息处理
- 数据结构与算法
- 创建进程系统调用
- 信号处理系统调用
- 其它系统调用

FORK、EXIT、和WAIT系统调用(1)

- 在创建和撤消进程时必须分配或释放内存、必须更新进程表，包括由内核和FS保存的部分。这些操作都由进程管理器协调。

- fork系统调用过程
  - Check to see if process table is full.
  - Try to allocate memory for the child's data and stack.
  - Copy the parent's data and stack to the child's memory.
  - Find a free process slot and copy parent's slot to it.
  - Enter child's memory map in process table.
  - Choose a PID for the child.
  - Tell kernel and file system about child.
  - Report child's memory map to kernel.
  - Send reply messages to parent and child.

FORK、EXIT、和WAIT系统调用(2)

- 在下列两个事件都已经发生的情况下进程才会完全终止
  - 进程自己已经退出（或已经被一个信号杀死）
  - 它的父进程已经执行了WAIT系统调用以观察发生了什么。

- 已经退出或被杀死而它的父进程还没有为它执行WAIT的进程将进入某种挂起状态，有时被称为僵死状态(Zombie State)，这种进程不再参与调度，它的报警时钟被关闭（如果原来是开的），但它仍将留在进程表中。它的内存被释放。

- 僵死是一种临时状态，很少会持续较长的时间，当父进程最后执行WAIT时，将释放进程表项，并通知文件系统和内核。

EXEC系统调用(1)

- exec是MINIX中最复杂的系统调用，它用新的内存映像替换当前的内存映像，包括设置新的堆栈
- exec的过程
  - Check permissions is the file executable?
  - Read the header to get the segment and total sizes.
  - Fetch the arguments and environment from the caller.
  - Allocate new memory and release unneeded old memory.
  - Copy stack to new memory image.
  - Copy data (and possibly text) segment to new memory image.
  - Check for and handle setuid, setgid bits.
  - Fix up process table entry.
  - Tell kernel that process is now runnable.

EXEC系统调用(2)

- 在实现过程中，需要考虑可执行文件是否能放得进虚地址空间。
  - 这个问题的原因在于内存是以1024字节的click而不是字节为单位分配的。
  - 由于整个内存管理都是以click为单位的，每个click必须只属于一个段，不允许出现一半是数据、一半堆栈的情形。

EXEC系统调用(3)

- 设置栈的初始状态
  - execve(name, argv, envp);
  - 由execve库函数在它的内部构造完整的初始堆栈，并把它的起始地址和长度传给进程管理器，再复制到用户的进程空间中，并负责重新定位。放在用户进程所分配的内存空间的顶端。
  - 通常执行文件的主程序形式为main(argc, argv, envp);相应地认为三个参数将按C的调用习惯被传递进来，并按此编译访问这三个参数的代码。为此，让程序不从main开始，而是把称为C运行起始函数的一小段汇编语言函数，crtso，链接到代码段的地址0，使它首先被执行。

```
push ecx            ! push environ
push edx            ! push argv
push eax            ! push argc
call _main          ! main(argc, argv, envp)
push eax            ! push exit status
call _exit
hlt                 ! force a trap if exit fails
```

EXEC系统调用(4)

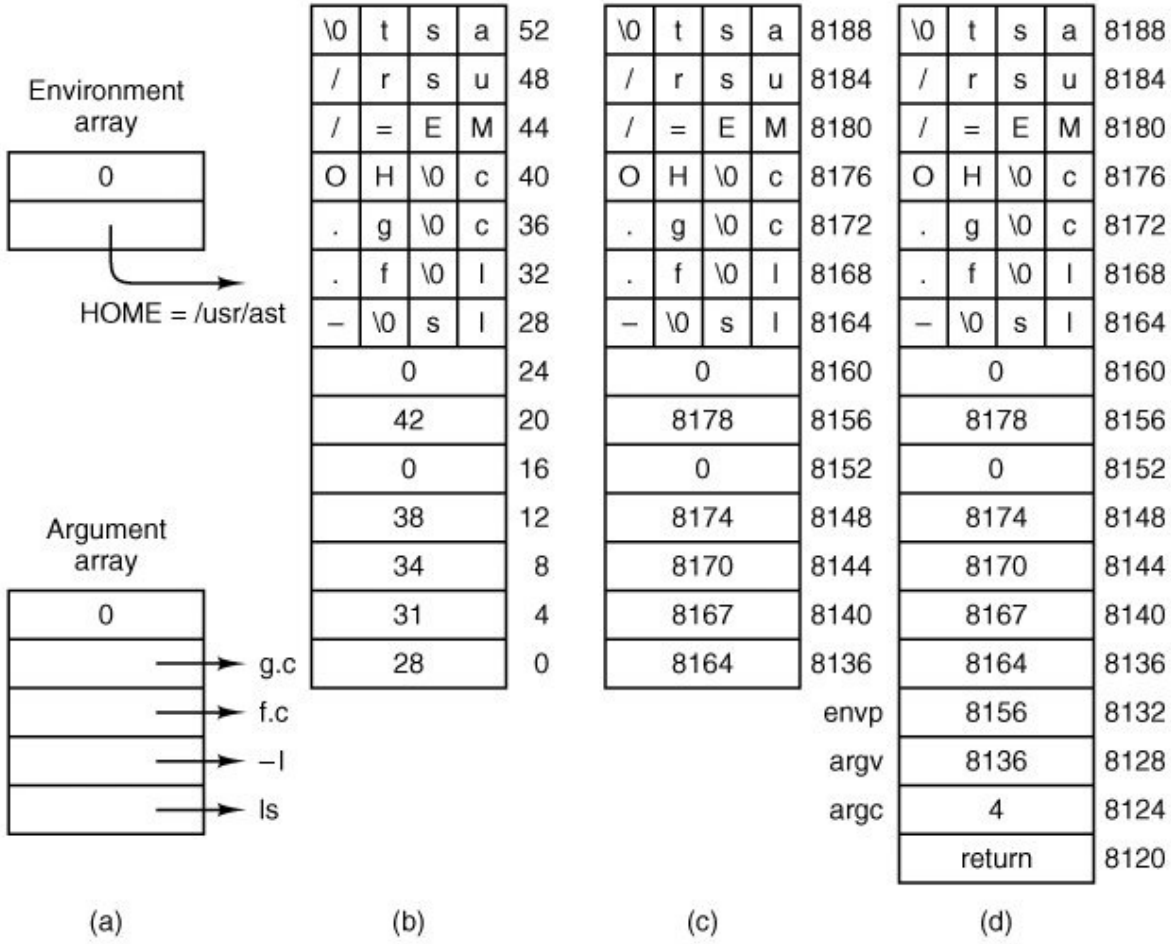
- 设置栈的初始状态

例:

ls –l f.c g.c

execve("/bin/ls", argv, envp);

- (a)传给execve的数组
- (b)在shell地址空间中的函数execve构造的栈
- (c)经PM重定位后的栈
- (d)当main开始执行时的栈



brk系统调用

- 库过程brk和sbrk用来调整数据段的上限。
- 前者的参数是绝对长度（以字节为单位），并以此调用BRK
- 后者的参数是相对于当前长度的正的或负的增量，并由此计算出新的数据段长度，然后调用BRK。
- 执行BRK对内存管理器来说是非常简单的，需要做的全部工作只是检查各个部分是否仍在地址空间中、调整表、并通知内核。

4.7 MINIX3进程管理器概述

- 内存布局
- 消息处理
- 数据结构与算法
- 创建进程系统调用
- 信号处理系统调用
- 其它系统调用

信号处理过程

- 在系统中定义了一组信号，每个信号都有一个默认的动作：Kill进程或忽略信号
- 亦可以通过系统调用改变信号响应方式，即将一个信号处理函数与特定信号绑定
  - 当信号发生时，就会由相应的信号处理函数去执行

- 信号处理三阶段
  - 准备阶段: program code prepares for possible signal.
  - 响应阶段: signal is received and action is taken.
  - 清理阶段: restore normal operation of the process.

信号处理的准备阶段(1)

- 在准备阶段可以调用系统调用以修改它对信号的响应。
  - 其中最通用的是SIGACTION，用它可以说明进程将忽略某些信号、捕获某些信号（用执行位于进程内部、用户定义的信号处理程序替换缺省处理）、或者恢复某些信号的缺省处理。
  - 另一个系统调用SIGPROCMASK可以阻塞一个信号，它使得一个信号被暂时保存起来，只有当进程在后来某个时候解除对这个信号的阻塞时才会响应这个信号。

信号处理的准备阶段(2)

- 在MINIX3中，信号处理的准备阶段完全由进程管理器处理。
  - 对于每个进程，都有几个sigset_t变量，每个可能的信号由他们中的一个位表示。一个这样的变量定义要忽略的信号集，另一个定义要捕获的信号集，依此类推。
  - 每个进程还都有一个sigaction结构的数组，每个信号对应一个数组元素
    - 每个sigaction结构元素：一个变量保存相应信号指定处理过程的地址，另一个变量sigset_t用于指定在该处理过程执行时应阻塞的信号

```
struct sigaction {
    __sighandler_t sa_handler;    /* SIG_DFL, SIG_IGN, SIG_MESS,
                                     or pointer to function */
  sigset_t sa_mask;               /* signals to be blocked during handler */
  int sa_flags;                   /* special flags */
}
```

信号处理的响应阶段

- 在信号发生时，进程管理器开始确定哪个进程应该得到这个信号。
  - 如果信号应该被捕获，就必须把它传递给目的进程，这需要保存有关进程状态的信息以使进程可以恢复正常运行。
  - 如果信号不需要捕获，缺省的动作将被执行，这可能涉及调用文件系统生成一个core文件（把进程的映像写到一个文件里，它可能会被调试器检查），同时终止进程，这涉及到进程管理器、文件系统、内核。
  - 一个信号可能需要传送给一组进程，所以内存管理器可能使这些动作重复多次。

捕获信号处理过程

- 如果信号应该被捕获，就必须把它传递给目的进程，这需要保存有关进程状态的信息以使进程可以恢复正常运行。
- 这些信息被保存在接收信号进程的堆栈上，因此必须检查是否有足够的堆栈空间。
- 随后内存管理器调用在内核中的系统任务把这些信息放到堆栈上。
- 系统任务还处理进程的程序计数器，使进程能够执行信号处理过程的代码。
- 在信号处理过程结束时将执行一个SIGRETURN系统调用，通过这个调用内存管理器和内核共同恢复进程的信号上下文和寄存器，使进程可以恢复正常运行。

MINIX3 信号类型

- MINIX3中的信号定义在/usr/include/signal.h中，POSIX标准所要求的文件之一。

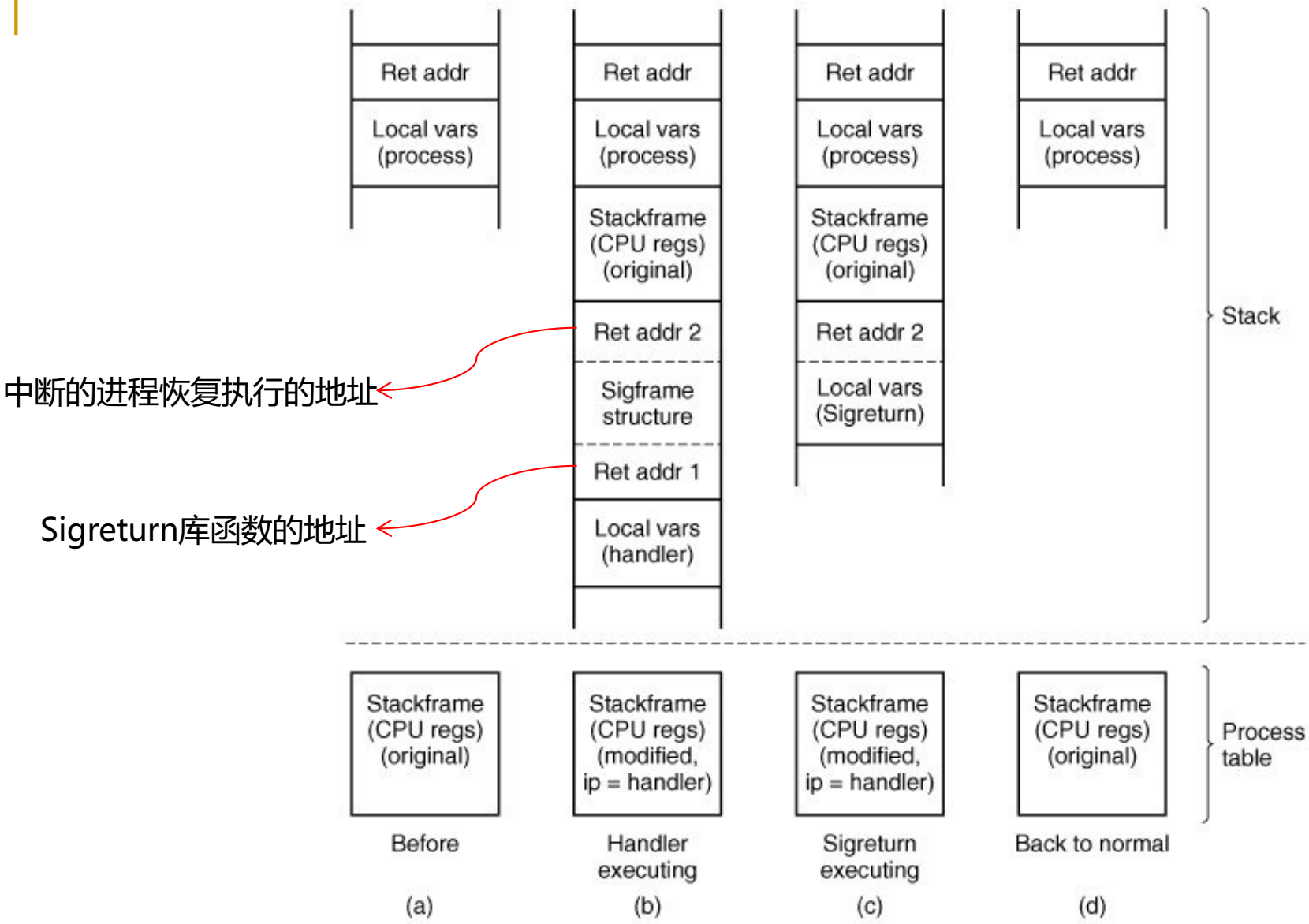
- 信号可以通过两种方式产生：
  - KILL系统调用
  - 内核产生

Signal	Description	Generated by
SIGHUP	Hangup	KILL system call
SIGINT	Interrupt	TTY
SIGQUIT	Quit	TTY
SIGILL	Illegal instruction	Kernel (*)
SIGTRAP	Trace trap	Kernel (M)
SIGABRT	Abnormal termination	TTY
SIGFPE	Floating point exception	Kernel (*)
SIGKILL	Kill (cannot be caught or ignored)	KILL system call
SIGUSR1	User-defined signal # 1	Not supported
SIGSEGV	Segmentation violation	Kernel (*)
SIGUSR2	User defined signal # 2	Not supported
SIGPIPE	Write on a pipe with no one to read it	FS
SIGALRM	Alarm clock, timeout	PM
SIGTERM	Software termination signal from kill	KILL system call
SIGCHLD	Child process terminated or stopped	PM
SIGCONT	Continue if stopped	Not supported
SIGSTOP	Stop signal	Not supported
SIGTSTP	Interactive stop signal	Not supported
SIGTTIN	Background process wants to read	Not supported
SIGTTOU	Background process wants to write	Not supported
SIGKMESS	Kernel message	Kernel
SIGKSIG	Kernel signal pending	Kernel
SIGKSTOP	Kernel shutting down	Kernel

信号处理的清理阶段

- 在信号处理过程结束时进程应该象什么都没有发生一样继续执行，需要记录CPU所有寄存器当前值。
- 进程表由于空间有限，只能保存一份副本，存放将进程恢复到原来状态所需的所有CPU寄存器的内容。

- 解决这个问题的方法
  - 进程在中断发生后，CPU所有的寄存器都被拷贝到进程表中的栈帧中
  - 在准备处理阶段，进程表中的栈桢被复制到进程自己的栈中，以保存
  - 信号处理过程是一个普通的过程，在它结束时SIGRETURN将被执行。
  - SIGRETURN的工作是把各个部分恢复成他们接收信号以前的状态，并进行清理。最重要的是通过使用保存在接收信号进程堆栈中的拷贝，进程表中的栈框被恢复到它原来的状态。


Ret addr
Local vars (process)
Stackframe (CPU regs) (original)
Ret addr 2
Sigframe structure
Ret addr 1
Local vars (handler)
Local vars (Sigreturn)
Stack
中断的进程恢复执行的地址
Sigreturn库函数的地址
Stackframe (CPU regs) (original)
Stackframe (CPU regs) (modified, ip = handler)
Process table
Before
Handler executing
Sigreturn executing
Back to normal
(a)
(b)
(c)
(d)

用户空间定时器

- MIINIX3中内核空间的定时器仅用于系统进程，用户进程的定时器由进程管理器来维护
- 进程管理器维护一个定时器队列，每次根据队列头部的定时器向时钟任务发出警报请求
- 在一个时钟中断后，若系统检测到一个到期的警报，进程管理器会收到一个通知，然后，它会检查自己的定时器队列，并向相应的用户进程发送信号。

4.7 MINIX3进程管理器概述

- 内存布局
- 消息处理
- 数据结构与算法
- 创建进程系统调用
- 信号处理系统调用
- 其它系统调用

其它系统调用

- 进程管理器还处理一些较简单的系统调用
  - time, stime, times,
  - getuid ， geteuid, ……
  - ptrace, reboot

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

头文件和数据结构

- pm.h
- glo.h
- mproc.h
- table.c -- call_vec数据结构
- ……

主程序

- 进程管理器是独立于内核和文件系统编译和连接的，所以它有自己的主程序。

- 它的主程序在内核初始化自己之后被启动，主程序在main.c中（18041行），在通过调用pm_init完成自己的初始化之后，进程管理器进入18051行的循环。

主程序

```
PUBLIC int main()
{
/* Main routine of the process manager. */
 int result, s, proc_nr;
 struct mproc *rmp;
 sigset_t sigset;

 pm_init();            /* initialize process manager tables */

 /* This is PM's main loop-  get work and do it, forever and forever. */
 while (TRUE) {
    get_work();        /* wait for an PM system call */

    /* Check for system notifications first. Special cases. */
    if (call_nr == SYN_ALARM) {
        pm_expire_timers(m_in.NOTIFY_TIMESTAMP);
        result = SUSPEND;       /* don't reply */
    } else if (call_nr == SYS_SIG) {     /* signals pending */
        sigset = m_in.NOTIFY_ARG;
        if (sigismember(&sigset, SIGKSIG)) (void) ksig_pending();
        result = SUSPEND;       /* don't reply */
    }
    /* Else, if the system call number is valid, perform the call. */
    else if ((unsigned) call_nr >= NCALLS) {
        result = ENOSYS;
    } else {
        result = (*call_vec[call_nr])();
    }

    /* Send the results back to the user to indicate completion. */
    if (result != SUSPEND) setreply(who, result);

    swap_in();          /* maybe a process can be swapped in? */

    /* Send out all pending reply messages, including the answer to
     * the call just made above.  The processes must not be swapped out.
     */
    for (proc_nr=0, rmp=mproc; proc_nr < NR_PROCS; proc_nr++, rmp++) {
        /* In the meantime, the process may have been killed by a
         * signal (e.g. if a lethal pending signal was unblocked)
         * without the PM realizing it.  If the slot is no longer in
         * use or just a zombie, don't try to reply.
         */
        if ((rmp->mp_flags & (REPLY | ONSWAP | IN_USE | ZOMBIE)) ==
          (REPLY | IN_USE)) {
            if ((s=send(proc_nr, &rmp->mp_reply)) != OK) {
                panic(__FILE__,"PM can't reply to", proc_nr);
            }
            rmp->mp_flags &= ~REPLY;
```

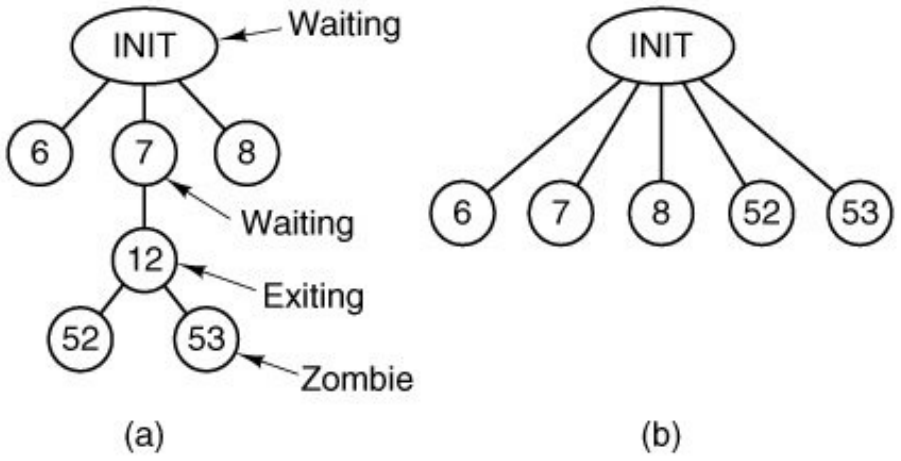
```
PRIVATE void get_work()
{
/* Wait for the next message and extract useful information from it. */
 if (receive(ANY, &m_in) != OK) panic(__FILE__,"PM receive error", NO_NUM);
 who = m_in.m_source;          /* who sent the message */
 call_nr = m_in.m_type;    /* system call number */

 /* Process slot of caller. Misuse PM's own process slot if the kernel is
  * calling. This can happen in case of synchronous alarms (CLOCK) or or
  * event like pending kernel signals (SYSTEM).
  */
 mp = &mproc[who < 0 ? PM_PROC_NR : who];
}
```

```
#define swap_in()          ((void)0)
```

默认情况下，交换模块没有被包含进来，函数没有实质的内容。若编译系统中选中，将在此会进行一个测试，判断进程能否被交换。

fork, exit, wait的实现

- fork, exit和wait系统调用是由文件forkexit.c中的do_fork、do_pm_exit、do_waitpid函数实现




```
PUBLIC int do_pm_exit()
{
/* Perform the exit(status) system call. The real work is done by pm_exit().
 * which is also called when a process is killed by a signal.
 */
 pm_exit(mp, m_in.status);
 return(SUSPEND);          /* can't communicate from beyond the grave */
}
```

exec的实现

- 由exec.c中的do_exec函数实现
- 若映像是可执行文件，则用其替换当前的内存映像
- 若映像是脚本文件，则装入解释程序的二进制文件，并将该脚本文件的文件名作为其参数

```
PUBLIC int do_exec()
{
/* Perform the execve(name, argv, envp) call.  The user library builds a
 * complete stack image, including pointers, args, environ, etc.  The stack
 * is copied to a buffer inside PM, and then to the new core image.
 */
 register struct mproc *rmp;
 struct mproc *sh_mp;
 int m, r, fd, ft, sn;
 static char mbuf[ARG_MAX];    /* buffer for stack and zeroes */
 static char name_buf[PATH_MAX]; /* the name of the file to exec */
 char *new_sp, *name, *basename;
 vir_bytes src, dst, text_bytes, data_bytes, bss_bytes, stk_bytes, vsp;
 phys_bytes tot_bytes;         /* total space for program, including gap */
 long sym_bytes;
 vir_clicks sc;
 struct stat s_buf[2], *s_p;
 vir_bytes pc;
```

brk的实现

- 当进程创建时，将获得一块用于存放数据和栈的连续内存区域，该区域是固定的
- 数据段和栈段可以共享其中的空隙
- brk用于调整数据段大小，在break.c中do_brk实现
- 首先检查新的大小是否可能，然后更新相关的表

```
PUBLIC int do_brk()
{
/* Perform the brk(addr) system call.
 *
 * The call is complicated by the fact that on some machines (e.g., 8088),
 * the stack pointer can grow beyond the base of the stack segment without
 * anybody noticing it.
 * The parameter, 'addr' is the new virtual address in D space.
 */

 register struct mproc *rmp;
 int r;
 vir_bytes v, new_sp;
 vir_clicks new_clicks;

 rmp = mp;
 v = (vir_bytes) m_in.addr;
 new_clicks = (vir_clicks) ( ((long) v + CLICK_SIZE - 1) >> CLICK_SHIFT);
 if (new_clicks < rmp->mp_seg[D].mem_vir) {
     rmp->mp_reply.reply_ptr = (char *) -1;
     return(ENOMEM);
 }
 new_clicks -= rmp->mp_seg[D].mem_vir;
 if ((r=get_stack_ptr(who, &new_sp)) != OK)     /* ask kernel for sp value */
     panic(__FILE__,"couldn't get stack pointer", r);
 r = adjust(rmp, new_clicks, new_sp);
 rmp->mp_reply.reply_ptr = (r == OK ? m_in.addr : (char *) -1);
 return(r);                /* return new address or -1 */
} ? end do_brk ?
```

信号处理实现

- 与信号处理相关的系统调用8个，均在signal.c中实现
  - do_xxxxx

System call	Purpose
sigaction	Modify response to future signal
sigprocmask	Change set of blocked signals
kill	Send signal to another process
alarm	Send ALRM signal to self after delay
pause	Suspend self until future signal
sigsuspend	Change set of blocked signals, then PAUSE
sigpending	Examine set of pending (blocked) signals
sigreturn	Clean up after signal handler

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

作业(OS设计与实现)

- 2, 6, 11, 12, 14, 18, 23, 24, 27, 30